

Aprende **OpenXava** con ejemplos

JPA

JUnit

Liferay

Hibernate Validator

PostgreSQL

Javier Paniza

Eclipse

HtmlUnit

*Aprende
OpenXava
con ejemplos*

EDICIÓN 1.1

© 2011 *Javier Paniza*

Sobre el libro

El propósito de este libro es enseñarte a desarrollar aplicaciones Java Web de gestión usando OpenXava y otras tecnologías, herramientas y marcos de trabajo relacionados. El camino para conseguir este objetivo va a ser desarrollar una aplicación desde cero, paso a paso, hasta obtener una aplicación de gestión completamente funcional.

¿Por qué un libro sobre OpenXava? Ya tenemos un montón de documentación gratuita. Sí, es cierto. De hecho, durante algún tiempo yo mismo pensaba que la documentación oficial de OpenXava, que es bastante exhaustiva y siempre está actualizada, era suficiente para aprender OpenXava. Sin embargo, algunos miembros de la comunidad OpenXava me pedían un libro, y esto hizo que empezara a pensar que lo de escribir un libro no era tan mala idea al fin y al cabo.

La documentación de OpenXava describe con precisión toda la sintaxis de OpenXava junto con su semántica. Sin duda, es una herramienta esencial que permite conocer todos los secretos de OpenXava. Pero, si eres principiante en el mundo *Java Enterprise* puede ser que necesites algo más que una documentación de referencia. Querrás tener mucho código de ejemplo y también aprender las demás tecnologías necesarias para crear una aplicación final.

En este libro aprenderás, no solo OpenXava, sino también JPA, Eclipse, PostgreSQL, JUnit, HtmlUnit, Hibernate Validator framework, Liferay, etc. Y lo más importante, vas a aprender técnicas para resolver casos comunes y avanzados a los que te enfrentarás al desarrollar aplicaciones de gestión.

No dudes en contactar conmigo para darme tu opinión y sugerencias sobre el libro. Hazlo en javierpaniza@yahoo.com.

Si encuentras errores gramaticales u ortográficos te devuelvo el dinero¹.

Gracias especiales para Emiliano Torres, César Llave Llerena y Gerardo Gómez Caminero por sus correcciones.

¹ Te pagaría el precio original en euros vía PayPal. Has de proporcionarme las correcciones correspondientes. Esta oferta solo aplica a la última edición eBook publicada

Contenido

Capítulo 1: Arquitectura y filosofía	1
1.1 Los conceptos	2
Desarrollo Dirigido por el Modelo Ligero	2
Componente de Negocio	3
1.2 Arquitectura de la aplicación	6
Perspectiva del desarrollador de aplicaciones	6
Perspectiva del usuario	7
Estructura del proyecto	8
1.3 Flexibilidad	9
Editores	9
Vista personalizada	10
1.4 Resumen	10
Capítulo 2: Java Persistence API	13
2.1 Anotaciones JPA	14
Entidad	14
Propiedades	15
Referencias	17
Clases incrustables	18
Colecciones	19
2.2 API JPA	21
2.3 Resumen	23
Capítulo 3: Anotaciones	25
3.1 Validación	26
Validación declarativa	26
Validaciones predefinidas	26
Validación propia	28
Aprende más sobre validaciones	29
3.2 Interfaz de usuario	30
La interfaz de usuario por defecto	30
La anotación @View	31
Refinando la presentación de los miembros	32
Aprende más acerca de la interfaz de usuario	35
3.3 Otras anotaciones	36
3.4 Resumen	36

Capítulo 4: Empezando con Eclipse y PostgreSQL	37
4.1 Nuestro objetivo: Una pequeña aplicación de facturación	38
4.2 Instalar PostgreSQL	38
4.3 Crear el proyecto en Eclipse	40
Instalar OpenXava	41
Crear el proyecto	41
Configurar Tomcat dentro de Eclipse	43
Crear tu primera entidad	45
4.4 Preparar la base de datos	47
Configurar persistence.xml	47
Actualizar esquema	48
4.5 Ejecutar la aplicación	50
4.6 Modificar la aplicación	51
4.7 Acceder a la base de dato desde Eclipse	52
4.8 Resumen	54
Capítulo 5: Modelar con Java	55
5.1 Modelo básico del dominio	56
Referencia (ManyToOne) como lista de descripciones (combo)	56
Estereotipos	58
Embeddable	59
Clave compuesta	61
Calcular valores por defecto	63
Referencias convencionales (ManyToOne)	65
Colección de entidades dependientes (ManyToOne con cascade)	66
5.2 Refinar la interfaz de usuario	69
Interfaz de usuario por defecto	69
Usar @View para definir la disposición	70
Usar @ReferenceView para refinar la interfaz de referencias	70
Refinar la introducción de elementos de colección	71
La interfaz de usuario refinada	73
5.3 Desarrollo ágil	74
5.4 Resumen	77
Capítulo 6: Pruebas automáticas	79
6.1 JUnit	80
6.2 ModuleTestBase para probar módulos	80
El código para la prueba	80
Ejecutar las pruebas desde Eclipse	83

6.3 Añadir el controlador JDBC al path de Java	84
6.4 Crear datos de prueba usando JPA	85
Los métodos setUp() y tearDown()	85
Crear datos con JPA	86
Borrar datos con JPA	88
Filtrar datos desde modo lista en una prueba	88
Usar instancias de entidad dentro de una prueba	89
6.5 Usar datos ya existentes para probar	90
6.6 Probar colecciones	91
Dividir la prueba en varios métodos	92
Verificar valores por defecto	93
Entrada de datos	94
Verificar los datos	95
6.7 Suite	96
6.8 Resumen	97
Capítulo 7: Herencia	99
7.1 Heredar de una superclase mapeada	100
7.2 Herencia de entidades	102
Nueva entidad Order	102
CommercialDocument como entidad abstracta	103
Invoice refactorizada para usar herencia	104
Crear Order usando herencia	105
Convención de nombres y herencia	106
Asociar Order con Invoice	106
7.3 Herencia de vistas	107
El atributo extendsView	107
Vista para Invoice usando herencia	108
Vista para Order usando herencia	110
Usar @ReferenceView y @CollectionView para refinar vistas	111
7.4 Herencia en las pruebas JUnit	113
Crear una prueba de módulo abstracta	113
Prueba base abstracta para crear pruebas de módulo concretas	115
Añadir nuevas pruebas a las pruebas de módulo extendidas	116
7.5 Resumen	117
Capítulo 8: Lógica de negocio básica	119
8.1 Propiedades calculadas	120
Propiedad calculada simple	120
Usar @DefaultValueCalculator	121

Propiedades calculadas dependientes de una colección	124
Valor por defecto desde un archivo de propiedades	126
8.2 Métodos de retrollamada JPA	128
Cálculo de valor por defecto multiusuario	128
Sincronizar propiedades persistentes y calculadas	130
8.3 Lógica desde la base de datos (@Formula)	132
8.4 Pruebas JUnit	134
Modificar la prueba existente	134
Verificar valores por defecto y propiedades calculadas	136
Sincronización entre propiedad persistente y calculada / @Formula	137
8.5 Resumen	139
Capítulo 9: Validación avanzada	141
9.1 Alternativas de validación	142
Añadir la propiedad delivered a Order	142
Validar con @EntityValidator	143
Validar con métodos de retrollamada JPA	145
Validar en el setter	145
Validar con Hibernate Validator	146
Validar al borrar con @RemoveValidator	146
Validar al borrar con un método de retrollamada JPA	148
¿Cuál es la mejor forma de validar?	148
9.2 Crear tu propia anotación de Hibernate Validator	149
Usar un Hibernate Validator en tu entidad	149
Definir tu propia anotación ISBN	150
Usar Apache Commons Validator para implementar la lógica	150
Llamar a un servicio web REST para validar el ISBN	152
Añadir atributos a tu anotación	154
9.3 Pruebas JUnit	155
Probar la validación al añadir a una colección	155
Probar validación al asignar una referencia y al borrar	157
Probar el Hibernate Validator propio	158
9.4 Resumen	159
Capítulo 10: Refinar el comportamiento predefinido	161
10.1 Acciones personalizadas	162
Controlador Typical	162
Refinar el controlador para un módulo	164
Escribir tu propia acción	165
10.2 Acciones genéricas	168

Código genérico con MapFacade	168
Cambiar el controlador por defecto para todos los módulos	170
Volvamos un momento al modelo	171
Metadatos para un código más genérico	173
Acciones encadenadas	174
Refinar la acción de búsqueda por defecto	175
10.3 Modo lista	179
Filtrar datos tabulares	180
Acciones de lista	180
10.4 Reutilizar el código de las acciones	183
Propiedades para crear acciones reutilizables	183
Módulos personalizados	184
Varias definiciones de datos tabulares por entidad	185
Obsesión por reutilizar	186
10.5 Pruebas JUnit	187
Probar el comportamiento personalizado para borrar	187
Probar varios módulos en el mismo método de prueba	188
10.6 Resumen	191
Capítulo 11: Comportamiento y lógica de negocio	193
11.1 Lógica de negocio desde el modo detalle	194
Crear una acción para ejecutar lógica personalizada	195
Escribiendo la lógica de negocio real en la entidad	196
Escribe menos código usando Apache Commons BeanUtils	197
Copiar una colección de entidad a entidad	198
Excepciones de aplicación	199
Validar desde la acción	200
Evento OnChange para ocultar/mostrar una acción por código	202
11.2 Lógica de negocio desde el modo lista	205
Acción de lista con lógica propia	205
Lógica de negocio en el modelo sobre varias entidades	207
11.3 Cambiar de módulo	209
Uso de IChangeModuleAction	210
Módulo de solo detalle	211
Volviendo al módulo que llamó	212
Objeto de sesión global y acción on-init	213
11.4 Pruebas JUnit	216
Probar la acción de modo detalle	216
Buscar una entidad para la prueba usando el modo lista y JPA	218
Probar que la acción se oculta cuando no aplica	219

Probar la acción de modo lista	220
Verificar datos de prueba	222
Probar casos excepcionales	222
11.5 Resumen	223
Capítulo 12: Referencias y colecciones	225
12.1 Refinar el comportamiento de las referencias	226
Las validaciones están bien, pero no son suficientes	226
Modificar los datos tabulares por defecto ayuda	227
Refinar la acción para buscar una referencia con una lista	228
Buscar la referencia tecleando en los campos	230
Refinar la acción para buscar cuando se teclea la clave	231
12.2 Refinar el comportamiento de las colecciones	234
Modificar los datos tabulares ayuda	234
Refinar la lista para añadir elementos a la colección	235
Refinar la acción que añade elementos a la colección	237
12.3 Pruebas JUnit	240
Adaptar OrderTest	240
Probar @SearchAction	241
Probar @OnChangeListener	242
Adaptar InvoiceTest	243
Probar @NewAction	244
Probar la acción para añadir elementos a la colección	247
12.4 Resumen	249
Capítulo 13: Seguridad y navegación con Liferay	251
13.1 Introducción a los portales Java	252
13.2 Instalar Liferay	253
13.3 Desplegar nuestra aplicación en Liferay	254
13.4 Navegación	256
Añadir las páginas para la aplicación y sus módulos	256
Llenar una página vacía con un Portlet	258
Añadir un menú de navegación	258
Copiar una página	259
Dos módulos en la misma página	260
Usar el CMS	262
13.5 Gestión de usuarios	264
Roles	264
Usuarios	265
13.6 Niveles de acceso	267

Limitar el acceso a toda la aplicación	267
Limitar el acceso a módulos individuales	268
Limitar funcionalidad	269
Limitar la visibilidad de los datos en modo lista	270
Limitar la visibilidad de los datos en modo detalle	272
13.7 Pruebas JUnit	274
13.8 Resumen	276
Apéndice A: Código fuente	277
Apéndice B: Aprender más	325

*Arquitectura
y filosofía*

capítulo 1

OpenXava es un marco de trabajo para desarrollo rápido de aplicaciones de gestión con Java. Es fácil de aprender y rápido para desarrollar. Al mismo tiempo es extensible y personalizable, además el código de la aplicación se estructura desde un punto de vista orientado a objetos puro. Por lo tanto, puedes enfrentarte a aplicaciones complejas con él.

La aproximación de OpenXava al desarrollo rápido no es por medio de usar entornos visuales (como Visual Basic o Delphi), o *scripting*, como PHP. Más bien, el enfoque de OpenXava es dirigido por el modelo (*model-driven*), donde el corazón de tu aplicación son clases Java que describen tu problema. De esta forma conseguimos productividad sin utilizar código spaghetti.

Este capítulo mostrará los conceptos en los que se fundamenta OpenXava y también una visión general de su arquitectura.

1.1 Los conceptos

Aunque OpenXava tiene una visión muy pragmática del desarrollo, está basado en un refinamiento de conceptos preexistentes, algunos populares y otros no tanto. El más popular es el Desarrollo Dirigido por el Modelo (*Model-Driven Development*, MDD), que OpenXava usa de una manera ligera. El otro concepto, el Componente de Negocio, es raíz y principio básico de OpenXava, además de ser la alternativa opuesta a MVC.

Veamos estos conceptos con más detalles.

1.1.1 Desarrollo Dirigido por el Modelo Ligero

Básicamente, MDD establece que únicamente se ha de desarrollar la parte del modelo de una aplicación, y el resto se generará a partir de este modelo. Tal y como se ve en la figura 1.1.



Figura 1.1 Desarrollo Dirigido por el Modelo

En el contexto de MDD el modelo es el medio para representar los datos y la

3 Capítulo 1: Arquitectura y filosofía

lógica de la aplicación. Puede ser, bien mediante una notación gráfica, como UML, o bien mediante una notación textual como un Lenguaje Específico del Dominio (*Domain-Specific Language*, DSL).

Por desgracia, el uso de MDD es muy complejo. Requiere de una gran cantidad de tiempo, pericia y herramientas². Aun así la idea tras MDD sigue siendo muy buena, por lo tanto OpenXava toma esa idea de una manera simplificada. Usa simples clases de Java con anotaciones para definir el modelo, y no usa generación de código, en vez de eso toda la funcionalidad de la aplicación es generada dinámicamente en tiempo de ejecución (tabla 1.1).

	Definición del modelo	Generación de la aplicación
MDD clásico	UML/DSL	Generación de código
OpenXava	Simple clases Java	Dinámicamente en tiempo de ejecución

Tabla 1.1 MDD / OpenXava comparison

Podemos decir pues, que OpenXava es un Marco de trabajo Ligero Dirigido por el Modelo. La figura 1.2 lo muestra.

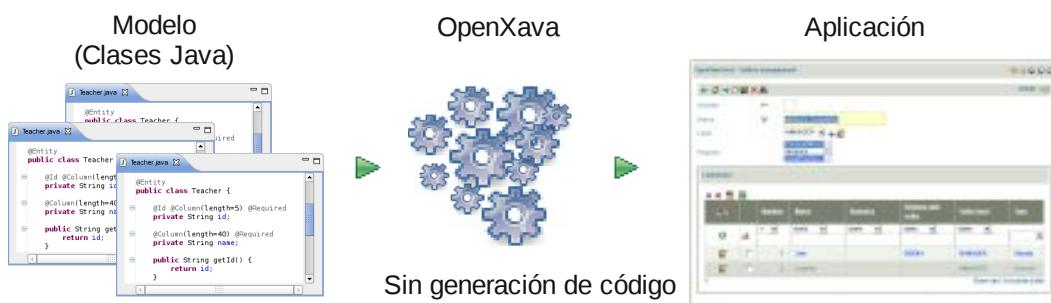


Figure 1.2 Desarrollo Dirigido por el Modelo Ligero en OpenXava

A partir de clases Java simples obtienes una aplicación lista para usar. La siguiente sección sobre el concepto de Componente de Negocio revelará algunos detalles importantes sobre la naturaleza de estas clases.

1.1.2 Componente de Negocio

Un Componente de Negocio consiste en todos los artefactos de software relacionados con un concepto de negocio. Los componentes de negocio son tan solo una forma de organizar el software. La otra forma de organizar software es MVC (Model-View Controller), donde clasificas el código por datos (modelo), interfaz de usuario (vista) y lógica (controlador).

2 See the Better Software with Less Code white paper (<http://www.lulu.com/content/6544428>)

La figura 1.3 muestra como se organizan los artefactos de software en una aplicación MVC. Todos los artefactos para la interfaz de usuario de la aplicación, tales como páginas JSP, JSF, Swing, código JavaFX, etc. están en el mismo lugar, la capa de la vista. Lo mismo ocurre para el modelo y el controlador. Esto contrasta con una arquitectura basada en componentes de negocio donde los artefactos de software se organizan alrededor de los conceptos de negocio.

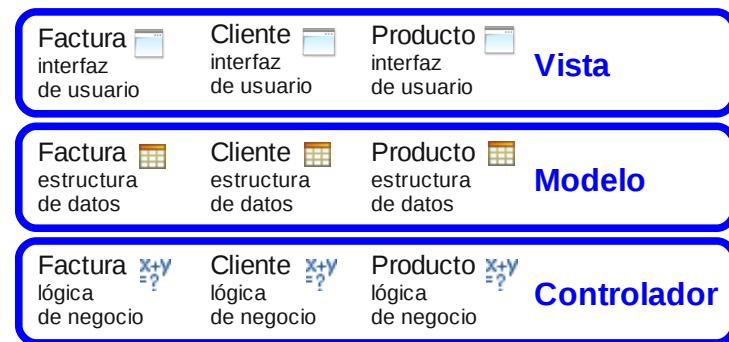


Figura 1.3 Enfoque Modelo Vista Controlador

Puedes verlo en la figura 1.4. Aquí, todos los artefactos de software acerca del concepto de Factura, como la interfaz de usuario, acceso a base de datos, lógica de negocio, etc. están en un mismo lugar.



Figura 1.4 Enfoque Componente de Negocio

¿Qué enfoque es mejor? Eso depende de tus necesidades.

Si tienes que cambiar frecuentemente la estructura de los datos y la lógica de negocio entonces la opción de los componentes de negocio es muy práctica, porque todas las cosas que necesitas tocar cuando haces un cambio están en el mismo sitio y no esparcidas por multitud de archivos.

La pieza básica para desarrollar aplicaciones OpenXava es el componente de negocio, y la forma de definir un componente de negocio en OpenXava es usando una simple clase Java con anotaciones. Tal como muestra el listado 1.1.

Listado 1.1 Invoice: Una clase Java para definir un componente de negocio

```
@Entity // Base de datos
@Table(name="GSTFCT") // Base de datos
@View(members= // Interfaz de usuario
      "year, number, date, paid;" +
      "customer, seller;" +
      "details;" +
      "amounts [ amountsSum, vatPercentage, vat ]"
```

5 Capítulo 1: Arquitectura y filosofía

```
)  
public class Invoice {  
  
    @Id // Base de datos  
    @Column(length=4) // Base de datos  
    @Max(9999) // Validación  
    @Required // Validación  
    @DefaultValueCalculator( // Lógica de negocio declarativa  
        CurrentYearCalculator.class  
    )  
    private int year; // Estructura de datos (1)  
  
    @ManyToOne(fetch=FetchType.LAZY) // Base de datos  
    @DescriptionsList // Interfaz de usuario  
    private Seller seller; // Estructura de datos  
  
    public void applyDiscounts() { // Lógica de negocio programática (2)  
        ...  
    }  
    ...  
}
```

Como puedes ver, todo acerca del concepto de negocio de factura se define en un único lugar, la clase `Invoice`. En esta clase defines cosas de base de datos, estructura de los datos, lógica de negocio, interfaz de usuario, validación, etc.

Esto se hace usando la facilidad de metadatos de Java, las famosas anotaciones. La tabla 1.2 muestra las anotaciones usadas en este ejemplo.

Faceta	Metadatos	Implementado por
Base de datos	@Entity, @Table, @Id, @Column, @ManyToOne	JPA
Interfaz de usuario	@View, @DescriptionsList	OpenXava
Validación	@Max, @Required	Hibernate Validator, OpenXava
Lógica de negocio	@DefaultValueCalculator	OpenXava

Tabla 1.2 Metadatos (anotaciones) usadas en el componente de negocio `Invoice`

Gracias a los metadatos puedes hacer la mayor parte del trabajo de una forma declarativa y el motor de JPA³, Hibernate Validator y OpenXava harán el trabajo sucio por ti.

Además, usamos Java básico, como propiedades (`year` y `seller`, 1) para definir la estructura de los datos, y los métodos (`applyDiscounts()`, 2) para la lógica de negocio programada.

Todo lo que se necesita escribir sobre factura está en `Invoice.java`. Es un componente de negocio. La magia de OpenXava es que puede producir una

³ JPA (Java Persistence API) is the Java standard for persistence (see chapter 2)

aplicación funcional a partir de componentes de negocio.

1.2 Arquitectura de la aplicación

Has visto como los componentes de negocio son las células básicas para construir una aplicación OpenXava, es más, puedes crear una aplicación OpenXava completa usando únicamente componentes de negocio. No obstante, hay otros ingredientes que puedes usar en una aplicación OpenXava.

1.2.1 Perspectiva del desarrollador de aplicaciones

Aunque puedes crear una aplicación completamente funcional usando solo componentes de negocio, a veces es necesario añadir algún que otro elemento adicional para poder ajustar el comportamiento de tu aplicación a tus necesidades. Una aplicación completa de OpenXava tiene la forma de la figura 1.5.

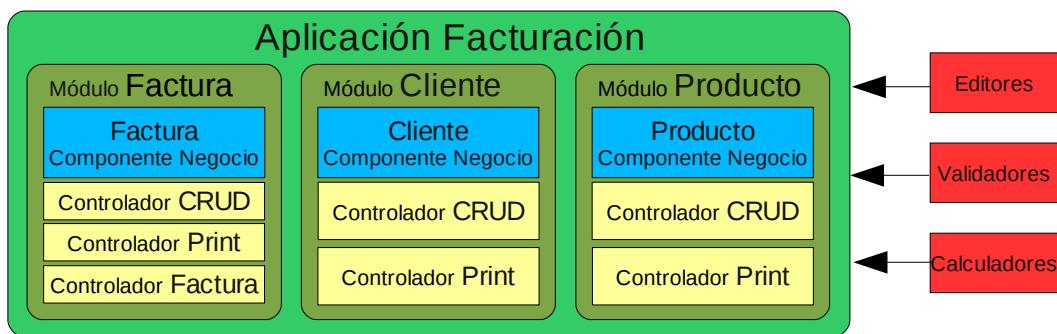


Figure 1.5 Forma de una aplicación OpenXava

A parte de componentes de negocio en la figura 1.5 puedes encontrar módulos, controladores, editores, validadores y calculadores. Veamos que son estas cosas:

- **Componentes de negocio:** Clases de Java que describen los conceptos de negocio en todos sus aspectos. Estas son las únicas piezas requeridas en una aplicación OpenXava.
- **Módulos:** Un módulo es lo que el usuario final ve. Es la unión de un componente de negocio y varios controladores. Puedes omitir la definición de los módulos, en ese caso se asume un módulo por cada componente de negocio.
- **Controladores:** Un controlador es una colección de acciones. Para el usuario, las acciones son botones o vínculos que él puede pulsar; para el desarrollador son clases con lógica a hacer cuando el usuario pulsa en esos botones. Los controladores definen el comportamiento de la aplicación, y normalmente son reutilizables. OpenXava incluye un

7 Capítulo 1: Arquitectura y filosofía

conjunto de controladores predefinidos, y por supuesto, puedes definir los tuyos propios.

- **Editores:** Componentes de la interfaz de usuario para definir la forma en que los miembros de un componente de negocio son visualizados y editados. Es una manera de personalizar la generación de la interfaz de usuario.
- **Validadores:** Lógica de validación reutilizable que puedes usar en cualquier componente de negocio.
- **Calculadores:** Lógica de negocio reutilizable que puedes usar en algunos puntos de los componentes de negocio.

1.2.2 Perspectiva del usuario

El usuario ejecuta los módulos, usualmente tecleando una URL en su navegador, o navegando hasta él dentro de un portal. Un módulo de OpenXava normalmente consta de un modo lista para navegar por los objetos (figura 1.6), y un modo detalle para editarlos (figura 1.7).

The screenshot shows a table with columns: Zone, Number, and Name. The rows contain data such as 'CENTRAL VALENCIA', 'VALENCIA SURETE', 'VALENCIA NORTE', 'CASTELLON DE LA PLANAX', 'ALICANTE CENTROX', 'ALMA 2', 'ALMA 3', 'ALMA 4', 'ALMA 5', and 'ALMA 6'. Each row has edit and delete icons. The top right corner says 'Detail - List'. Below the table are buttons for 'Delete selected', 'Selected to lowercase', and 'Change page row count'. A message at the bottom says 'There are 63 records in list (Hide them)'.

Acciones de los controladores

Cambio de modo: Detalle/Lista

Filtro

Datos como están definidos en el componente de negocio

Paginación

Acciones de los controladores

Figure 1.6 Modo lista de un módulo OpenXava



Figure 1.7 Modo detalle de un módulo OpenXava

Esto muestra visualmente lo que es un módulo: una pieza funcional de software generada a partir de un componente de negocio (datos y lógica de negocio) y varios controladores (comportamiento).

1.2.3 Estructura del proyecto

Has visto el punto de vista conceptual y del usuario de una aplicación, pero

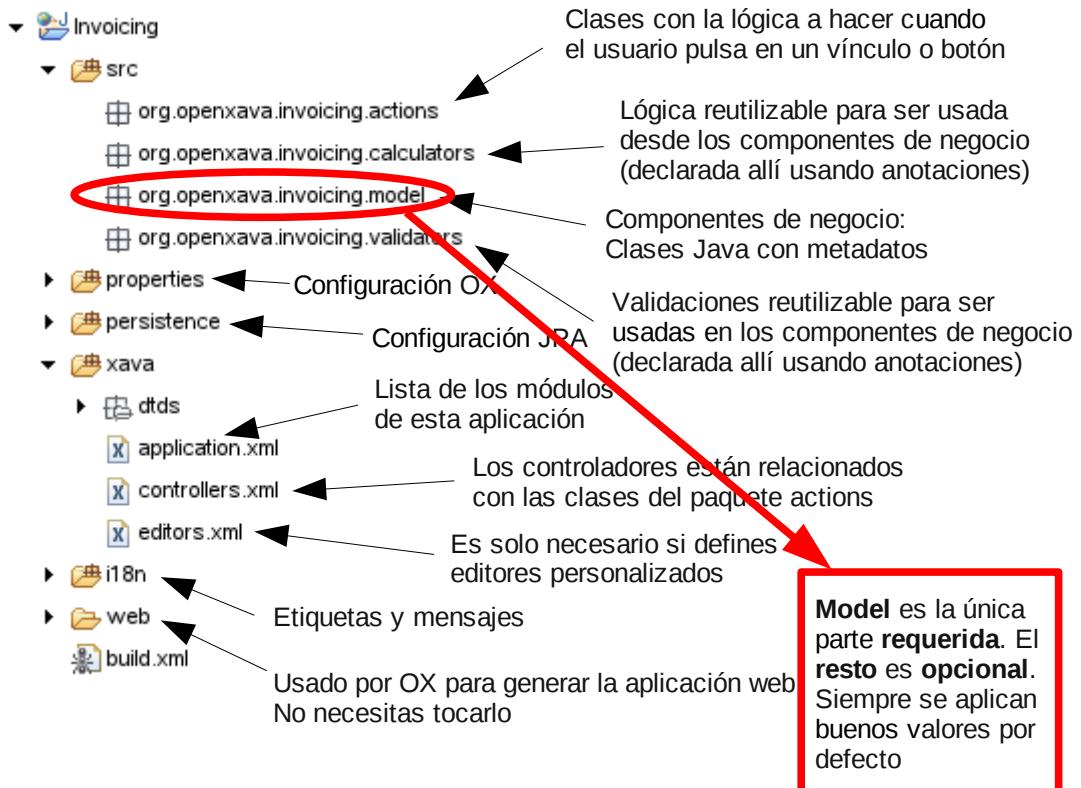


Figura 1.8 Estructura de un proyecto OpenXava típico

9 Capítulo 1: Arquitectura y filosofía

¿qué aspecto tiene una aplicación OpenXava para ti como desarrollador? La figura 1.8 muestra la estructura de un proyecto OpenXava típico.

Solo las clases en el paquete `model`, los componentes de negocio, son obligatorias. Esto es a vista de pájaro. Aprenderás muchos más detalles en el resto del libro.

1.3 Flexibilidad

OpenXava genera automáticamente una aplicación desde clases con metadatos. Esto incluye la generación automática de la interfaz de usuario. Puedes pensar que esto es demasiado “automático”, y es fácil que la interfaz de usuario resultante no cumpla con tus requerimientos, especialmente si estos son muy específicos. Esto no es así, las anotaciones de OpenXava te ofrecen flexibilidad suficiente para interfaces de usuario muy potentes que cubren la mayoría de los casos.

A pesar de eso, OpenXava te proporciona puntos de extensión para darte la oportunidad de personalizar la generación de la interfaz de usuario. Estos puntos de extensión incluyen editores y vistas personalizadas.

1.3.1 Editores

Los editores son los elementos de la interfaz de usuario para ver y editar los miembros de tu componente de negocio. OpenXava usa editores predefinidos para los tipos básicos, pero puedes crear tus propios editores.

La figura 1.9 muestra cómo para números y cadenas se usan editores predefinidos, pero para la propiedad color se usa un editor personalizado

personalizado. Puedes usar JSP, JavaScript, HTML, AJAX, o la tecnología de presentación web que quieras, para crear tu editor personalizado, y entonces asignarlo a los miembros o tipos que deseas.

Esta es una manera bastante reutilizable de personalizar la generación de la interfaz de usuario en tu aplicación OpenXava.



Figura 1.9 Editor personalizado

1.3.2 Vista personalizada

A veces necesitas visualizar tu componente de negocio usando una interfaz de usuario especial, por ejemplo, usando una galería de fotos, un mapa, un calendario, etc. Para esto puedes usar una vista personalizada, que te permite generar la interfaz de usuario usando JavaScript, HTML, JSP, etc. a mano, y entonces usarla dentro de tu aplicación OpenXava. La figura 1.10 muestra un módulo OpenXava que usa una vista personalizada.

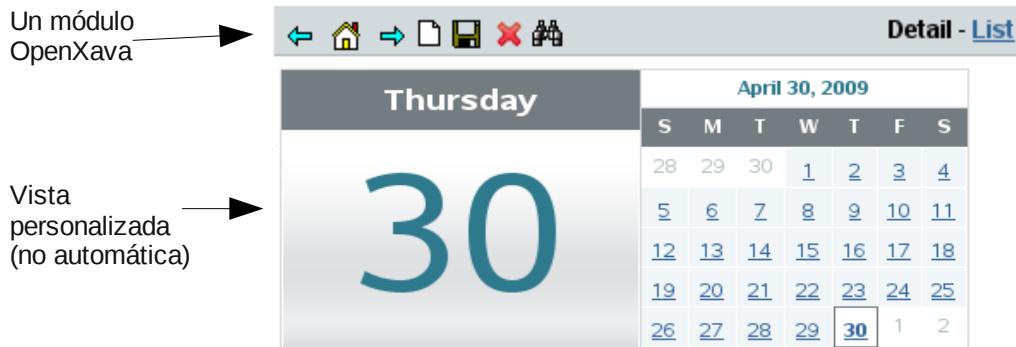


Figura 1.10 Un módulo OpenXava que usa una vista personalizada

Resumiendo, OpenXava genera la interfaz de usuario automáticamente para ti, pero siempre tienes la opción de hacerlo tú mismo.

1.4 Resumen

OpenXava usa un enfoque dirigido por el modelo para hacer desarrollo rápido, donde tú escribes el modelo y obtienes una aplicación completa a partir de él. Lo especial de OpenXava es que el modelo está formado por componentes de negocio.

Un componente de negocio te permite estructurar la aplicación alrededor de los conceptos de negocio. En OpenXava una simple y llana clase Java con anotaciones es suficiente para definir un componente de negocio, haciendo el desarrollo de la aplicación bastante declarativo.

A parte de los componentes de negocio una aplicación OpenXava tiene módulos, controladores, validadores, calculadores, etc. que puedes usar para personalizar tu aplicación. Es posible, incluso, personalizar la forma en que OpenXava genera la interfaz de usuario con los editores y las vistas personalizadas.

OpenXava es una solución pragmática para el desarrollo Java Empresarial. Genera muchísimas cosas automáticamente, pero a la vez es suficientemente

11 Capítulo 1: Arquitectura y filosofía

flexible como para ser útil en el desarrollo de aplicaciones de gestión de la vida real.

Como conclusión, podrás desarrollar aplicaciones con simples clases Java con anotaciones. En los siguientes dos capítulos aprenderás más detalles sobre las anotaciones que podemos usar con OpenXava.

Java

Persistence

API

capítulo 2

Java Persistence API (JPA) es el estándar Java para hacer mapeo objeto-relacional. El mapeo objeto-relacional te permite acceder a los datos en una base de datos relacional usando un estilo orientado a objetos. En tu aplicación solo trabajas con objetos, estos objetos se declaran como persistentes, y es responsabilidad del motor JPA leer y grabar los objetos desde la base de datos a la aplicación.

JPA mitiga el famoso problema de desajuste de impedancia, que se produce porque las bases de datos relacionales tienen una estructura, tablas y columnas con datos simples, y las aplicaciones orientadas a objetos otra, clases con referencias, colecciones, interfaces, herencia, etc. Es decir, si en tu aplicación estás usando clases Java para representar conceptos de negocio, tendrás que escribir bastante código SQL para escribir los datos desde tus objetos a la base de datos y viceversa. JPA lo hace para ti.

Este capítulo es una introducción a JPA. Para una completa inmersión en esta tecnología estándar necesitarías un libro completo sobre JPA, de hecho, cito algunos en el sumario de este capítulo.

Por otra parte, si ya conoces JPA puedes saltarte este capítulo.

2.1 Anotaciones JPA

JPA tiene 2 aspectos diferenciados, el primero es un conjunto de anotaciones Java para añadir a tus clases marcándolas como persistentes y dando detalles acerca del mapeo entre las clases y las tablas. Y el segundo es un API para leer y escribir objetos desde tu aplicación. Veamos primero las anotaciones.

2.1.1 Entidad

En la nomenclatura JPA a una clase persistente se le llama entidad. Podemos decir que una entidad es una clase cuyas instancias se graban en la base de datos. Usualmente cada entidad representa un concepto del dominio, por lo tanto usamos una entidad JPA como base para definir un componente de negocio en OpenXava, de hecho puedes crear un aplicación completa de OX a partir de simples entidades JPA. El listado 2.1 muestra como se define una entidad JPA.

Listado 2.1 Anotaciones para la definición de una entidad JPA

```
@Entity // Para definir esta clase como persistente
@Table(name="GSTCST") // Para indicar la tabla de la base de datos (opcional)
public class Customer {
```

Como puedes ver solo has de marcar tu clase con la anotación `@Entity`, y opcionalmente también con la anotación `@Table`, en este caso estamos diciendo

15 Capítulo 2: Java Persistence API

que la entidad `Customer` se graba en la tabla `GSTCST` de la base de datos. De ahora en adelante, JPA guardará y recuperará información entre los objetos `Customer` en la aplicación y la tabla `GSTCST` en la base de datos, como muestra la figura 2.1.

Además, marcar `Customer` con `@Entity` es suficiente para que OpenXava la reconozca como un componente de negocio. Sí, en OpenXava “entidad” es sinónimo de componente de negocio.

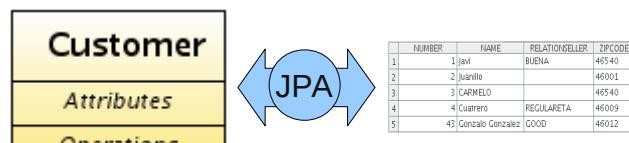


Figura 2.1 JPA mapea clases con tablas

2.1.2 Propiedades

El estado básico de una entidad se representa mediante propiedades. Las propiedades de una entidad son propiedades Java convencionales, con *getters* y *setters*⁴, como ves en el listado 2.2.

Listado 2.2 Definición de propiedad en una entidad

```
private String name;  
  
public String getName() {  
    return name;  
}  
  
public void setName(String name) {  
    this.name = name;  
}
```

Por defecto las propiedades son persistentes, es decir, JPA asume que la propiedad `name` (listado 2.2) se almacena en la columna llamada 'name' de la tabla en la base de datos. Si quieres que una determinada propiedad no se guarde en la base de datos has de marcarla como `@Transient` (listado 2.3). Nota que hemos anotado el campo, puedes anotar el *getter* en su lugar, si así lo prefieres (listado 2.4).

Esta norma aplica a todas la anotaciones JPA, puedes anotar el campo (acceso basado en el campo) o el *getter* (acceso basado en la propiedad), pero no mezcles los dos estilos en la misma entidad.

Listado 2.3 Propiedad transitoria

```
@Transient // Marcada como transitoria, no se almacena en la base de datos  
private String name;  
  
public String getName() {
```

4 To learn more options about properties and fields look at section 2.1.1 of JPA 1.0 Specification

```

        return name;
    }

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

```

Listado 2.4 Propiedad transitoria usando acceso basado en propiedad

```

private String name;

@Transient // Marcamos el getter, por tanto todas las anotaciones JPA
public String getName() { // en este entidad tienen que estar en los getters
    return name;
}

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

```

Otras anotaciones útiles para las propiedades son @Column para especificar el nombre y longitud de la columna de la tabla, e @Id para indicar que propiedad es la propiedad clave. Puedes ver el uso de estas anotaciones en la ya utilizable entidad Customer en el listado 2.5.

Listado 2.5 La entidad Customer con anotaciones @Id y @Column

```

@Entity
@Table(name="GSTCST")
public class Customer {

    @Id // Indica que number es la propiedad clave (1)
    @Column(length=5) // Aquí @Column indica solo la longitud (2)
    private int number;

    @Column(name="CSTNAM", length=40) // La propiedad name se mapea a la columna
    private String name; // CSTNAM en la base de datos

    public int getNumber() {
        return number;
    }

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

    public String getName() {
        return name;
    }

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

```

Es obligatorio que al menos una propiedad sea clave (1). Has de marcar la propiedad clave con @Id, y normalmente se mapea contra la columna clave de la tabla. @Column puede usarse para indicar la longitud sin el nombre de columna (2). La longitud es usada por el motor JPA para la generación de esquema, pero también es usada por OpenXava para conocer el tamaño del editor en la interfaz de usuario. A partir de la entidad Customer del listado 2.5 OpenXava genera la interfaz de usuario que puedes ver en la figura 2.2.

Ahora que ya sabes como definir propiedades básicas en tu entidad, aprendamos como declarar relaciones entre entidades usando referencias y colecciones.

2.1.3 Referencias

Una entidad puede hacer referencia a otra entidad. Únicamente has de definir una referencia Java convencional anotada con la anotación JPA @ManyToOne, tal cual se ve en el listado 2.6.

Listado 2.6 Una referencia otra entidad usando @ManyToOne

```
@Entity
public class Invoice {

    @ManyToOne( // La referencia se almacena como una relación a nivel de base de datos (1)
        fetch=FetchType.LAZY, // La referencia se cargará bajo demanda (2)
        optional=false) // La referencia tiene que tener valor siempre
    @JoinColumn(name="INVCST") // INVCST es la columna para la clave foranea (3)
    private Customer customer; // Una referencia Java convencional (4)

    // Getter y setter para customer
```

Como puedes ver hemos declarado una referencia a Customer dentro de Invoice en un estilo Java simple y llano (4). @ManyToOne (1) es para indicar que esta referencia se almacenará en la base de datos como una relación muchos-a-un entre la tabla para Invoice y la tabla para Customer, usando la columna INVCST (3) como clave foránea. @JoinColumn (3) es opcional. JPA asume valores por defecto para las columnas de unión (CUSTOMER_NUMBER en este caso).

Si usas fetch=FetchType.LAZY (3) los datos del objeto Customer no se cargarán hasta que se usen por primera vez. Es decir, en el momento justo que uses la referencia a Customer, por ejemplo, llamando al método

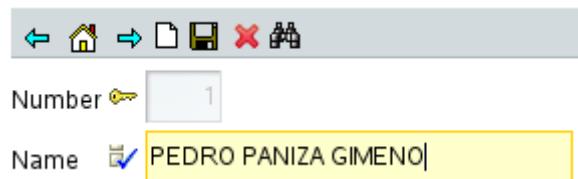


Figura 2.2 Interfaz de usuario para la entidad Customer con 2 propiedades

`invoice.getCustomer().getName()` los datos del Customer son cargados desde la base de datos. Es aconsejable usar siempre *lazy fetching*.

Una referencia Java convencional normalmente corresponde a una relación `@ManyToOne` en JPA y a una asociación `*..1` en UML (figura 2.3).



Figura 2.3 Una referencia a otra entidad: Una asociación `*..1` en UML

La figura 2.4 muestra la interfaz de usuario que OpenXava genera automáticamente para una referencia.

Has visto como hacer referencia a otras entidades, pero también puedes hacer referencia a otros objetos que no son entidades, por ejemplo, a objetos incrustados.

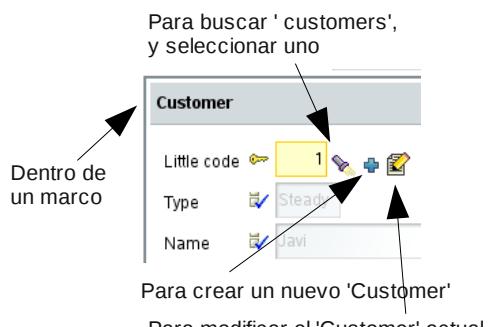


Figura 2.4 Interfaz de usuario para una referencia

2.1.4 Clases incrustables

Además de entidades puedes usar clases incrustables para modelar algunos conceptos de tu dominio. Si tienes una entidad A que tiene una referencia a B, modelarías B como una clase incrustable cuando:

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

A veces el mismo concepto puede ser modelado como incrustable o como entidad. Por ejemplo, el concepto de dirección. Si una dirección es compartida por varias personas entonces tienes que usar una referencia a una entidad, mientras que si cada persona tiene su propia dirección quizás un objeto incrustable sea mejor opción.

Modelemos una dirección (*address*) como una clase incrustable. Es fácil, simplemente crea una simple clase Java y anótala como `@Embeddable`, tal como muestra el listado 2.7.

19 Capítulo 2: Java Persistence API

Listado 2.7 Definición de clase incrustable

```
@Embeddable // Para definir esta clase como incrustable
public class Address {

    @Column(length=30) // Puedes usar @Column como en una entidad
    private String street;

    @Column(length=5)
    private int zipCode;

    @Column(length=20)
    private String city;

    // Getters y setters
    ...
}
```

Y ahora, crear una referencia a Address desde una entidad también es fácil. Se trata simplemente de una referencia Java normal anotada como @Embedded (listado 2.8).

Listado 2.8 Una referencia a una clase incrustable desde una entidad

```
@Entity
@Table(name="GSTCST")
public class Customer {

    @Embedded // Referencia a una clase incrustable
    private Address address; // Una referencia Java convencional

    // Getter y setter para address
    ...
}
```

Desde el punto de vista de la persistencia un objeto incrustable se almacena en la misma tabla que su entidad contenedora. En este caso las columnas street, zipCode y city están en la tabla para Customer. Address no tiene tabla propia.

La figura 2.5 muestra la interfaz de usuario que OpenXava genera automáticamente para una referencia a una clase incrustable.

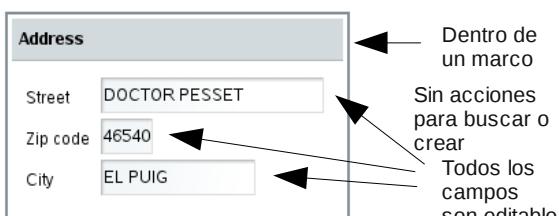


Figura 2.5 Interfaz de usuario de una referencia incrustada

2.1.5 Colecciones

Una entidad puede tener una colección de entidades. Solo has de definir una colección de Java convencional anotada con las anotaciones JPA @OneToMany o

@ManyToMany, tal como muestra el listado 2.9.

Listado 2.9 Una colección de entidades usando @OneToMany

```
@Entity
public class Customer {

    @OneToMany(   // La colección es persistente (1)
        mappedBy="customer") // La referencia customer de Invoice se usa
                           // para mapear la relación a nivel de base de datos (2)
    private Collection<Invoice> invoices; // Una colección Java convencional (3)

    // Getter y setter para invoices
    ...
}
```

Como puedes ver declaramos una colección de entidades Invoice dentro de Customer en un estilo Java plano (3). @OneToMany (1) es para indicar que esta colección se almacena en la base de datos con una relación uno-a-muchos entre la tabla para Customer y la tabla para Invoice, usando la columna de customer en Invoice (normalmente una clave foránea hacia la tabla Customer desde la tabla Invoice).

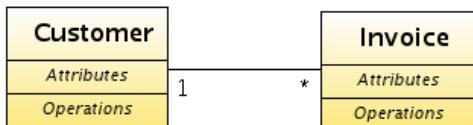


Figure 2.6 Una colección de entidades: Una asociación UML 1..*

Una colección Java convencional normalmente corresponde a una relación @OneToMany o @ManyToMany en JPA y a una asociación con una multiplicidad 1..* o *..* en UML (figura 2.6).

No es posible definir una colección de objetos incrustables en JPA (al menos en v1.0), pero es fácil simular la semántica de los objetos incrustados usando una colección de entidades con el atributo cascade de @OneToMany, justo como se muestra en el listado 2.10.

Listado 2.10 Colección de entidades con cascade REMOVE para incrustamiento

```
@Entity
public class Invoice {

    @OneToMany (mappedBy="invoice",
        cascade=CascadeType.REMOVE) // Cascade REMOVE para simular incrustamiento
    private Collection<InvoiceDetail> details;

    // Getter y setter para details
    ...
}
```

Así, cuando una factura (Invoice) se borra sus líneas (details) se borran también. Podemos decir que una factura (invoice) *tiene* líneas (details).

Puedes ver la interfaz de usuario que OpenXava genera automáticamente para

21 Capítulo 2: Java Persistence API

una colección de entidades en la figura 2.7.

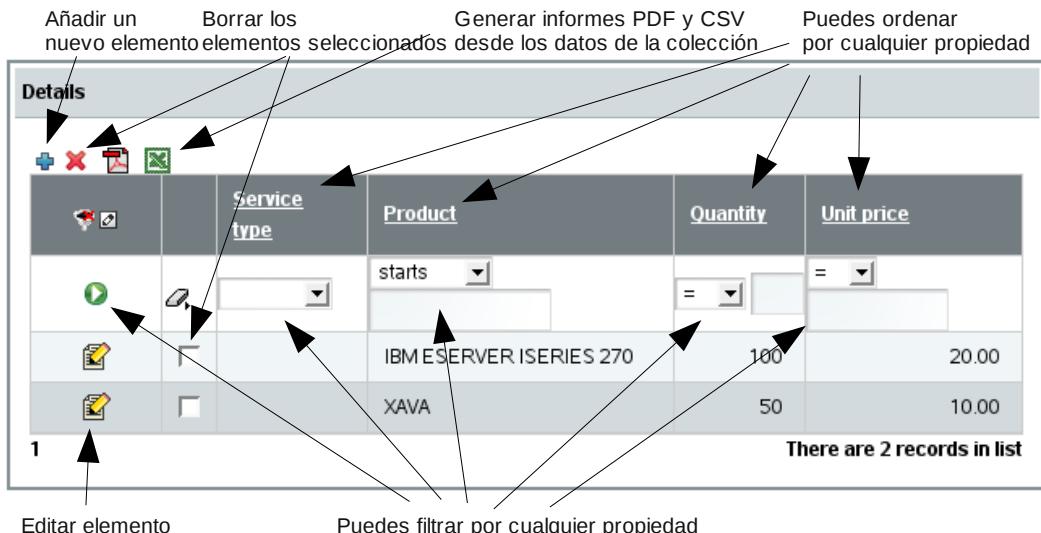


Figura 2.7 Interfaz de usuario para una colección de entidades

Hay algunas ligeras diferencias en el comportamiento de la interfaz de usuario si usas *cascade REMOVE* o *ALL*. Con *cascade REMOVE* o *ALL* cuando el usuario pulsa para añadir un nuevo elemento, puede introducir todos los datos del elemento, por otra parte si tu colección no es *cascade REMOVE* o *ALL*, cuando el usuario pulsa para añadir un nuevo elemento se muestra una lista de entidades para escoger.

Has visto como escribir tus entidades usando anotaciones JPA, y como OpenXava las interpreta para así generar una interfaz de usuario adecuada. Ahora vas a aprender como usar el API JPA para leer y escribir de la base de datos de desde tu propio código.

2.2 API JPA

La clase de JPA más importante es `javax.persistence.EntityManager`. Un `EntityManager` te permite grabar, modificar y buscar entidades.

El listado 2.11 muestra la forma típica de usar JPA en una aplicación no OpenXava.

Listado 2.11 La forma típica de usar el API de JPA

```
EntityManagerFactory f = // Necesitas un EntityManagerFactory para crear un manager  
Persistence.createEntityManagerFactory("default");
```

```

EntityManager manager = f.createEntityManager(); // Creas el manager
manager.getTransaction().begin(); // Has de empezar una transacción
Customer customer = new Customer(); // Ahora creas tu entidad
customer.setNumber(1); // y la rellenas
customer.setName("JAVI");
manager.persist(customer); // persist marca el objeto como persistente
manager.getTransaction().commit(); // Al confirmar la transacción los cambios se
// efectúan en la base de datos
manager.close(); // Has de cerrar el manager

```

Ves como es una forma muy verbosa de trabajar. Demasiado código burocrático. Si así lo prefieres puedes usar código como éste dentro de tus aplicaciones OpenXava, aunque OpenXava te ofrece una forma más sucinta de hacerlo, míralo en el listado 2.12.

Listado 2.12 Código JPA para guardar una entidad en una aplicación OpenXava

```

Customer customer = new Customer();
customer.setNumber(1);
customer.setName("PEDRO");
XPersistence.getManager().persist(customer); // Esto es suficiente (1)

```

Dentro de un aplicación OpenXava puedes obtener el *manager* mediante la clase `org.openxava.jpa.XPersistent`. No necesitas cerrar el *manager*, ni arrancar y parar la transacción. Este trabajo sucio lo hace OpenXava por ti. El código que ves en el listado 2.12 es suficiente para grabar una nueva entidad en la base de datos (1).

Si quieras modificar una entidad existente has de hacerlo como en el listado 2.13.

Listado 2.13 Modificando un objeto usando JPA

```

Customer customer = XPersistence.getManager()
    .find(Customer.class, 1); // Primero, busca el objeto a modificar (1)
customer.setName("PEDRITO"); // Entonces, cambias el estado del objeto. Nada más

```

Para modificar un objeto solo has de buscarlo y modificarlo. JPA es responsable de grabar los cambios en la base de datos al confirmar la transacción (a veces antes), y OpenXava confirma las transacciones JPA automáticamente.

En el listado 2.13 has visto como encontrar por clave primaria, usando `find()`. Pero, JPA te permite usar consultas, míralo en el listado 2.14.

Listado 2.14 Usando consultas para buscar entidades entities

```

Customer pedro = (Customer) XPersistence.getManager()
    .createQuery(
        "from Customer c where c.name = 'PEDRO'") // Consulta JPQL (1)
    .getSingleResult(); // Para obtener una única entidad (2)

List pedros = XPersistence.getManager()
    .createQuery(

```

23 Capítulo 2: Java Persistence API

```
"from Customer c where c.name like 'PEDRO%')") // Consulta JPQL  
.getResultSet(); // Para obtener una colección de entidades (3)
```

Puedes usar el lenguaje de consultas de JPA (Java Persistence Query Language, JPQL, 1) para crear consultas complejas sobre tu base de datos, y obtener una entidad única, usando el método `getSingleResult()` (2), o una colección de entidades mediante el `getResultSet()` (3).

2.3 Resumen

Este capítulo ha sido una breve introducción a la tecnología JPA. Por desgracia, muchas e interesantes cosas sobre JPA se nos han quedado en el tintero, como la herencia, el polimorfismo, las claves compuestas, relaciones uno a uno y muchos a muchos, relaciones unidireccionales, métodos de retrollamada, consultas avanzadas, etc. De hecho, hay más de 60 anotaciones en JPA 1.0. Necesitaríamos varios libros completos para aprender todos los detalles sobre JPA.

Afortunadamente, tendrás la oportunidad de aprender algunos casos de uso avanzados de JPA en el transcurso de este libro. Y si aun quieres aprender más, lee otros libros y referencias, como por ejemplo:

- *Java Persistence API Specification* por Linda DeMichiel, Michael Keith y otros (disponible en jcp.org como parte de JSR-220).
- *Pro EJB 3: Java Persistence API* por Mike Keith and Merrick Schincariol.
- *Java Persistence with Hibernate* por Christian Bauer y Gavin King.

JPA es una tecnología indisputable en el universo Java de Empresa, por tanto todo el conocimiento y código que acumulemos alrededor de JPA es siempre una buena inversión.

Aparte de las anotaciones estándar de JPA que has visto en este capítulo, hay otras anotaciones útiles que puedes usar en tus entidades. Veámoslas en el siguiente capítulo.

Anotaciones

capítulo 3

Las anotaciones son la herramienta que Java provee para definir metadatos en tus aplicaciones, en otras palabras, es la forma de hacer desarrollo declarativo con Java, donde dices el “qué” y no el “cómo”.

En el capítulo 2 has visto las anotaciones JPA para hacer mapeo objeto-relacional. En este capítulo verás las anotaciones que puedes usar en una aplicación OpenXava para definir validaciones, la interfaz de usuario y otros aspectos para ajustar la aplicación a tus necesidades.

El objetivo de este capítulo es introducirte a estas anotaciones, pero no te muestra todas su sutilezas y posibles casos de uso; lo cual requeriría varios libros de los gordos.

3.1 Validación

OpenXava incluye un marco de validación fácil de usar y extensible. Además soporta Hibernate Validator.

3.1.1 Validación declarativa

La forma preferida de hacer validación en OpenXava es mediante anotaciones, es decir, de manera declarativa. Por ejemplo, solo has de marcar una propiedad como @Required (listado 3.1).

Listado 3.1 Una validación declarativa: marcando una propiedad como requerida

```
@Required // Esto fuerza a validar esta propiedad como requerida al grabar
private String name;
```

Y OpenXava hará la validación correspondiente al grabar (figura 3.1).

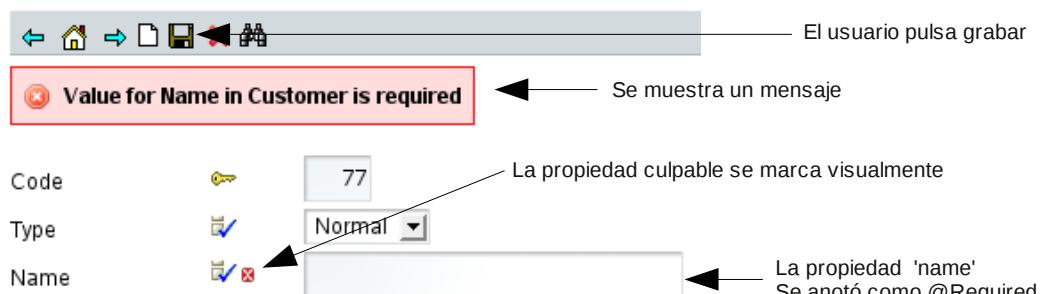


Figura 3.1 Efecto visual de una validación

3.1.2 Validaciones predefinidas

Las anotaciones de validación que OpenXava tiene incluidas (tabla 3.1) han sido definidas con Hibernate Validator (excepto @RemoveValidator). Hibernate

27 Capítulo 3: Anotaciones

Validator es un marco de validación muy popular (no es el motor de persistencia Hibernate).

Anotación	Aplica a	Validación
@Required	Propiedad	Comprueba si la propiedad tiene valor
@PropertyValidator	Propiedad	Permite definir una lógica de validación personalizada
@EntityValidator	Entidad	Permite definir una lógica de validación personalizada
@RemoveValidator	Entidad	Permite definir una lógica de validación personalizada al borrar

Tabla 3.1 Validaciones predefinidas de OpenXava

Las anotaciones de Hibernate Validator son reconocidas por OpenXava, por tanto puedes usar todos las anotaciones predefinidas de Hibernate Validator en tus aplicaciones OpenXava (tabla 3.2).

Anotación	Aplica a	Validación
@Length(min=, max=)	Propiedad (cadena)	Comprueba que la longitud de la cadena esté dentro del rango
@Max(value=)	Propiedad (numérica o cadena representando un valor numérico)	Comprueba que el valor sea igual o menor al máximo
@Min(value=)	Propiedad (numérica o cadena representando un valor numérico)	Comprueba que el valor sea igual o mayor al mínimo
@NotNull	Propiedad	Comprueba que el valor no sea nulo
@NotEmpty	Propiedad	Comprueba que la cadena no sea nula ni vacía
@Past	Propiedad (fecha o calendario)	Comprueba que la fecha esté en el pasado
@Future	Propiedad (fecha o calendario)	Comprueba que la fecha esté en el futuro
@Pattern(regex="regexp", flag=)*	Propiedad (cadena)	Comprueba que la propiedad cumpla la expresión regular dado un <i>match flag</i>

Tabla 3.2 Validaciones de Hibernate predefinidas, utilizables desde OpenXava

Anotación	Aplica a	Validación
@Range(min=, max=)	Propiedad (numérica o cadena representando un valor numérico)	Comprueba que el valor este entre min y max (ambos incluidos)
@Size(min=, max=)	Propiedad (array, colección, mapa)	Comprueba que la cantidad de elementos esté entre min y max (ambos incluidos)
@AssertFalse	Propiedad	Comprueba que el método se evalúe a falso (útil para restricciones expresadas con código en vez de por anotaciones)
@AssertTrue	Propiedad	Comprueba que el método se evalúe a cierto (útil para restricciones expresadas con código en vez de por anotaciones)
@Valid	Propiedad (objeto)	Realiza la validación recursivamente en el objeto asociado. Si el objeto es una colección o un array, los elementos son validados recursivamente. Si el objeto es un mapa, los elementos valor son validados recursivamente
@Email	Propiedad (cadena)	Comprueba que la cadena cumpla con la especificación de formato para una dirección de correo electrónico
@CreditCardNumber	Propiedad (cadena)	Comprueba si la cadena es un número de tarjeta de crédito bien formateado (derivado del algoritmo del Luhn)
@Digits	Propiedad (numérica o cadena representando un valor numérico)	Comprueba que la propiedad sea un número con como máximo los enteros indicados en integerDigits y los decimales indicados en fractionalDigits
@EAN	Propiedad (cadena)	Comprueba que la cadena sea un código EAN o UPC-A correctamente formateado

Tabla 3.2(cont.) Validaciones de Hibernate predefinidas

3.1.3 Validación propia

Añadir tu propia lógica de validación a tu entidad es muy fácil porque las anotaciones `@PropertyValidator`, `@EntityValidator` y `@RemoveValidator` te permiten indicar una clase (el validador) con la lógica de validación.

Por ejemplo, si quieras tu propia lógica para validar una propiedad `unitPrice`, has de escribir algo parecido al código del listado 3.2.

29 Capítulo 3: Anotaciones

Listado 3.2 Definir una validación propia con @PropertyValidator

```
@PropertyValidator(UnitPriceValidator.class) // Contiene la lógica de validación  
private BigDecimal unitPrice;
```

Y ahora puedes escribir la lógica que quieras dentro de la clase UnitPriceValidator, como ves en el listado 3.3.

Listado 3.3 Lógica de validación propia en una clase validador

```
public class UnitPriceValidator  
    implements IPropertyValidator { // Tiene que implementar IPropertyValidator (1)  
  
    public void validate( // Requerido por IPropertyValidator (2)  
        Messages errors, // Aquí añades los mensajes de error(3)  
        Object object, // El valor a validar  
        String objectName, // El nombre de entidad, normalmente para usar en el mensaje  
        String propertyName)// El nombre de propiedad, normalmente para usar en el mensaje  
    {  
        if (object == null) return;  
        if (!(object instanceof BigDecimal)) {  
            errors.add( // Si añades un error la validación fallará  
                "expected_type", // Id de mensaje en el archivo i18n  
                propertyName, // Argumentos para el mensaje i18n  
                objectName,  
                "bigdecimal");  
            return;  
        }  
        BigDecimal n = (BigDecimal) object;  
        if (n.intValue() > 1000) {  
            errors.add("not_greater_1000"); // Id de mensaje en el archivo i18n  
        }  
    }  
}
```

Como ves tu clase validador ha de implementar IPropertyValidator (1), esto te obliga a tener un método validate() (2) que recibe un objeto Messages, que llamamos errors (3); que es un contenedor de los mensajes de error. Solo necesitas añadir algún mensaje de error para hacer que falle la validación.

Esta es una forma sencilla de hacer tus propias validaciones, además la lógica de validación en tus validadores puede reutilizarse por toda tu aplicación. Aunque, si lo que quieras es crear validaciones reutilizables una opción mejor es crear tu propia anotación de validación usando Hibernate Validator; es más largo que usar una clase validador, pero es más elegante si reutilizas la validación muchas veces. Crearemos un Hibernate Validator propio en el capítulo 9.

3.1.4 Aprende más sobre validaciones

Esta sección es solo una breve introducción a la validación en OpenXava. Puedes aprender más detalles sobre este tema en el capítulo 3 (Modelo) de la *Guía de Referencia* de OpenXava y en la *Reference Documentation* de Hibernate

Validator.

Adicionalmente, aprenderás casos de validación más avanzados en el capítulo 9 de este mismo libro.

3.2 Interfaz de usuario

Aunque OpenXava genera automáticamente una interfaz de usuario bastante funcional a partir de una entidad JPA desnuda, esto es solo útil para casos muy básicos. En aplicaciones de la vida real es necesario refinrar la manera en que la interfaz de usuario es generada. En OpenXava esto se hace con anotaciones que, con un nivel de abstracción alto, definen el aspecto de la interfaz de usuario.

3.2.1 La interfaz de usuario por defecto

Por defecto, OpenXava genera una interfaz de usuario que muestra todos los miembros de la entidad en secuencia. Con una entidad como la del listado 3.4

Listado 3.4 Una entidad desnuda, sin anotaciones de vista

```
@Entity
public class Seller {

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

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

    @OneToMany(mappedBy="seller")
    private Collection<Customer> customers;

    // Getters y setters
    ...
}
```

OpenXava producirá para ti la interfaz de usuario de la figura 3.2.

The figure shows a screenshot of the OpenXava interface for the Seller entity. The interface is a web-based form with a header 'Customers' and a table below it. The table has columns: Number, Name, and Remarks. There are two records in the list, with values 1 and 2 respectively. Arrows point from various annotations in the code to specific parts of the UI:

- An arrow points from the `@Id` annotation to the 'Number' column header.
- An arrow points from the `@Column(length=3)` annotation to the 'Number' column editor.
- An arrow points from the `@Required` annotation to the 'Name' column header.
- An arrow points from the `@OneToMany` annotation to the 'customers' collection.
- An annotation 'La longitud del editor de 'name' es 40 por @Column(length=40)' points to the 'Name' column editor.
- An annotation 'Por @Id' points to the 'Number' column header.
- An annotation 'Por @Required' points to the 'Name' column header.

Figura 3.2 Interfaz de usuario por defecto para una entidad

31 Capítulo 3: Anotaciones

Como ves, muestra los miembros (`number`, `name` y `customers` en este caso) en el mismo orden en que fueron declarados en el código Java. OpenXava usa las anotaciones JPA y de validación para generar la mejor interfaz de usuario posible, por ejemplo, determina el tamaño del editor a partir de `@Column(length)`, muestra la llavecita de clave para la propiedad con `@Id` y muestra un icono para indicar que es requerido si la propiedad está marcada con `@Required`, y así por el estilo.

Esta interfaz por defecto es útil para casos simples, pero para interfaces de usuario más avanzadas necesitas una forma de personalizar. OpenXava te proporciona anotaciones para hacerlo, tal como `@View` para definir la disposición de los miembros.

3.2.2 La anotación `@View`

`@View` sirve para definir la disposición de los miembros en la interfaz de usuario. Se define a nivel de entidad. Mira el ejemplo en el listado 3.5.

Listado 3.5 Anotación `@View` para definir la disposición de la interfaz de usuario

```
@Entity  
@View(members=  
    "year, number, date, paid;" + // Coma indica en la misma línea  
    "discounts [" + // Entre corchetes indica dentro de un marco  
    "  customerDiscount, yearDiscount;" +  
    "];" +  
    "comment;" + // Punto y coma indica nueva línea  
    "customer { customer }" + // Entre llaves indica dentro de una pestaña  
    "details { details }" +  
    "amounts { amountsSum; vatPercentage; vat }" +  
    "deliveries { deliveries }"  
)  
public class Invoice {
```

La interfaz de usuario resultante está en la figura 3.3.



Figura 3.3 Anotación `@View` para definir la disposición de la interfaz de usuario

Como ves, definir la disposición de los miembros es fácil, solo necesitas enumerarlos dentro de una cadena, usando comas para separar los elementos, punto y coma para salto de línea, corchetes para grupos (marcos), llavecitas para secciones (pestañas) y así por el estilo.

Puedes tener varias vistas por cada entidad, para eso usa la anotación @Views, dando un nombre a cada vista (listado 3.6).

Si dejas una vista sin nombre, será la vista por defecto. Los nombres de vista se usarán desde otras partes de la aplicación para escoger que vista usar.

Listado 3.6 Varias vista para una entidad

```
@Entity
@Views({
    @View(
        members="number, name; address; invoices"
    ),
    @View(
        name="Simple", members="number, name"
    )
})
public class Customer {
```

Con @View defines la distribución, pero también necesitas definir la forma en que cada miembro es visualizado, para eso, OpenXava te proporciona un conjunto de anotaciones bastante útiles que verás en la siguiente sección.

3.2.3 Refinando la presentación de los miembros

OpenXava te permite refinar la interfaz de usuario para cualquier propiedad, referencia o colección de infinidad de maneras. Sólo necesitas añadir la anotación correspondiente. Por ejemplo, por defecto una referencia (una relación @ManyToOne) se visualiza usando un marco con una vista detallada, si quieres mostrar esa referencia usando

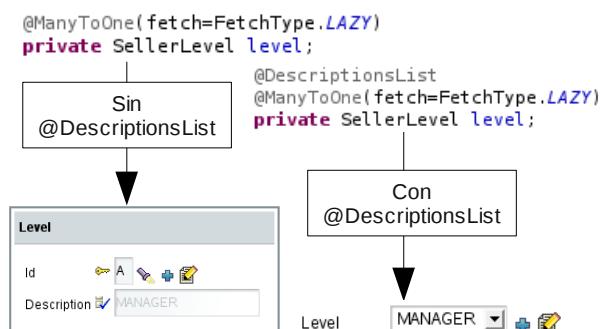


Figura 3.4 Efecto de @DescriptionsList en una relación @ManyToOne

un combo solo has de anotar la referencia con @DescriptionsList (figura 3.4).

Puede que quieras el efecto de una anotación solo para algunas vistas. Para ello usa el atributo forViews disponible en todas las anotaciones de interfaz de usuario (listado 3.7).

33 Capítulo 3: Anotaciones

Listado 3.7 forViews para limitar el efecto de una anotación a algunas vistas

```
@Views ({ // You have several views for Seller
    @View(members=" ... "),
    @View(name="Simplest", members=" ... "),
    @View(name="Simple", members=" ... "),
    @View(name="Complete", members=" ... "),
})
public class Seller {

    @DescriptionsList(forViews="Simplest, Simple") // El combo solo se usará
    @ManyToOne(fetch=FetchType.LAZY)           // para 'level' en las vistas Simplest y Simple
    private SellerLevel level;
}
```

La tabla 3.3 muestra todas las anotaciones OpenXava para personalizar la interfaz de usuario de los miembros de una entidad.

Anotación	Descripción	Aplica a
@Action	Asocia una acción a una propiedad o referencia en la vista	Propiedades y referencias
@AsEmbedded	Hace que comportamiento en la vista de una referencia (o colección) a entidad sea como en el caso de un objeto incrustado (o colección de entidades con CascadeType.REMOVE)	Referencia y colecciones
@CollectionView	La vista del objeto referenciado (cada elemento de la colección) que será usada para visualizar el detalle	Colecciones
@Condition	Restringe los elementos que aparecen en la colección	Colecciones
@DescriptionsList	Para visualizar una referencia como una lista de descripciones (un combo)	Referencias
@DetailAction	Añade una acción al detalle que está siendo editado en una colección	Colecciones
@DisplaySize	El tamaño en caracteres del editor en la interfaz de usuario usado para visualizar esta propiedad	Propiedades
@EditAction	Permite definir una acción propia para editar el elemento de la colección	Colecciones
@EditOnly	El usuario final podrá modificar los elementos existentes en la colección, pero no añadir o quitar elementos	Colecciones
@Editor	Nombre del editor a usar para visualizar el miembro en esta vista	Propiedades, referencias y colecciones.

Tabla 3.3 Anotaciones para personalizar la visualización de los miembros

Anotación	Descripción	Aplica a
@HideDetailAction	En una colección permite definir una acción propia para ocultar la vista de detalle	Colecciones
@LabelFormat	Formato para visualizar la etiqueta de esta propiedad o referencia (visualizada como lista descripciones)	Propiedades y referencias
@ListAction	Para añadir acciones a la lista en una colección	Colecciones
@ListProperties	Propiedades a mostrar en la lista que visualiza la colección	Colecciones
@NewAction	Permite definir una acción propia para añadir un nuevo elemento a la colección	Colecciones
@NoCreate	El usuario final no podrá crear nuevos objetos del tipo referenciado desde aquí	Referencias y colecciones
@NoFrame	La referencia no se visualizará dentro de un marco	Referencias
@NoModify	El usuario final no podrá modificar el objeto actual referenciado desde aquí	Referencias y colecciones
@NoSearch	El usuario no tendrá el vínculo para hacer búsquedas con una lista, filtros, etc.	Referencias
@OnChange	Acción a ejecutar cuando el valor de la propiedad o referencia cambia	Propiedades y referencias
@OnChangeSearch	Acción a ejecutar para hacer la búsqueda de la referencia cuando el usuario teclea los valores clave	Referencias
@OnSelectElementAction	Permite definir una acción a ejecutar cuando un elemento de la colección es seleccionado o deseleccionado	Colecciones
@ReadOnly	El miembro nunca será editable por el usuario final en las vistas indicadas	Propiedades, referencias y colecciones
@ReferenceView	Vista del objeto referenciado a usar para visualizar esta referencia	Referencia
@RemoveAction	Permite definir una acción propia para quitar el elemento de la colección	Colecciones

Tabla 3.3(cont.) Anotaciones para personalizar la visualización de los miembros

35 Capítulo 3: Anotaciones

Anotación	Descripción	Aplica a
@RemoveSelectedAction	Permite definir una acción propia para quitar los elementos seleccionados de la colección	Colecciones
@RowStyle	Para indicar el estilo de la fila para la lista y colecciones	Entidad (mediante @Tab) y colecciones
@SaveAction	Permite definir una acción propia para grabar el elemento de la colección	Colecciones
@SearchAction	Permite definir una acción propia para buscar	Referencias
@ViewAction	Permite definir una acción propia para visualizar el elemento de la colección	Colecciones
@XOrderBy	La versión eXtendida de @OrderBy (JPA)	Colecciones

Tabla 3.3(cont.) Anotaciones para personalizar la visualización de los miembros

Puedes pensar que hay muchas anotaciones en la tabla 3.3, pero en realidad hay todavía más, porque la mayoría de estas anotaciones tienen una versión en plural para definir diferentes valores para diferentes vistas (listado 3.8).

Listado 3.8 Las anotaciones en plural definen varias veces la misma anotación

```
@DisplaySizes({ // Para usar varias veces @DisplaySize
    @DisplaySize(forViews="Simple", value=20), // name tiene 20 para display
    // size en la vista Simple
    @DisplaySize(forViews="Complete", value=40) // name tiene 40 para display
    // size en la vista Complete
})
private String name;
```

No te preocupes si aún no sabes como usar todas las anotaciones. Las irás aprendiendo poco a poco a medida que desarrolles aplicaciones OpenXava.

3.2.4 Aprende más acerca de la interfaz de usuario

Esta sección te introduce brevemente a la generación de interfaz de usuario con OpenXava. Por desgracia, muchas e interesantes cosas se nos han quedado en el tintero, como por ejemplo, los grupos y secciones anidados, la herencia de vistas, detalles sobre como usar todas las anotaciones de IU, etc.

A través de este libro aprenderás técnicas avanzadas sobre la interfaz de usuario. Adicionalmente, puedes aprender más detalles sobre este tema en el capítulo 4 (Vista) de la *Guía de Referencia* de OpenXava y en *OpenXava API Doc* de *org.openxava.annotations*.

3.3 Otras anotaciones

Aparte de las validaciones y de la interfaz de usuario, OpenXava dispone de algunas otras anotaciones interesantes. Puedes verlas en la tabla 3.4.

Usaremos la mayoría de estas anotaciones para desarrollar los ejemplos de este libro. También puedes encontrar una explicación exhaustiva en la *Guía de Referencia de OpenXava* y en *OpenXava API Doc* de `org.openxava.annotations`.

Anotación	Descripción	Aplica a
<code>@DefaultValueCalculator</code>	Para calcular el valor inicial	Propiedades y referencias
<code>@Hidden</code>	Una propiedad oculta tiene significado para el desarrollador pero no para el usuario	Propiedades
<code>@Depends</code>	Declara que una propiedad depende de otra(s)	Propiedades
<code>@Stereotype</code>	Un estereotipo es la forma de determinar un comportamiento específico para un tipo	Propiedades
<code>@Tab</code>	Define el comportamiento para la presentación de los datos tabulares (modo lista)	Entidades
<code>@SearchKey</code>	Una propiedad o referencia marcada como clave de búsqueda se usará por el usuario para buscar	Propiedades y referencias

Tabla 3.4 Otras anotaciones

3.4 Resumen

Este capítulo ha mostrado como usar anotaciones Java para hacer programación declarativa con OpenXava. Cosas como la interfaz de usuario o las validaciones, que típicamente son pura programación, se pueden hacer con tan solo anotar nuestro código.

Puede que te sientas intimidado por tantas anotaciones. No te preocunes demasiado. La mejor forma de aprender es con ejemplos, y el resto del libro es justo eso, un conjunto de ejemplos y más ejemplos que te mostrarán como usar estas anotaciones.

Empecemos a aprender.

*Empezando
con Eclipse y
PostgreSQL*

capítulo 4

Este capítulo es la génesis de tu primera aplicación. Después de una breve revisión de la aplicación que queremos desarrollar, configuraremos todas las herramientas que necesitas para desarrollar con OpenXava.

Vas a instalar PostgreSQL y Eclipse; configurar el Tomcat dentro del Eclipse y crear el proyecto para tu aplicación.

Este capítulo es una introducción a OpenXava, PostgreSQL y Eclipse, por tanto está un poco sobre explicado, especialmente en lo que a Eclipse se refiere. Por eso, si ya eres un usuario de Eclipse experimentado, simplemente echa un vistazo rápido al capítulo, y pasa directamente al siguiente.

4.1 Nuestro objetivo: Una pequeña aplicación de facturación

La aplicación para esta parte del libro es una pequeña aplicación de facturación (Invoicing) con facturas, clientes, productos y así por el estilo. Esta aplicación es una mera excusa para aprender algunos casos típicos en aplicaciones de gestión. Puedes aplicar todo lo que aprendas con esta aplicación a cualquier otra aplicación de gestión en cualquier otro dominio.

Ten en cuenta que esta aplicación de facturación es una herramienta didáctica. No la uses “tal cual” para un sistema de facturación real.

4.2 Instalar PostgreSQL

Vamos a usar PostgreSQL como base de datos para tu primera aplicación. Hemos escogido PostgreSQL porque es de código abierto, de alta calidad y muy popular.

Ve a la sección de descargas de www.postgresql.org, escoge el *pre-built binary package* para tu plataforma y descárgalo.

Primero instálalo. En el caso de Linux/Unix simplemente ejecuta el archivo descargado, como muestra el listado 4.1.

Listado 4.1 Lanzar el instalador de PostgreSQL en Linux

```
$ su  
Password:  
$ chmod +x postgresql-8.3.7-1-linux.bin  
$ ./postgresql-8.3.7-1-linux.bin
```

Si estás usando Windows Vista has de desactivar UAC para poder instalar PostgreSQL. Sigue los siguientes pasos:

- Ve al Panel de control.
- En el Panel de Control, pulsa en Cuentas de usuario.

39 Capítulo 4: Empezando con Eclipse y PostgreSQL

- En la ventana de Cuentas de usuario, pulsa en Cuentas de usuario.
- En la ventana de tareas de Cuentas de usuario, pulsa en Activar o desactivar control de Cuentas de usuario.
- Si el mensaje de Control de cuentas de usuario aparece pulsa en Continuar.
- Desmarca Usar el Control de cuentas de usuario (UAC) para ayudar a proteger el equipo, entonces pulsa en Aceptar.
- Pulsa en Reiniciar ahora para aplicar los cambios.

Para iniciar el asistente en Windows (cualquier versión) haz doble click en *postgresql-8.3.7-1-windows.exe*. Obviamente el número de versión no será el mismo en tu caso.

Ahora, solo has de seguir un simple asistente, pulsando en el botón 'Siguiente' hasta el final. La tabla 4.1 te muestra los pasos del asistente.

	Descripción	Pantallazo
1	Página de presentación	
2	Directorio de instalación para PostgreSQL. Puedes dejar el valor por defecto	Please specify the directory where PostgreSQL will be installed. Installation Directory <input type="text" value="C:\opt\PostgreSQL\8.3"/> 
3	Directorio para los datos. Puedes dejar el valor por defecto	Please select a directory under which to store your data. Data Directory <input type="text" value="C:\opt\PostgreSQL\8.3\data"/> 
4	Contraseña para el usuario 'postgres'. Teclea 'openxava', por ejemplo	Please provide a password for the database superuser (postgres). Password <input type="password" value="*****"/> Retype password <input type="password" value="*****"/>
5	El puerto. Deja el de por defecto	Please select the port number the server should listen on. Port <input type="text" value="5432"/>

Tabla 4.1 Pasos del asistente para instalar PostgreSQL

	Descripción	Pantallazo
6	La localización. Puedes dejar el valor por defecto	Select the locale to be used by the new database cluster. Locale [Default locale] ▾
7	Antes de comenzar la instalación	Setup is now ready to begin installing PostgreSQL on your computer.
8	Espera mientras PostgreSQL se instala	Please wait while Setup installs PostgreSQL on your computer. Installing Creating link /opt/PostgreSQL/8.3/share/postgresql/timezone/Poland 100 %
9	Ahora PostgreSQL está instalado	Setup has finished installing PostgreSQL on your computer.

Tabla 4.1(cont.) Pasos del asistente para instalar PostgreSQL

Después de seguir los pasos de la tabla 4.1, ya tienes el PostgreSQL instalado en tu máquina. El siguiente paso es crear una base de datos nueva y arrancar el servidor PostgreSQL. Para Linux/Unix sigue las instrucciones del listado 4.2.

Listado 4.2 Crear una nueva base de datos y arrancar PostgreSQL en linux

```
/etc/init.d/postgresql-8.3 start # Arranca la db
exit # No necesitas permisos de root a partir de ahora
/opt/PostgreSQL/8.3/bin/createdb -Upostgres invoicing # Crea una nueva base
# de datos llamada 'invoicing'
```

En Windows teclea las instrucciones mostradas en el listado 4.3 desde la línea de órdenes (cmd.exe) de tu Windows.

Listado 4.3 Crear una nueva base de datos en Windows desde la línea de órdenes

```
"C:\Program Files\PostgreSQL\8.3\bin\createdb" -Upostgres invoicing
```

El servidor PostgreSQL se ha registrado como un servicio, por tanto se inicia automáticamente cada vez que arrandas tu ordenador.

Si estás usando Windows Vista puedes activar UAC de nuevo si así lo deseas.

En este punto has creado una nueva base de datos llamada 'invoicing', y has iniciado el servidor de base de datos PostgreSQL. Tu base de datos ya está lista para usar. Creemos la aplicación.

4.3 Crear el proyecto en Eclipse

Eclipse es, sin duda, el IDE ubicuo dentro del mundo Java (junto con NetBeans). OpenXava viene “de casa” listo para usar con Eclipse. Vas a desarrollar tu aplicación de facturación usando Eclipse. Al final de este capítulo tendrás una primera versión funcional de tu aplicación desarrollada como un proyecto de Eclipse.

41 Capítulo 4: Empezando con Eclipse y PostgreSQL

Este libro asume que estás usando la edición “Eclipse IDE for Java EE Developers” de Eclipse con Java 5 o superior. Si es necesario, obtén tu Eclipse de www.eclipse.org y Java de www.java.com.

4.3.1 Instalar OpenXava

Ve a www.openxava.org y descarga la última distribución de OpenXava. Es un archivo zip, algo así como openxava-4.0.zip⁵. Simplemente descomprímelo, y tendrás un entorno listo para empezar a desarrollar. Míralo en la figura 4.1.

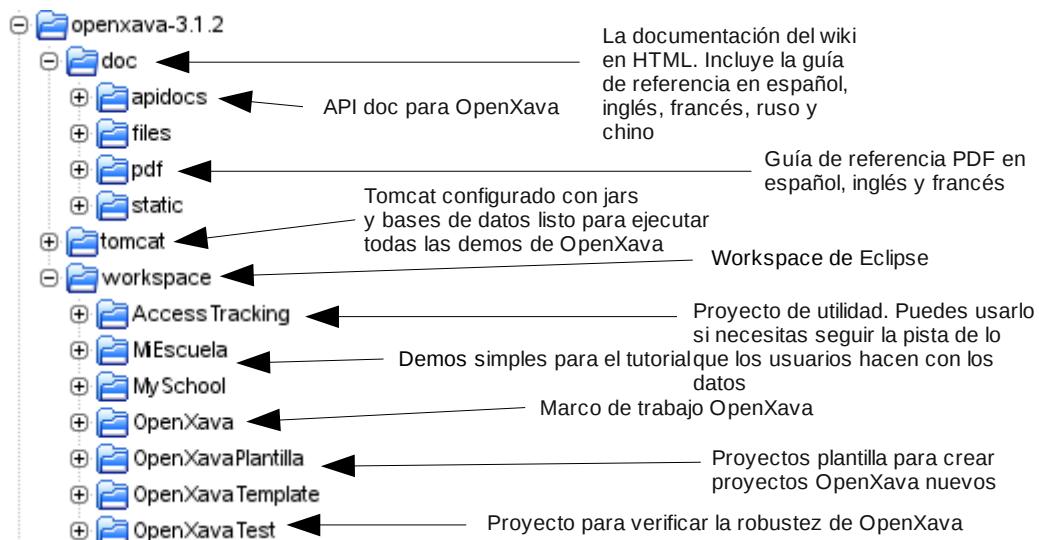


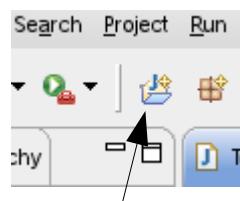
Figure 4.1 Contenido de la distribución de OpenXava

Como ves, las distribución de OpenXava incluye un Tomcat y un *workspace* de Eclipse, todo configurado y listo para usar. Empecemos a trabajar.

4.3.2 Crear el proyecto

Crear un proyecto OpenXava nuevo es simple, se trata de crear un proyecto Java convencional con Eclipse, y entonces ejecutar una tarea ant, nada más.

Arranca tu Eclipse y abre el *workspace* de OpenXava con él (*File > Switch Workspace*). En primer lugar has de crear un proyecto Java nuevo. Presiona el botón para crear un nuevo proyecto Java (figura 4.2), entonces aparecerá un asistente.



Pulsa aquí para crear un proyecto Java nuevo

Figura 4.2

⁵ Aunque los pantallazos en este libro sean de OpenXava 3.1.4, todo el código ha sido probado con OpenXava 4

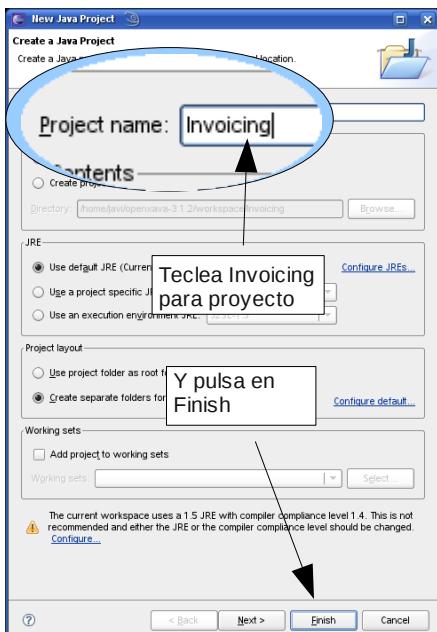


Figura 4.3 Nuevo proyecto Java

Teclea el nombre del proyecto, Invoicing, y presiona en Finish (figura 4.3). Después de esto tendrás un proyecto Java vacío **Proyecto Java vacío** llamado Invoicing (figura 4.4). Ahora tienes que convertirlo en un proyecto OpenXava. Ve a *CreateNewProject.xml* del proyecto OpenXavaTemplate, y ejecútalo como un *ant build* (figura 4.5).

Ahora, has de introducir el nombre de proyecto, teclea “Invoicing” y presiona OK (figura 4.6). Espera unos pocos segundos hasta que la tarea ant termine. Entonces selecciona el proyecto Invoicing en tu Eclipse, y presiona F5 para refrescarlo. Tienes un proyecto OpenXava completo.

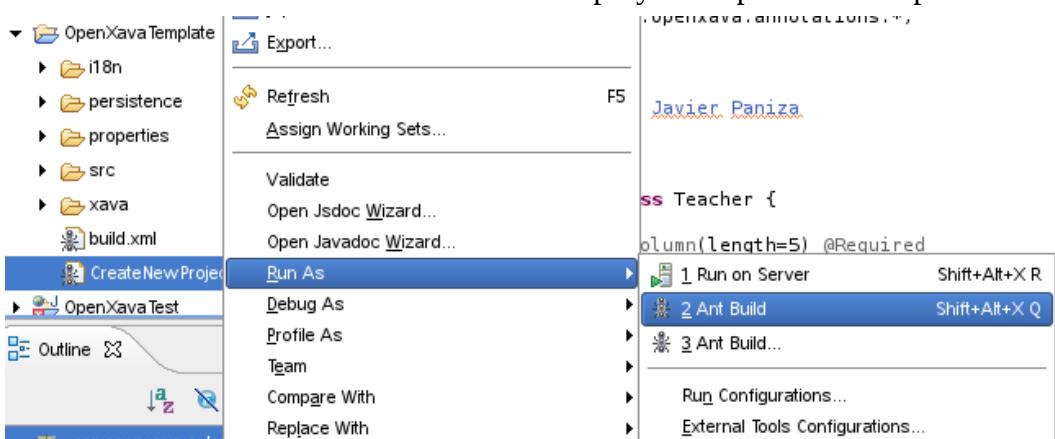
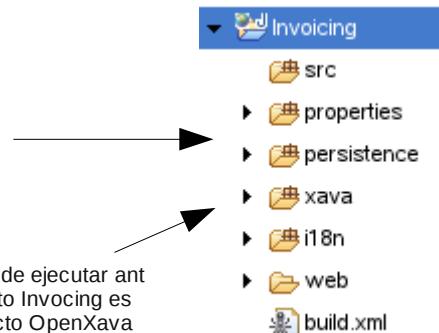


Figura 4.5 Ejecutar ant build CreateNewProject



Teclea “Invoicing” cuando ant te pregunte el nombre del proyecto



Después de ejecutar ant el proyecto Invoicing es un proyecto OpenXava

Figura 4.6 Despues de ejecutar ant tienes un proyecto OpenXava

Invoicing
 src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

build.xml

Invoicing

src

properties

persistence

xava

i18n

web

43 Capítulo 4: Empezando con Eclipse y PostgreSQL

Tu proyecto ya está listo para empezar a escribir código, sin embargo antes de eso vamos a configurar el Tomcat dentro de Eclipse para usarlo como plataforma para ejecutar tu aplicación.

4.3.3 Configurar Tomcat dentro de Eclipse

Arrancar el Tomcat desde dentro del Eclipse tiene varias ventajas, como poder depurar, ver los mensajes de log y trazas dentro del Eclipse, ir desde una traza al código con un solo click, etc.

Añadir Tomcat como servidor de ejecución a Eclipse

Ve a la opción de menú de Eclipse option *Windows > Preferences > Server > Runtime Environments*. Esto te mostrará el asistente de la figura 4.7.

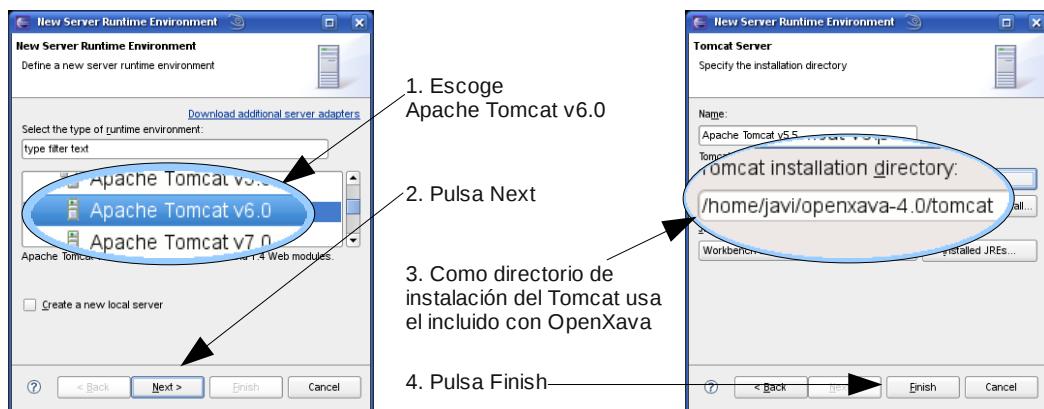


Figura 4.7 Asistente para añadir Tomcat como servidor de ejecución a Eclipse

Añadir un servidor Tomcat a Eclipse

Ya tienes el Tomcat añadido como un entorno de ejecución. Ahora has de crear un servidor que use este entorno de ejecución. Ve al menú de Eclipse *Window > Show View > Other*. Y sigue las instrucciones de la figura 4.8.

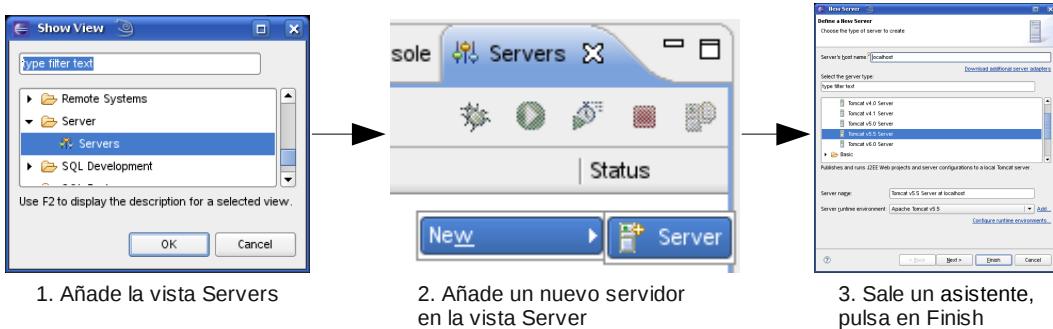


Figura 4.8 Pasos para añadir un servidor a Eclipse

Ya tienes un Tomcat configurado en tu Eclipse.

Crear la fuente de datos

Una fuente de datos es el medio que usa un servidor de aplicaciones para saber como conectarse a una base de datos. Desde nuestras aplicaciones solo referenciamos a fuentes de datos (y no directamente a las bases de datos), por tanto hemos de configurar las fuentes de datos en el Tomcat para apuntar a las bases de datos correctas. Definamos la fuente de datos para nuestra aplicación Invoicing.

Ve al proyecto Servers de Eclipse, edita el archivo *context.xml* dentro de la carpeta de tu servidor (figura 4.9).

En *context.xml* has de añadir una nueva fuente de datos llamada *InvoicingDS*, contra tu base de datos PostgreSQL. Añade el código del listado 4.4 al final de *context.xml* justo antes del último *</Context>*.

Listado 4.4 Fuente de datos a añadir en context.xml de Tomcat dentro de Eclipse

```
<Resource name="jdbc/InvoicingDS" auth="Container"
  type="javax.sql.DataSource"
  maxActive="20" maxIdle="5" maxWait="10000"
  username="postgres" password="openxava"
  driverClassName="org.postgresql.Driver"
  url="jdbc:postgresql://localhost/invoicing"/>
```

Para que esta fuente de datos funcione necesitas añadir el controlador de PostgreSQL a tu Tomcat. Abre tu navegador de internet y ve a jdbc.postgresql.org, y descarga de ahí el controlador jdbc para PostgreSQL. Es un archivo jar. Después de descargarlo, cópialo a la carpeta *lib* de tu Tomcat (en *openxava-4.x.x/tomcat*).

Añadir la aplicación Invoicing al servidor Tomcat

Acabamos de configurar el Tomcat para conectarse a tu base de datos PostgreSQL. Ahora, solo queda añadir la aplicación Invoice al Tomcat.

Para añadir tu aplicación al Tomcat ve a la vista *Servers* y sigue las instrucciones de la figura 4.10.

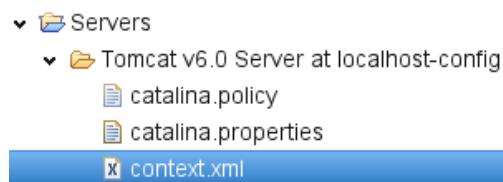


Figura 4.9 Edita context.xml para añadir fuentes de datos

45 Capítulo 4: Empezando con Eclipse y PostgreSQL

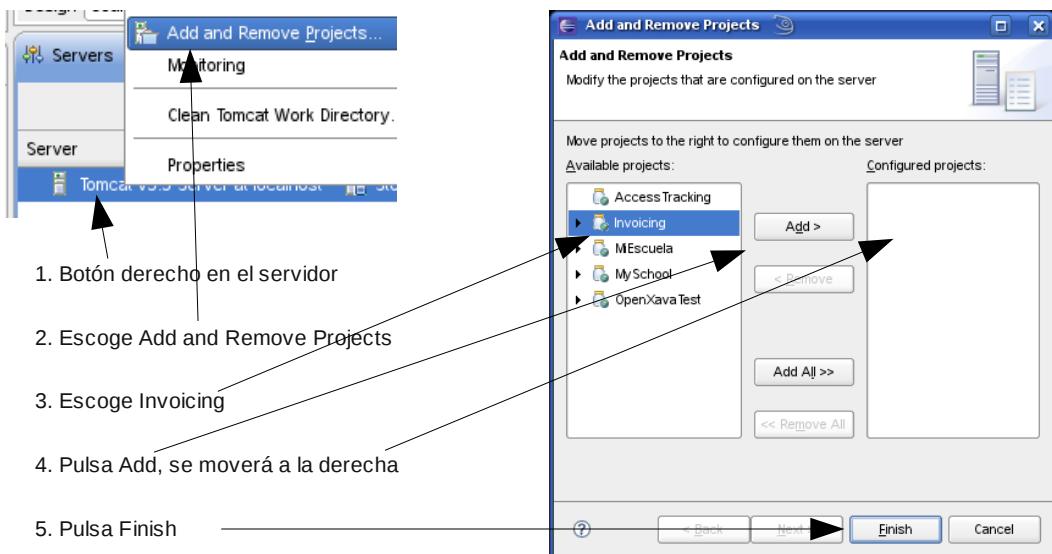


Figura 4.10 Añadir aplicación Invoicing al servidor Tomcat

Después de esto, tu proyecto ya está listo para rodar en Tomcat. Y tú estás listo para escribir tu primera entidad y ejecutar tu aplicación por primera vez.

4.3.4 Crear tu primera entidad

Al fin tienes tu entorno de desarrollo configurado. Ahora, desarrollar es muy fácil: solo has de añadir entidades para ir haciendo crecer tu aplicación. Creamos tu primera entidad y ejecutemos la aplicación.

Lo primero es crear un paquete para que contenga las clases del modelo (las entidades). Sigue las instrucciones de la figura 4.11 para crear un paquete llamado `org.openxava.invoicing.model`.

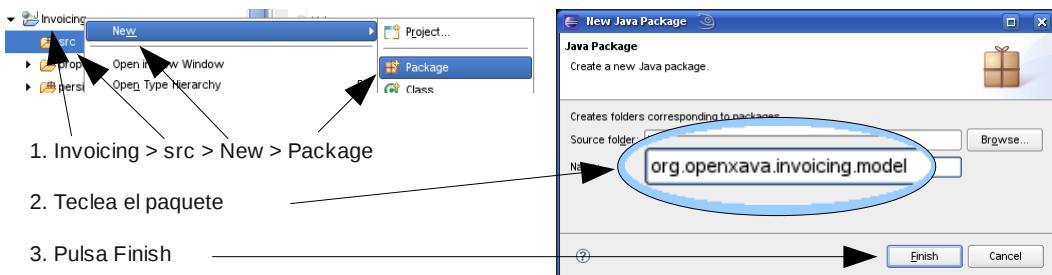


Figura 4.11 Crear un nuevo paquete 'org.openxava.invoicing.model'

Acabas de aprender a crear un paquete con Eclipse, a partir de ahora no usaremos una figura para esto.

Ahora, puedes crear tu primera entidad. Empezaremos con una versión

simplificada de Customer con solo number y description. Sigue las instrucciones de la figura 4.12 para crear una nueva clase Customer.

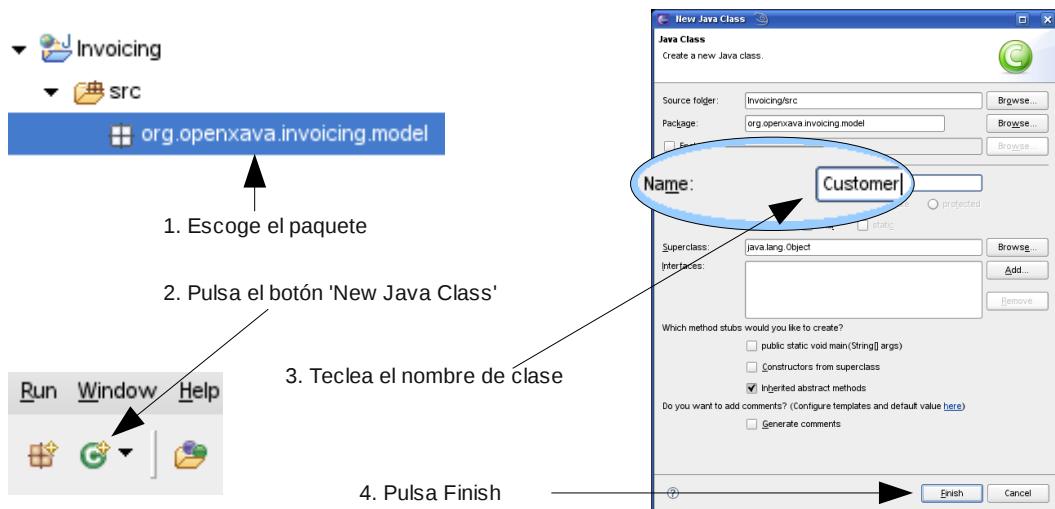


Figura 4.12 Crear la nueva clase 'Customer'

Has aprendido como crear una clase Java en Eclipse. A partir de ahora no usaremos una figura para eso.

El código inicial que Eclipse proporciona para Customer es muy simple, míralo en el listado 4.5.

Listado 4.5 La clase desnuda que inicialmente crea Eclipse para Customer

```
package org.openxava.invoicing.model;

public class Customer {
```

Ahora, te toca a ti llenar esta clase para convertirla en una entidad adecuada para OpenXava. Solo necesitas añadir la anotación @Entity y las propiedades number y description. Tal y como se ve en el listado 4.6.

Listado 4.6 La primera versión de la entidad Customer

```
package org.openxava.invoicing.model;

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

@Entity // Esto marca la clase Customer como una entidad
public class Customer {

    @Id // La propiedad number es la clave. Las claves son obligatorias (required) por defecto
    @Column(length=6) // La longitud de columna se usa a nivel UI y a nivel DB
    private int number;
```

```
@Column(length=50) ← La longitud de columna se usa a nivel UI y a nivel DB  
@Required // Se mostrará un error de validación si la propiedad name se deja en blanco  
private String name;  
  
public int getNumber() {  
    return number;  
}  
  
public void setNumber(int number) {  
    this.number = number;  
}  
  
public String getName() {  
    return name;  
}  
  
public void setName(String name) {  
    this.name = name;  
}  
  
}
```

Con esto tienes el código suficiente (justo una clase) para ejecutar tu aplicación. Hagámoslo.

4.4 Preparar la base de datos

La base de datos para tu aplicación ya está creada, aunque está vacía, por eso has de crear las tablas necesarias. Por ahora una tabla para Customer.

Primero asegúrate de que tu base de datos PostgreSQL está arrancada. Si no, iníciala usando la línea del listado 4.7 desde la línea de órdenes de tu sistema operativo.

Listado 4.7 Arrancar PostgreSQL en Linux

```
/opt/PostgreSQL/8.3/bin/postgres -D /opt/PostgreSQL/8.3/data # Arranca la db
```

En el caso de Windows PostgreSQL ha sido definido como servicio, por tanto es iniciado por defecto. Puedes iniciar y detener PostgreSQL desde *Panel de control > Sistema y mantenimiento > Herramientas administrativas > Servicios* de tu Windows.

4.4.1 Configurar persistence.xml

Has de configurar tu aplicación para que vaya contra tu base de datos. Para eso modifica el archivo *persistence.xml* que puedes encontrar en la carpeta *Invoicing/persistence/META-INF*. Edítalo, quitando las unidades de persistencia existentes y añadiendo las que hay en el listado 4.8.

Listado 4.8 persistence.xml con las unidades de persistencia configuradas

```

<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">

    <!-- Tomcat + PostgreSQL -->
    <persistence-unit name="default">
        <non-jta-data-source>java:comp/env/jdbc/InvoicingDS</non-jta-data-source>
        <class>org.openxava.session.GalleryImage</class>
        <properties>
            <!-- Dialecto de PostgreSQL -->
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.PostgreSQLDialect"/>
        </properties>
    </persistence-unit>

    <!-- JUnit PostgreSQL -->
    <persistence-unit name="junit">
        <properties>
            <!-- Clase del controlador de PostgreSQL -->
            <property name="hibernate.connection.driver_class"
                value="org.postgresql.Driver"/>

            <!-- Dialecto de PostgreSQL -->
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.PostgreSQLDialect"/>

            <property name="hibernate.connection.url"
                value="jdbc:postgresql://localhost/invoicing"/>

            <!-- El usuario de la base datos -->
            <property name="hibernate.connection.username" value="postgres"/>

            <!-- La contraseña del usuario de la base de datos -->
            <property name="hibernate.connection.password" value="openxava"/>
        </properties>
    </persistence-unit>

</persistence>

```

La unidad de persistencia **default** es la usada para acceder a la base de datos desde la aplicación, esto ocurre dentro del Tomcat. La unidad de persistencia **junit** se usa desde las pruebas junit y la herramienta “actualizar esquema”, es decir, desde dentro del Eclipse.

4.4.2 Actualizar esquema

Cada vez que hacemos un cambio en la estructura de nuestro modelo, como crear nuevas entidades, propiedades persistentes, referencias o colecciones, necesitamos actualizar el esquema de la base de datos. Por ejemplo, en este caso necesitas una tabla nueva para Customer con dos columnas para number y

49 Capítulo 4: Empezando con Eclipse y PostgreSQL

description.

Configurar updateSchema

Es posible actualizar automáticamente el esquema de la base de datos a partir de tus entidades. Esto se consigue mediante la tarea ant updateSchema de *Invoicing/build.xml*. Antes de usarla por primera vez, necesitas poner en el *path* tu controlador JDBC. Edita el archivo *Invoicing/build.xml*, busca la tarea updateSchema y editala dejándola como en el listado 4.9.

Listado 4.9 Modificar updateSchema para apuntar al controlador de PostgreSQL

```
<target name="updateSchema">
    <ant antfile="../OpenXava/build.xml" target="updateSchemaJPA">
        <property name="persistence.unit" value="junit"/>
        <!-- The path of your JDBC driver -->
        <property name="schema.path" value=
            "/openxava-4.0/tomcat/lib/postgresql-8.3-604.jdbc3.jar"
        />
    </ant>
</target>
```

Como valor para schema.path has de poner la ruta del controlador JDBC para PostgreSQL.

Ahora, updateSchema está lista para usar.

Ejecutar updateSchema

Creemos la tabla de la base de datos automáticamente. Primero pulsa Control-B para hacer un *build* de tu proyecto; después sigue las instrucciones en la figura 4.13.

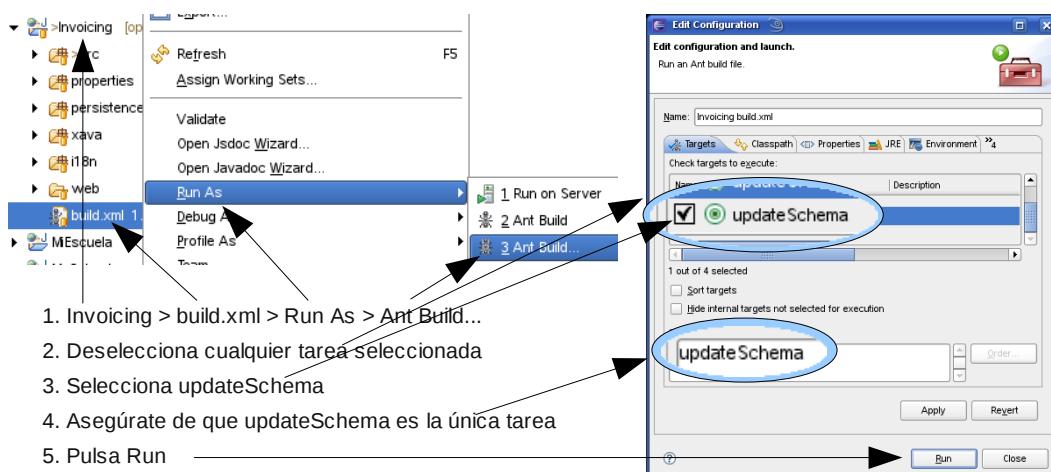
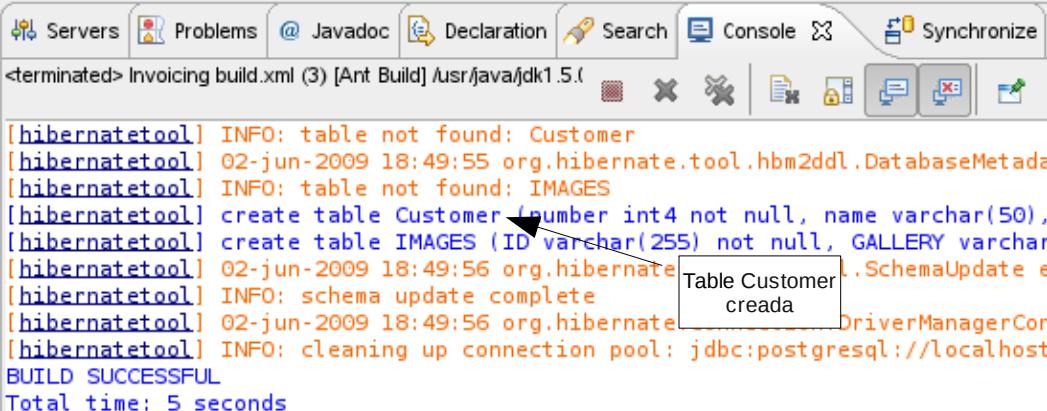


Figura 4.13 Ejecutar updateSchema

Después de ejecutar updateSchema verás una vista de consola en tu Eclipse, como la que se ve en la figura 4.14.



```
[hibernatetool] INFO: table not found: Customer
[hibernatetool] 02-jun-2009 18:49:55 org.hibernate.tool.hbm2ddl.DatabaseMetadata
[hibernatetool] INFO: table not found: IMAGES
[hibernatetool] create table Customer (number int4 not null, name varchar(50),
[hibernatetool] create table IMAGES (ID varchar(255) not null, GALLERY varchar
[hibernatetool] 02-jun-2009 18:49:56 org.hibernate.SchemaUpdate e
[hibernatetool] INFO: schema update complete
[hibernatetool] 02-jun-2009 18:49:56 org.hibernate.DriverManagerCon
[hibernatetool] INFO: cleaning up connection pool: jdbc:postgresql://localhost
BUILD SUCCESSFUL
Total time: 5 seconds
```

Figura 4.14 Resultado de schemaUpdate: Tabla Customer creada

Has aprendido como crear y actualizar el esquema de la base de datos a partir de tus entidades, a partir de ahora no usaremos una figura para esto. Recuerda que cada vez que la estructura de tus entidades cambie tienes que volver a ejecutar updateSchema. Una forma fácil de hacerlo es mediante el botón de *External Tools* (figura 4.15).

La tarea ant updateSchema está basada en Hibernate Tools.

Ahora que tienes la tabla para Customer creada puedes, al fin, ejecutar tu aplicación.

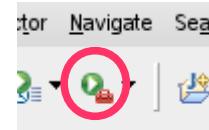


Figura 4.15
External Tools

4.5 Ejecutar la aplicación

Lo primero es iniciar tu servidor siguiendo las instrucciones de la figura 4.16.

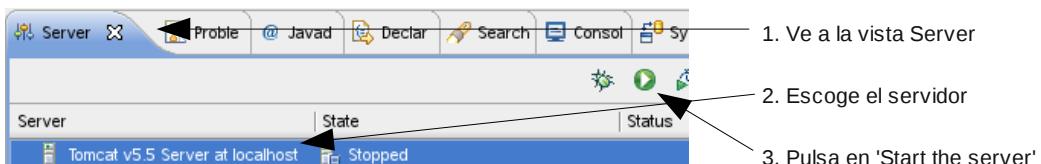


Figura 4.16 Arrancar servidor Tomcat dentro de Eclipse

Ya tienes tu aplicación ejecutándose. Para verla, abre tu navegador y ve a la URL:

<http://localhost:8080/Invoicing/modules/Customer>

Ahora, estás viendo tu aplicación por primera vez. Úsala para crear nuevos

51 Capítulo 4: Empezando con Eclipse y PostgreSQL

clientes (*customers*) como indica la figura 4.17.

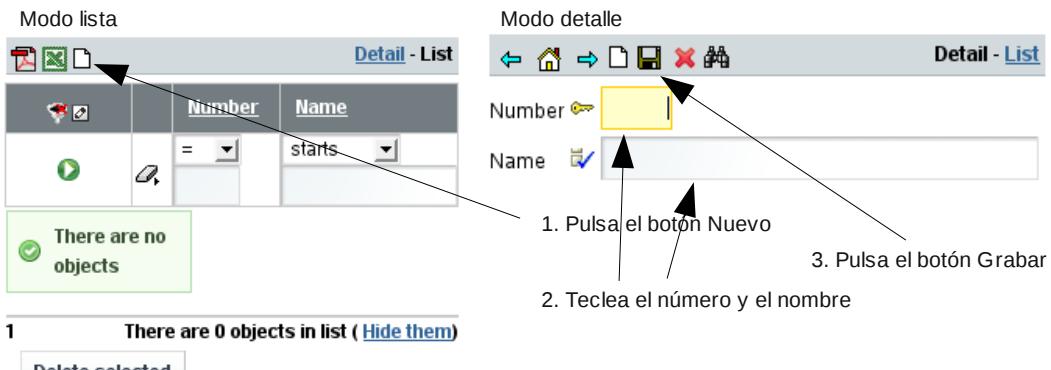


Figura 4.17 Usa el módulo Customer para añadir nuevos clientes

Enhорabuena, tienes tu entorno configurado y tu aplicación funcionando.

4.6 Modificar la aplicación

A partir de ahora, desarrollar con OpenXava es muy fácil. Simplemente, escribes una clase, actualizas el esquema de la base de datos y ya puedes ver el resultado en el navegador. Probémoslo.

Crea una nueva entidad para Product con el código mostrado en el listado 4.10. Si no recuerdas como hacerlo revisa la sección 4.3.4.

Listado 4.10 La primera versión de la entidad Product

```
package org.openxava.invoicing.model;  
  
import javax.persistence.*;  
  
import org.openxava.annotations.*;  
  
@Entity  
public class Product {  
  
    @Id @Column(length=9)  
    private int number;  
  
    @Column(length=50) @Required  
    private String description;  
  
    public int getNumber() {  
        return number;  
    }  
  
    public void setNumber(int number) {  
        this.number = number;  
    }  
}
```

```

public String getDescription() {
    return description;
}

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

}

```

Ahora, presiona Ctrl-B (para construir el proyecto), actualiza el esquema de la base de datos (sigue la instrucciones de la sección 4.4.2), abre tu navegador y ve a la URL:

<http://localhost:8080/Invoicing/modules/Product>

Sí, ya tienes un nuevo módulo en marcha, y solo has tenido que escribir una simple clase. Ahora puedes concentrarte en hacer crecer tu aplicación.

4.7 Acceder a la base de dato desde Eclipse

Aunque ya tienes a punto todas las herramientas que necesitas para desarrollar tu aplicación, a veces es útil ejecutar directamente sentencias SQL contra tu base de datos; y esto puedes hacerlo desde dentro del mismo Eclipse. Esta sección es para ayudarte a configurar el Eclipse para ello.

En primer lugar, sigue las instrucciones de la figura 4.18 para cambiar a la perspectiva *Database Development* dentro de Eclipse.

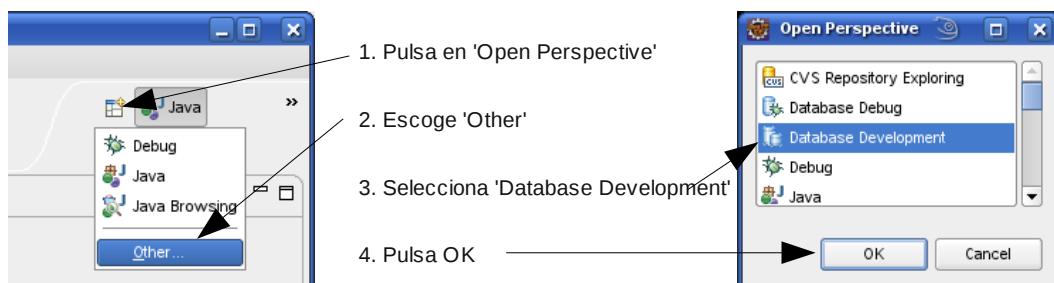


Figura 4.18 Cambiar a la perspectiva Database Development en Eclipse

53 Capítulo 4: Empezando con Eclipse y PostgreSQL

Una vez allí, crea un *New Connection Profile* siguiendo las instrucciones en la figura 4.19.

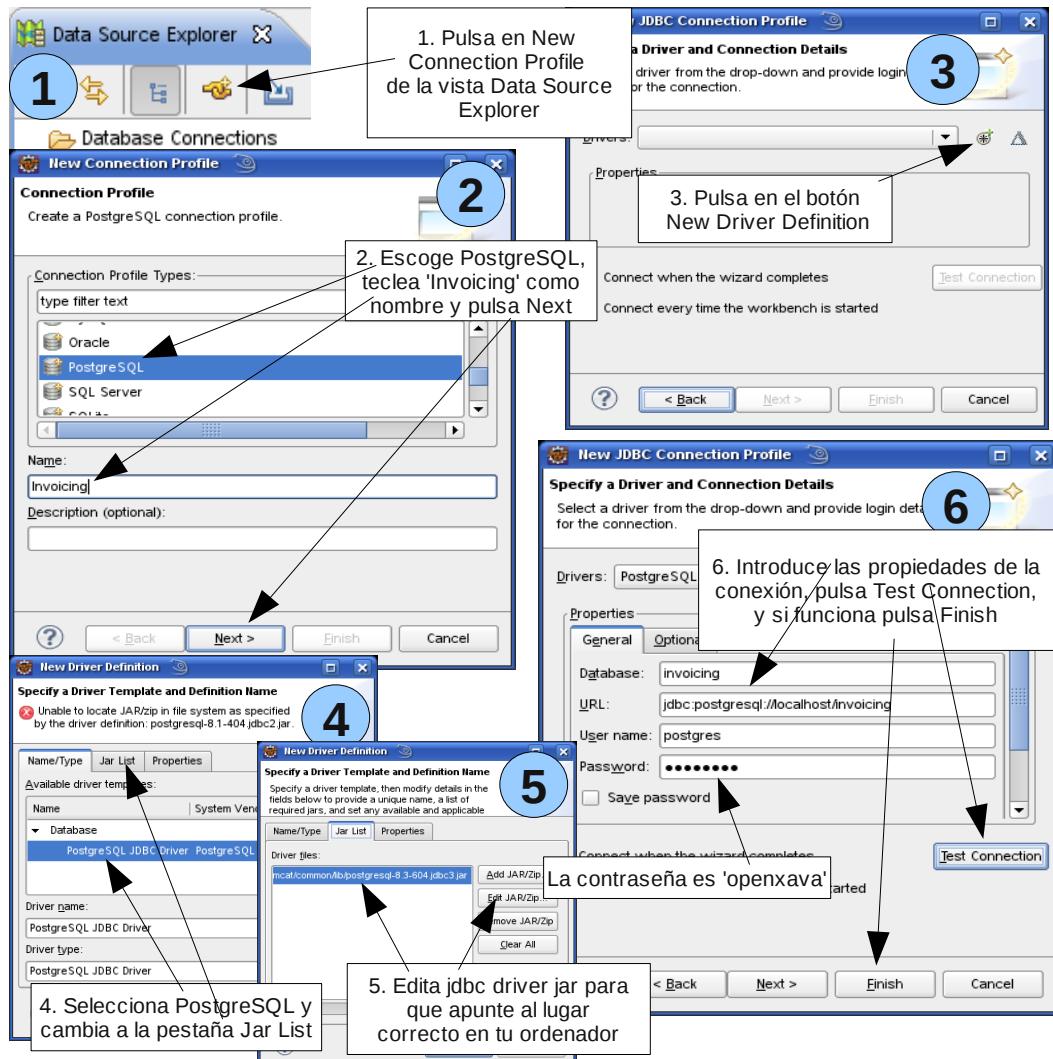


Figura 4.19 Añadir un New JDBC Connection Profile

Ahora, ya puedes ejecutar cualquier sentencia SQL que quieras, para hacerlo abre un scrapbook. Sigue las instrucciones en la figura 4.20.

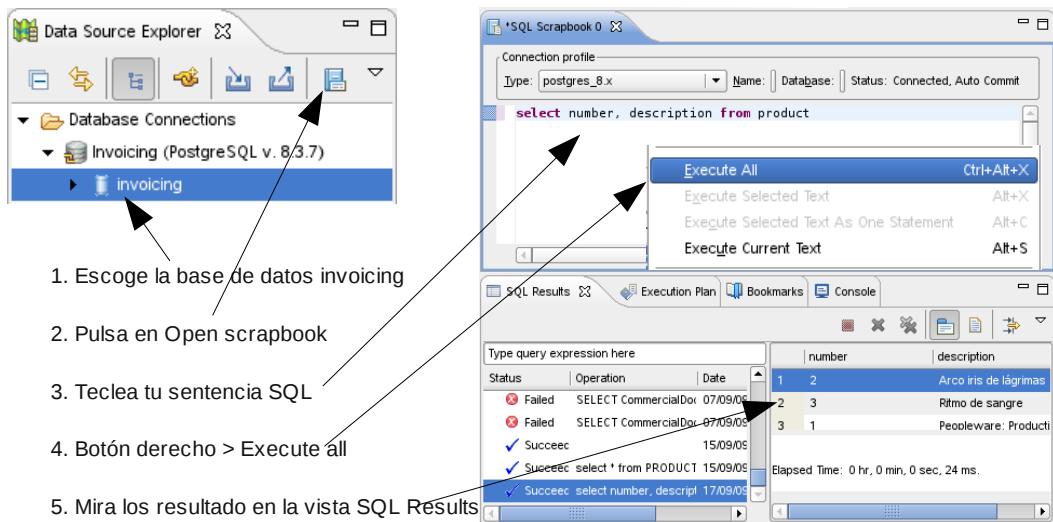


Figura 4.20 Ejecutar una sentencia SQL usando SQL scrapbook

De ahora en adelante, solo has de cambiar a la perspectiva *Database Development* en tu Eclipse cuando quieras ejecutar una sentencia SQL. Incluso, puedes añadir la vista *Data Source Explorer* a la perspectiva Java si así lo prefieres.

Esta herramienta de Eclipse para manejo de base de datos es solo una opción. Si estás acostumbrado a otra herramienta para manejar tu base de datos PostgreSQL úsalala.

4.8 Resumen

Después de este capítulo tienes instalado PostgreSQL, Eclipse y OpenXava. Además, lo tienes configurado todo para poder trabajar. Ahora, tienes tu entorno listo para desarrollar tu aplicación.

También, tienes una primera versión de tu aplicación Invoicing funcionando.

Pero lo más importante es que has aprendido cómo crear un nuevo proyecto, un nuevo paquete, una nueva clase, cómo actualizar el esquema de la base de datos, cómo ejecutar un módulo OpenXava, y algunas otras cosas útiles que usarás en el resto del libro.

*Modelar
con Java*

capítulo 5

Ahora que tienes tu entorno configurado y sabes como desarrollar con él, es hora de dar forma a tu proyecto. En este capítulo, crearás todas las entidades de tu proyecto y tendrás tu aplicación funcionando en un santiamén.

Asumo que sabes crear una nueva entidad con Eclipse, como actualizar el esquema de la base de datos cada vez que cambias tus entidades y como ejecutar la aplicación, porque ya has leído el capítulo 4.

5.1 Modelo básico del dominio

Primero crearemos las entidades para tu aplicación Invoicing. El modelo del dominio es más bien básico, pero suficiente para aprender bastantes cosas interesantes. Lo puedes ver en la figura 5.1.

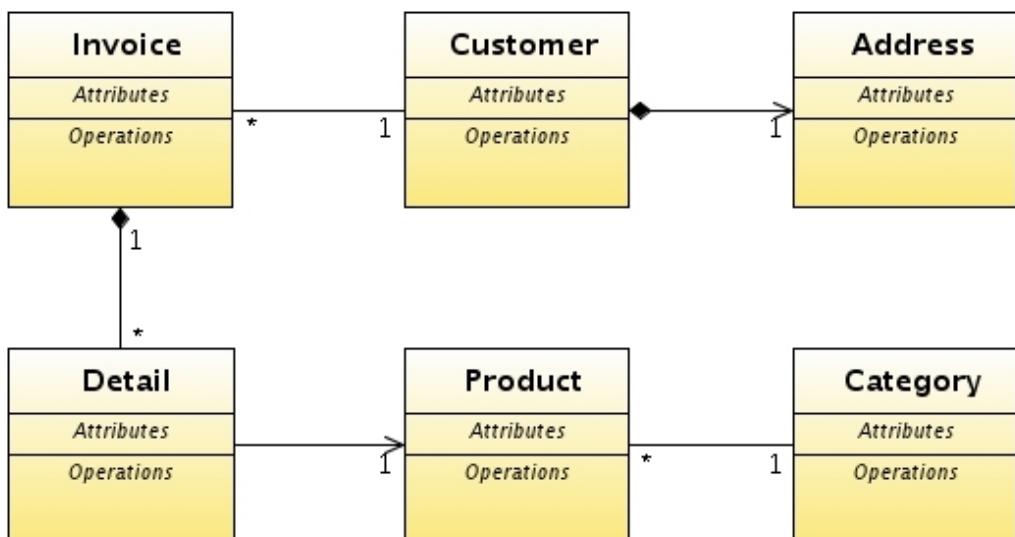


Figura 5.1 Diagrama UML inicial para la aplicación Invoicing

Empezaremos con seis clases, y más adelante añadiremos algunas más. Recuerda que ya tienes una versión inicial de Customer y Product.

5.1.1 Referencia (ManyToOne) como lista de descripciones (combo)

Empecemos con el caso más simple. Vamos a crear una entidad Category y asociarla a Product, visualizándola con un combo.

El código para la entidad Category está en el listado 5.1.

Listado 5.1 Entidad Category con generación de oid UUID

```

package org.openxava.invoicing.model;

import javax.persistence.*;

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

@Entity
public class Category {

    @Id
    @Hidden // La propiedad no se muestra al usuario. Es un identificador interno
    @GeneratedValue(generator="system-uuid") // Identificador Universal Único (1)
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid;

    @Column(length=50)
    private String description;

    // Getters y setters
    ...
}

```

Sin duda, la entidad más simple posible. Solo tiene un identificador y una propiedad `description`. En este caso usamos el algoritmo Identificador Universal Único (1) para generar el identificador. La ventaja de este generador de identificadores es que puedes migrar tu aplicación a otras bases de datos (DB2, MySQL, Oracle, Informix, etc) sin tocar tu código. Los otros generadores de identificadores de JPA usan la base de datos para generar el identificador, por lo que no son tan portables como UUID.

Ahora puedes ejecutar el módulo `Category` y añadir algunas categorías. Recuerda actualizar el esquema de la base de datos primero.

Ahora, asociaremos `Product` con `Category`. Míralo en el listado 5.2.

Listado 5.2 Product con una referencia a Category

```

@Entity
public class Product {

    @Id @Column(length=9)
    private int number;

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

    @ManyToOne( // La referencia se almacena como una relación en la base de datos
        fetch=FetchType.LAZY, // La referencia se carga bajo demanda
        optional=true) // La referencia puede estar sin valor
    @DescriptionsList // Así la referencia se visualiza usando un combo
    private Category category; // Una referencia Java convencional
}

```

```
// Getters y setters
...
}
```

Es una simple relación muchos-a-uno de JPA, como la que aprendiste en el capítulo 2. En este caso, gracias a la anotación @DescriptionsList se visualiza usando un combo (figura 5.2).

Ahora es el momento de completar la entidad Product.



Figura 5.2 Referencia a Category visualizada como combo

5.1.2 Estereotipos

La entidad Product necesita tener al menos precio, además estaría bien que tuviese fotos y un campo para observaciones. Vamos a usar estereotipos para conseguirlo. Un estereotipo especifica un uso específico de un tipo. Por ejemplo, puedes usar String para almacenar nombres, comentarios o identificadores, y puedes usar BigDecimal para almacenar porcentajes, dinero o cantidades. Es decir, hacemos diferentes usos del mismo tipo. Los estereotipo son justo para marcar este uso específico.

La mejor forma de entender que es un estereotipo es verlo en acción. Añadamos las propiedades price, photo, morePhotos y remarks a tu entidad Product (listado 5.3).

Listado 5.3 Nuevas propiedades para Product que usan @Stereotype

```
@Stereotype("MONEY") // La propiedad price se usa para almacenar dinero
private BigDecimal price; // BigDecimal se suele usar para dinero

@stereotype("PHOTO") // El usuario puede ver y cambiar una foto
private byte [] photo;

@stereotype("IMAGES_GALLERY") // Una galería de fotos completa está disponible
@Column(length=32) // La cadena de 32 de longitud es para almacenar la clave de la galería
private String morePhotos;

@stereotype("MEMO") // Esto es para un texto grande, se usará un área de texto o equivalente
private String remarks;

// Getters y setters
```

Has visto como usar estereotipos, solo has de poner el nombre del estereotipo y OpenXava hará un tratamiento especial. Si ejecutas el módulo para Product ahora verás lo que hay en la figura 5.3.

59 Capítulo 5: Modelar con Java

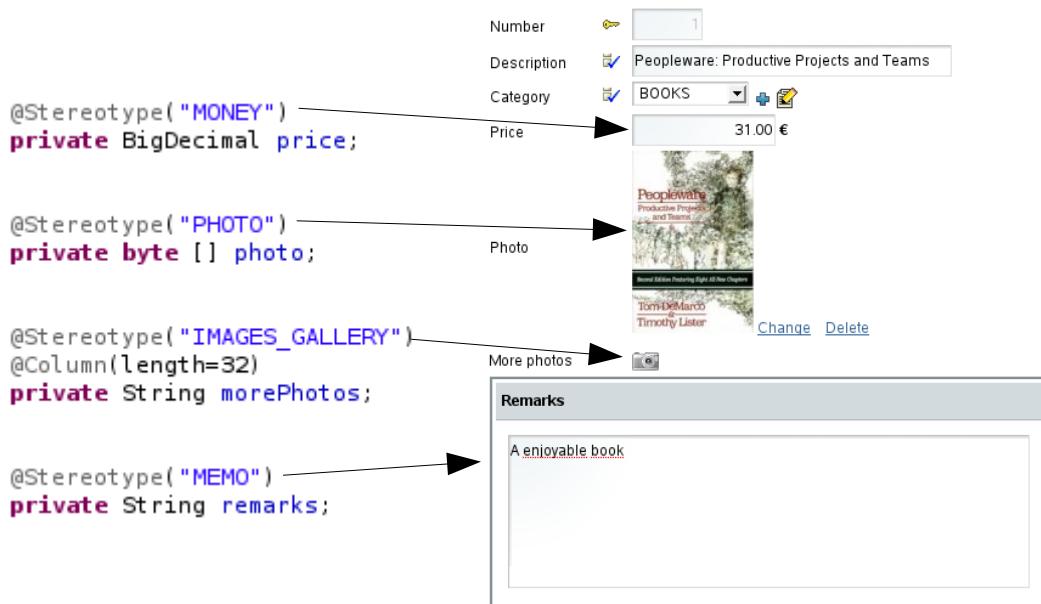


Figura 5.3 Efecto visual de los esterotipos en la interfaz de usuario

Como puedes ver en la figura 5.3, cada estereotipo produce un efecto en la interfaz de usuario. Los esterotipos tienen efecto en los tamaños, validaciones, editores, etc. Y te permiten reutilizar funcionalidad predefinida con facilidad. Por ejemplo, sólo marcando una simple propiedad `String` con `@Stereotype("IMAGES_GALLERY")`

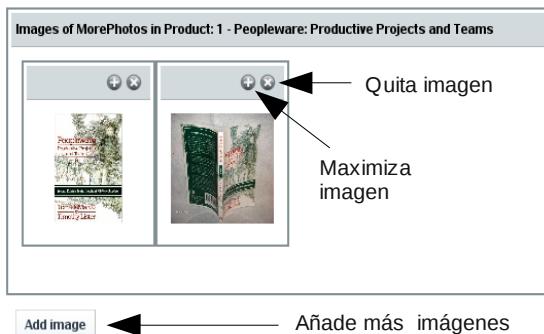


Figura 5.4 Galería de imágenes producida por el estereotipo IMAGE_GALLERY

LABEL,BOLD_LABEL, DATETIME, ZERO_FILLED, HTML_TEXT, IMAGE_LABEL, EMAIL, TELEPHONE, WEBURL, IP, ISBN, CREDIT_CARD, EMAIL_LIST.

tendrás disponible toda una galería de imágenes. Pulsando en la propiedad `morePhotos` verás la galería de la figura 5.4.

A parte de estos, OpenXava tiene muchos esterotipos predefinidos que te pueden ser útiles, tales como

5.1.3 Embeddable

Vamos a añadir una dirección (address) a nuestro, hasta ahora algo desnudo,

Customer. La dirección del Customer no está compartida por otros objetos Customer, y cuando un Customer se borra, su dirección (address) es borrada también, por lo tanto modelaremos el concepto de dirección como una clase incrustable. Aprendiste a hacer esto en la sección 2.1.4 (del capítulo 2).

Añade la clase Address a tu proyecto. Su código está en listado 5.4.

Listado 5.4 Address se ha modelado como una clase incrustable (embeddable)

```
@Embeddable // Usamos @Embeddable en vez de @Entity
public class Address {

    @Column(length=30) // Los miembros se anotan igual que en las entidades
    private String street;

    @Column(length=5)
    private int zipCode;

    @Column(length=20)
    private String city;

    @Column(length=30)
    private String state;

    // Getters y setters
    ...
}
```

Como ves, es una clase normal y corriente anotada como @Embeddable. Sus propiedades se anotan de la misma manera que en las entidades, aunque las clases incrustables no soportan toda la funcionalidad de las entidades.

Ahora, puedes usar Address en cualquier entidad. Simplemente añade una referencia a ella en tu entidad Customer, dejándola como en el listado 5.5.

Listado 5.5 La entidad Customer con una referencia a la incrustable Address

```
@Entity
public class Customer {

    @Id
    @Column(length=6)
    private int number;

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

    @Embedded // Así para referenciar a una clase incrustable
    private Address address; // Una referencia Java convencional

    public Address getAddress() {
        if (address == null) address = new Address(); // Así nunca es nulo
        return address;
    }
}
```

61 Capítulo 5: Modelar con Java

```
public void setAddress(Address address) {  
    this.address = address;  
}  
  
// Otros getters y setters  
...  
}
```

Los datos de Address se almacenan en la misma tabla que los de Customer. Y desde una perspectiva de la interfaz de usuario hay un marco alrededor de Address, aunque si no te gusta el marco solo has de anotar la referencia con @NoFrame, como muestra el listado 5.6.

Listado 5.6 Address con @NoFrame

```
@Embedded @NoFrame // Con @NoFrame no se muestra marco para address  
private Address address;
```

La figura 5.5 muestra la interfaz de usuario para una referencia incrustada con y sin @NoFrame.



Figura 5.5 Interfaz de usuario con referencias incrustables con y sin @NoFrame

Ahora que tenemos las entidades básicas en marcha, es el momento de enfrentarnos a la entidad principal de la aplicación: Invoice. Empecemos poco a poco.

5.1.4 Clave compuesta

No vamos a usar una clave compuesta para Invoice. Es mejor evitar el uso de claves compuestas. Siempre tienes la opción de usar un identificador oculto autogenerated. Aunque, algunas veces tienes la necesidad de conectarte a bases de datos legadas o puede que el diseño del esquema lo haya hecho alguien que le gustan las claves compuestas, y no tengas otra opción que usar claves compuestas aunque no sea lo ideal. Por lo tanto, vamos a aprender como usar una clave compuesta, aunque al final cambiaremos a una clave simple autogenerated.

Empecemos con una versión sencilla de la entidad Invoice. Mírala en el listado 5.7.

Listado 5.7 Primera versión de Invoice, con clave compuesta

```

@Entity
@IdClass(InvoiceKey.class) // La clase id contiene todas las propiedades clave (1)
public class Invoice {

    @Id // Aunque tenemos la clase id aún es necesario marcarlo como @Id (2)
    @Column(length=4)
    private int year;

    @Id // Aunque tenemos la clase id aún es necesario marcarlo como @Id (2)
    @Column(length=6)
    private int number;

    @Required
    private Date date;

    @Stereotype("MEMO")
    private String remarks;

    // Getters y setters

}

```

Si quieras usar year y number como clave compuesta para Invoice, una forma de hacerlo, es marcándolos con @Id (2), y además tener una clase id (1). La clase id tiene que tener year y number como propiedades. Puedes verla en el listado 5.8.

Listado 5.8 InvoiceKey: La clase id para Invoice

```

public class InvoiceKey
    implements java.io.Serializable { // La clase key tiene que ser serializable

    private int year; // Contiene las propiedades marcadas ...
    private int number; // ... como @Id en la entidad

    @Override
    public boolean equals(Object obj) { // Ha de definir el método equals
        if (obj == null) return false;
        return obj.toString().equals(this.toString());
    }

    @Override
    public int hashCode() { // Ha de definir el método hashCode
        return toString().hashCode();
    }

    @Override
    public String toString() {
        return "InvoiceKey::" + year + ":" + number;
    }

    // Getters y setters para year y number

}

```

En el listado 5.8 se ven algunos de los requerimientos para una clase id, como

63 Capítulo 5: Modelar con Java

el ser serializable e implementar hashCode() y equals().

Has visto como usar una clave compuesta. Pero dado que tenemos control sobre nuestro esquema, al final vamos a usar un identificador UUID para Invoice. Reescribe la entidad Invoice y déjala como en el listado 5.9.

Listado 5.9 Invoice usando una clave simple con UUID

```
package org.openxava.invoicing.model;

import java.util.*;
import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
import org.openxava.annotations.*;

@Entity // @IdClass eliminada
public class Invoice {

    @Id @GeneratedValue(generator="system-uuid") @Hidden
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid; // Añadida como clave oculta autogenerada
                        // Acuerdate de añadir getOid() y setOid()

    @Column(length=4) // @Id quitado
    @DefaultValueCalculator(CurrentYearCalculator.class)
    private int year;

    @Column(length=6) // @Id quitado
    @DefaultValueCalculator(value=NextNumberForYearCalculator.class,
        properties=@PropertyValue(name="year"))
    private int number;

    ...
}
```

También borra la clase InvoiceKey. Usar una clave oculta autogenerada para Invoice tiene varios beneficios prácticos sobre una clave compuesta: No has de escribir la aburrida InvoiceKey, puedes modificar el número de factura sin perder ninguna asociación con otros objetos y puedes almacenar en la misma tabla pedidos (*orders*) y facturas (*invoices*) con el par año/numero repetido.

El código que tienes es suficiente para hacer funcionar el módulo Invoice. Hazlo y añade algunas facturas si quieras. Aunque todavía queda mucho trabajo por hacer en Invoice, como asignar los valores por defecto para year, number y date.

5.1.5 Calcular valores por defecto

Si has probado el módulo Invoice, habrás visto que necesitas teclear el año, el número y la fecha. Estaría bien tener valor por defecto. Es fácil de hacer usando

la anotación @DefaultValueCalculator. En el listado 5.10 ves como podemos añadir los valores por defecto para year y date:

Listado 5.10 Calculadores valor defecto incluidos en OpenXava para año y fecha

```
@Id @Column(length=4)
@DefaultValueCalculator(CurrentYearCalculator.class) // Año actual
private int year;

@Required
@DefaultValueCalculator(CurrentDateCalculator.class) // Fecha actual
private Date date;
```

A partir de ahora cuando el usuario pulse en el botón 'nuevo' el campo para año será el año actual, y el campo para la fecha la fecha actual. Estos dos calculadores (CurrentYearCalculator y CurrentDateCalculator) están incluidos en OpenXava. Explora el paquete org.openxava.calculators para ver otros calculadores predefinidos que pueden ser útiles.

Pero a veces necesitas tu propia lógica para calcular el valor por defecto. Por ejemplo, para number queremos sumar uno al último número de factura dentro de este mismo año. Crear tu propio calculador con tu lógica es fácil. Primero, crea un paquete org.openxava.invoicing.calculators para los calculadores. Y crea en él una clase NextNumberForYearCalculator, con el código del listado 5.11.

Listado 5.11 Calculador propio para el valor por defecto del número de factura

```
package org.openxava.invoicing.calculators;

import javax.persistence.*;
import org.openxava.calculators.*;
import org.openxava.jpa.*;

public class NextNumberForYearCalculator
    implements ICalculator { // Un calculador tiene que implementar ICalculator

    private int year; // Este valor se injectará (usando su setter) antes de calcular

    public Object calculate() throws Exception { // Hace el cálculo
        Query query = XPersistence.getManager() // Una consulta JPA
            .createQuery("select max(i.number) from Invoice i" +
                " where i.year = :year"); // La consulta devuelve el número de factura
        // máximo del año indicado
        query.setParameter("year", year); // Ponemos el año injectado como parámetro
        // de la consulta
        Integer lastNumber = (Integer) query.getSingleResult();
        return lastNumber == null?1:lastNumber + 1; // Devuelve el último número
        // de factura del año + 1
        // o 1 si no hay último número
    }

    public int getYear() {
        return year;
    }
}
```

```

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

}

```

Tu calculador tiene que implementar `ICalculator` (y por lo tanto tener un método `calculate()`). Declaramos una propiedad `year` para poner en ella el año del cálculo. Para implementar la lógica usamos una consulta JPA. Recuerda que aprendiste a usar JPA en el capítulo 2. Ahora solo queda anotar la propiedad `number` en la entidad `Invoice` (listado 5.12).

Listado 5.12 Propiedad number de Invoice anotada con un calculador propio

```

@Id @Column(length=6)
@DefaultValueCalculator(value=NextNumberForYearCalculator.class,
    properties=@PropertyValue(name="year") // Para inyectar el valor de year de Invoice
    // en el calculador antes de llamar a calculate()
)
private int number;

```

En este caso ves algo nuevo, `@PropertyValue`. Usándolo, estás diciendo que el valor de la propiedad `year` en la `Invoice` actual se moverá a la propiedad `year` del calculador antes de hacer el cálculo.

Ahora cuando el usuario pulse en 'nuevo' el siguiente número de factura disponible para este año estará en el campo. La forma de calcular el número de factura no es el mejor para muchos usuarios concurrentes añadiendo facturas. No te preocupes, lo mejoraremos más adelante.

Puedes ver el efecto visual del calculador para valor por defecto en la figura 5.6. Los valores por defecto son solo los valores iniciales, el usuario los puede cambiar si así lo desea.

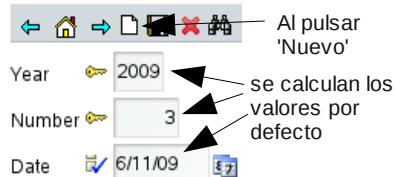


Figura 5.6 Efecto de los calculadores de valores por defecto

5.1.6 Referencias convencionales (ManyToOne)

Ahora que tenemos todas las propiedades atómicas listas para usar, es tiempo de añadir relaciones con otras entidades. Empezaremos añadiendo una referencia desde `Invoice` a `Customer`, porque una factura sin cliente no parece demasiado útil. Añade el código del listado 5.13 a la entidad `Invoice`.

Listado 5.13 Referencia a Customer desde Invoice

```

@ManyToOne(fetch=FetchType.LAZY, optional=false) // customer es obligatorio
private Customer customer;

```

```
// Getters y setters para customer
```

No hace falta más. Sin embargo, antes de probar el nuevo módulo Invoice has de borrar las facturas actuales (ejecutando 'DELETE FROM Invoice'), y entonces actualizar el esquema. Borrar las facturas que hemos añadido es necesario porque `optional=false` de la anotación `@ManyToOne` no permite facturas sin cliente⁶.

El módulo Invoice es como lo que se muestra en la figura 5.7.

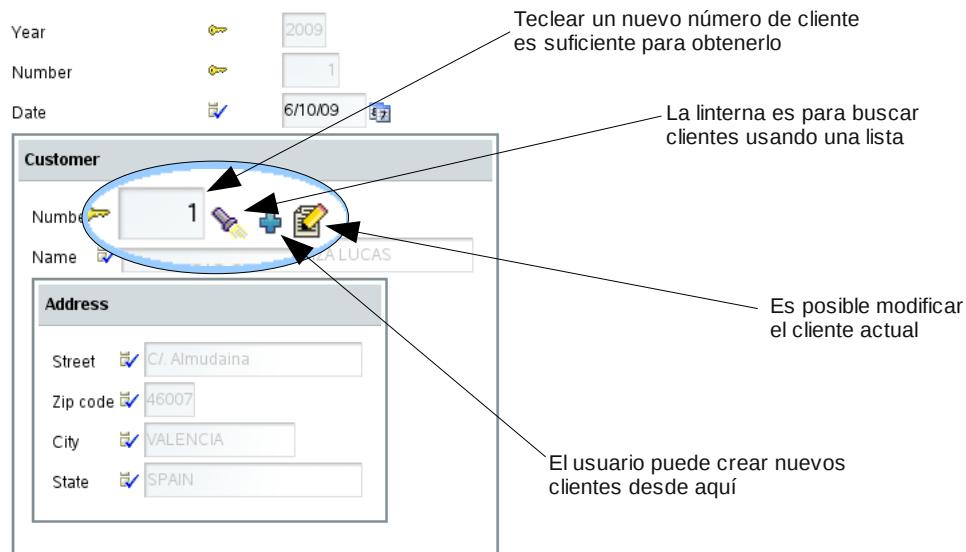


Figura 5.7 Interfaz de usuario de la referencia customer en Invoice

No hay más trabajo que hacer aquí. Añadamos una colección de líneas de detalle a Invoice.

5.1.7 Colección de entidades dependientes (`ManyToOne` con cascade)

Usualmente una factura necesita tener varias líneas con productos, cantidades, etc. Estos detalles son parte de la factura, no son compartidos por otras facturas y cuando una factura se borra sus líneas de detalle son borradas con ella. Por tanto, la forma más natural de modelar los detalles de una factura es usando objetos incrustados. Por desgracia, JPA (al menos en la 1.0) no soporta colecciones de objetos incrustables. Aunque la verdad es que esto no es un gran problema, porque puedes usar colecciones de entidades con el tipo de cascada `REMOVE` o `ALL`. Veámoslo en el caso de la definición de `details` en la entidad Invoice en

⁶ Si quieres preservar los datos actuales pon `optional=true`, actualiza el esquema, actualiza los datos de cliente en las facturas existentes y pon `optional=false` de nuevo

67 Capítulo 5: Modelar con Java

el listado 5.14.

Listado 5.14 Colección details en Invoice

```
@OneToMany( // Para declararla como una colección persistente
    mappedBy="parent", // El miembro de Detail que almacena la relación
    cascade=CascadeType.ALL) // Indica que es una colección de entidades dependientes
private Collection<Detail> details = new ArrayList<Detail>();

// Getter y setter para details
```

Usando `cascade=CascadeType.ALL` cuando la factura se borra sus líneas se borran también. O si añades una nueva línea de detalle a una factura ya persistente, la línea es grabada automáticamente, y cuando marcas una factura como persistente sus líneas se marcan como persistentes también, y así por el estilo. Podemos decir que las entidades de la colección `details` son dependientes de `Invoice`. Esto es una forma bastante buena de simular usando entidades la semántica de los objetos incrustados. También puedes definir una colección dependiente con `cascade=CascadeType.REMOVE`.

Para que esta colección funcione necesitas escribir la clase `Detail` (listado 5.15).

Listado 5.15 Primera versión de la entidad Detail para Invoice

```
package org.openxava.invoicing.model;

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

@Entity
public class Detail {

    @ManyToOne // Sin lazy fetching porque falla al quitar un detalle desde el padre
    private Invoice parent; // Así la relación entre Detail e Invoice es bidireccional

    @Id @GeneratedValue(generator="system-uuid") @Hidden
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid; // Identificador UUID, ver entidad Category (Listado 5.1)

    private int quantity;

    @ManyToOne(fetch=FetchType.LAZY, optional=true)
    private Product product;

    // Getters y setters
    ...
}
```

De momento solo tenemos `quantity` y `product`, pero es suficiente para tener la `Invoice` funcionando con `details`. Puedes ver en la figura 5.8 como el usuario puede añadir, editar y borrar elementos de la colección, filtrar y ordenar

los datos y exportar a PDF y Excel.

Pero, la figura 5.8 enfatiza el hecho de que las propiedades a mostrar por defecto son las propiedades planas, es decir, las propiedades de las referencias no se incluyen por defecto. Este hecho produce una interfaz de usuario fea para nuestra colección de líneas de factura en nuestro caso, porque solo se muestra la propiedad quantity.

Por defecto muestra solo las propiedades planas de Detail
En este caso solo la propiedad quantity

```
@OneToMany(mappedBy="parent", cascade=CascadeType.ALL)
private Collection<Detail> details;
```

Con @ListProperties el desarrollador puede especificar las propiedades a mostrar, incluyendo propiedades de referencia de cualquier nivel

```
@OneToMany(mappedBy="parent", cascade=CascadeType.ALL)
@ListProperties("product.number, product.description, quantity")
private Collection<Detail> details;
```

Number of Product	Description of Product	Quantity
	Arco iris de lágrimas	2
		1

El usuario siempre tiene la opción de personalizar las propiedades listadas

Figure 5.8 Efecto de @ListProperties en la interfaz de usuario de detalles

La figura 5.8 también muestra como puedes usar la anotación `@ListProperties` para definir propiedades a mostrar en la interfaz de usuario de la colección inicialmente. Decimos “inicialmente” porque el usuario puede personalizar las propiedades a mostrar en la colección y así adaptar los datos de la colección a sus necesidades o preferencias. `@ListProperties` es fácil de usar, míralo en el listado 5.16.

Listado 5.16 @ListProperties define las propiedades a mostrar en la colección

```
@OneToMany(mappedBy="parent", cascade=CascadeType.ALL)
@ListProperties("product.number, product.description, quantity")
private Collection<Detail> details = new ArrayList<Detail>();
```

Como puedes ver, solo has de poner como valor para `@ListProperties` la lista de las propiedades que quieras separadas por comas. Puedes usar propiedades calificadas, es decir, usar la notación del punto para acceder a las propiedades de referencias, tal como `product.number` y `product.description` en este caso.

5.2 Refinar la interfaz de usuario

¡Enhorabuena! Has finalizado tus clases del modelo del dominio, y tienes una aplicación funcionando. Tus usuarios pueden trabajar con productos, categorías, clientes e incluso crear facturas. En el caso de los productos, categorías y clientes la interfaz de usuario está bastante bien, aunque para factura todavía se puede mejorar un poco.

Ya has usado algunas anotaciones OpenXava para refinar la presentación, como `@DescriptionsList`, `@NoFrame` y `@ListProperties`. En esta sección usaremos más anotaciones de este tipo para dar a la interfaz de usuario de Invoice un mejor aspecto sin demasiado esfuerzo.

5.2.1 Interfaz de usuario por defecto

La figura 5.9 muestra cómo es la interfaz de usuario por defecto para Invoice. Como ves, OpenXava muestra todos los miembros, uno debajo de otro, en el orden en que los has declarado en el código fuente. También ves como en el caso de la referencia `customer` se usa la vista por defecto de `Customer`.

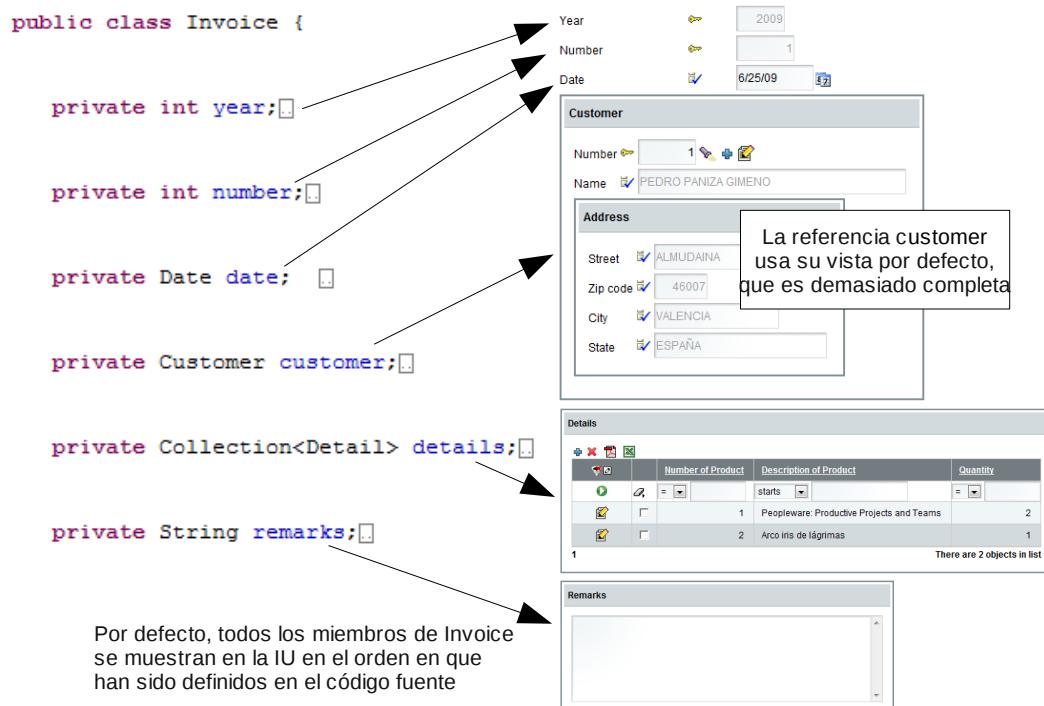


Figura 5.9 Interfaz de usuario por defecto generada automáticamente para Invoice

Vamos a hacer algunas pequeñas mejoras. Primero, definiremos la disposición de los miembros explícitamente, de esta forma podemos poner `year`, `number` y

date en la misma línea. Segundo, vamos a usar una vista más simple para customer. El usuario no necesita ver todos los datos del cliente cuando está introduciendo la factura.

5.2.2 Usar @View para definir la disposición

Para definir la disposición de los miembros de Invoice en la interfaz de usuario has de usar la anotación @View. Es fácil, solo has de enumerar los miembros a mostrar. Mira el código en el listado 5.17.

Listado 5.17 @View define la disposición de los miembros de Invoice

```
@View(members= // Esta vista no tiene nombre, por tanto será la vista usada por defecto
      "year, number, date;" + // Separados por coma significa en la misma línea
      "customer;" + // Punto y coma significa nueva línea
      "details;" +
      "remarks"
)
public class Invoice {
```

Mostramos todos los miembros de Invoice, pero usamos comas para separar year, number y date, así son mostrados en la misma línea, produciendo una interfaz de usuario más compacta, justo como puedes ver en la figura 5.10.

5.2.3 Usar @ReferenceView para refinar la interfaz de referencias

Todavía necesitas refinar la forma en que la referencia customer se visualiza, porque visualiza todos los miembros de Customer, y para introducir los datos de una Invoice una vista más simple del cliente puede ser mejor. Para hacer esto, has de definir una vista Simple en Customer, y entonces indicar en Invoice que quieres usar esa vista Simple de Customer para visualizarlo.

Primero definamos la vista Simple en Customer, como muestra el listado 5.18.

Listado 5.18 Un vista Simple para Customer, con solo number y name

```
@View(name="Simple", // Esta vista solo se usará cuando se especifique "Simple"
      members="number, name" // Muestra únicamente number y name en la misma línea
)
public class Customer {
```

Cuando una vista tiene un nombre, como en este caso, esa vista solo se usa cuando ese nombre se especifica. Es decir, aunque Customer solo tiene esta anotación @View, cuando tratas de visualizar un Customer no usará esta vista Simple, sino la generada por defecto. Si defines una @View sin nombre, esa vista

71 Capítulo 5: Modelar con Java

será la vista por defecto, aunque este no es el caso.

Ahora has de indicar que la referencia a Customer desde Invoice use esta vista Simple. Esto se hace mediante @ReferenceView. Mira el listado 5.19.

Listado 5.19 @ReferenceView para especificar la vista a usar por la referencia

```
@ManyToOne(fetch=FetchType.LAZY, optional=false)
@ReferenceView("Simple") // La vista llamada 'Simple' se usará para visualizar esta referencia
private Customer customer;
```

Realmente simple, solo has de indicar el nombre de la vista de la entidad referenciada que quieras usar.

Después de esto la referencia customer se mostrará de una forma más compacta, como se ve en la figura 5.11.



Figura 5.11 Referencia a Customer usando su vista Simple

Hemos refinado un poco la interfaz de Invoice.

5.2.4 Refinar la introducción de elementos de colección

Ya has hecho alguna mejora de la interfaz de usuario para la colección con @ListProperties. Aunque el aspecto de la colección cuando el usuario añade un nuevo elemento es todavía un tanto engorroso. Mira la figura 5.12.



Figura 5.12 Introducir una línea de Invoice es aparatoso, mejor algo más simple

La manera de simplificar la entrada de la línea de la factura es usando @View y @ReferenceView, que por cierto ya conoces..

Empecemos definiendo una vista por defecto para la entidad Detail. Justo como en el listado 5.20.

Listado 5.20 Definir una vista por defecto para Detail

```
@Entity
@View(members="product, quantity") // En la misma línea, ya que están separado por comas
public class Detail {
```

Como ves, cambiamos el orden, primero product, después quantity. Además, ponemos los dos en la misma línea. Todavía product se visualiza usando una vista demasiado grande. Puedes arreglar esto usando @ReferenceView, como muestra el listado 5.21.

Listado 5.21 Referencia a Product en Detail con @ReferenceView y @NoFrame

```
@ManyToOne(fetch=FetchType.LAZY, optional=true)
@ReferenceView("Simple") // Product se visualiza usando su vista Simple
@NoArgsConstructor // No se usa un marco alrededor de los datos de product
private Product product;
```

Nota que también usamos @NoFrame para eliminar el marco alrededor del producto, y así integrar visualmente la cantidad con el número de producto y descripción.

Por supuesto, necesitas una vista llamada Simple en la entidad Product.

73 Capítulo 5: Modelar con Java

Veámosla en el listado 5.22.

Listado 5.22 Vista Simple en Product para ser usada desde Detail

```
@Entity  
@View(name="Simple",  
      members="number, description") // number y description en la misma línea  
public class Product {
```

Ahora, el detalle contendrá en la misma línea el número de producto, la descripción del producto y la cantidad. Una interfaz limpia.

Ya que no usamos marco para product, las propiedades del producto se ven por el usuario como propiedades de Detail. Sería útil cambiar las etiquetas por defecto para number y description del producto para esta vista Simple para ser más claros de cara al usuario. Tal como muestra la figura 5.13.

Establecer las etiquetas para las propiedades de una entidad en una vista concreta es fácil. Solo necesitas editar el archivo *Invoicing-labels_en.properties* en */Invoicing/i18n*, y añadir las líneas del listado 5.23.

Listado 5.23 Entradas en Invoice-labels_en.properties para las etiquetas

```
Product.views.Simple.number=Product Nº  
Product.views.Simple.description=Product
```



Figura 5.13 Etiquetas más apropiadas para las propiedades del producto

Como ves, para definir las etiquetas se usa la notación del punto para la clave, definiendo primero el nombre de la entidad (Product en este caso), después "views", el nombre de la vista y el nombre de la propiedad. Obviamente, como valor se pone la etiqueta deseada.

5.2.5 La interfaz de usuario refinada

La figura 5.14 muestra el resultado de nuestros refinamientos en la interfaz de usuario de Invoice.

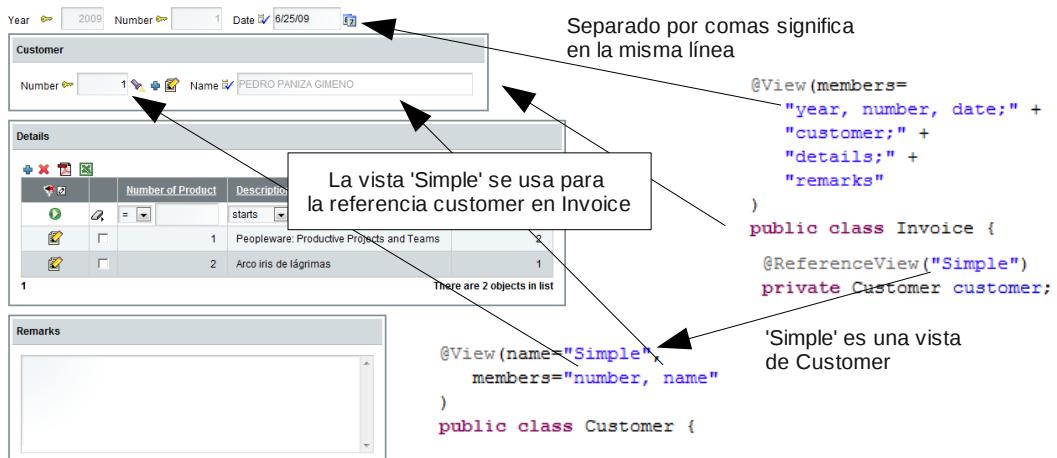


Figura 5.14 Interfaz de usuario para Invoice con @View y @ReferenceView

Has visto lo fácil que es usar @View y @ReferenceView para obtener una interfaz de usuario más compacta para Invoice.

Ahora tienes una interfaz de usuario suficientemente buena para empezar a trabajar, y realmente hemos hecho poco trabajo para conseguirlo.

5.3 Desarrollo ágil

Hoy en día el desarrollo ágil ya no es una “una técnica nueva y rompedora”, sino una forma establecida de hacer desarrollo de software, es más, es la forma ideal de desarrollar software para muchos.

Si no estás familiarizado con el desarrollo ágil puedes echar un vistazo a www.agilemanifesto.org. Básicamente, el desarrollo ágil favorece el uso de retroalimentación obtenida de un producto funcional sobre un diseño previo meticuloso. Esto da más protagonismo a los programadores y usuarios, y minimiza la importancia de los analistas y los arquitectos de software.

Este tipo de desarrollo necesita también un nuevo tipo de herramientas. Porque necesitas una aplicación funcional rápidamente. Tiene que ser tan rápido desarrollar la aplicación inicial como lo sería escribir la descripción funcional. Además, necesitas responder a las peticiones y opiniones del usuario rápidamente. El usuario necesita ver sus propuestas funcionando en corto tiempo.

OpenXava es ideal para el desarrollo ágil no sólo porque permite un desarrollo inicial muy rápido, sino porque también te permite hacer cambios y ver su efecto instantáneamente. Veamos un pequeño ejemplo de esto.

Por ejemplo, una vez que el usuario ve tu aplicación y empieza a jugar con

75 Capítulo 5: Modelar con Java

ella, se da cuenta que él trabaja con libros, música, programas y así por el estilo. Todos estos productos tienen autor, y sería útil almacenar el autor, así como ver los productos por autor.

Añadir esta nueva funcionalidad a tu aplicación es simple y rápido. Lo primero es crear una nueva clase para Author, con el código del listado 5.24.

Listado 5.24 Código para la entidad Author

```
package org.openxava.invoicing.model;

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

@Entity
public class Author {

    @Id @GeneratedValue(generator="system-uuid") @Hidden
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid;

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

    // Getters y setters
    ...
}
```

Ahora, añade el código del listado 5.25 a la ya existente entidad Product.

Listado 5.25 Referencia a Author desde Product

```
@ManyToOne(fetch=FetchType.LAZY)
@DescriptionsList
private Author author;

public void setAuthor(Author author) {
    this.author = author;
}

public Author getAuthor() {
    return author;
}
```

Así, tu entidad Product tiene una referencia a Author.

Realmente has escrito una cantidad pequeña de código. Para ver el efecto, solo necesitas construir tu proyecto (esto solo es pulsar Ctrl-B en tu Eclipse), lo cual es inmediato; actualizar el esquema de la base de datos, ejecutando la tarea ant updateSchema, solo dura unos pocos segundos. Ve al navegador y recarga la página con el módulo Product, y ahí verás, un combo para escoger el autor del producto, como muestra la figura 5.15.

¿Qué ocurre si el usuario quiere escoger un autor y ver todos sus productos? Está chupado. Solo has de hacer la relación entre Product y Author bidireccional. Ve a la clase Author y añade el código del listado 5.26.

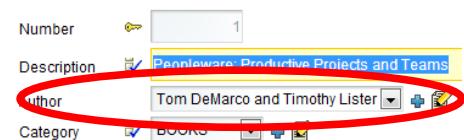


Figura 5.15 Referencia a Author desde Product

Listado 5.26 Colección products en entidad Author

```
@OneToMany(mappedBy="author")
@ListProperties("number, description, price")
private Collection<Product> products;

public Collection<Product> getProducts() {
    return products;
}

public void setProducts(Collection<Product> products) {
    this.products = products;
}
```

Ahora pulsas Ctrl-B (para construir) y refresca tu navegador con el módulo Author. Escoge un autor y verás sus productos. Tienes algo parecido a la figura 5.16. Sí, añades una nueva colección, refrescas tu navegador y tienes una interfaz de usuario completa para manejarla.

En este caso no ha sido necesario actualizar el esquema, solo has de hacerlo cuando la estructura de la base de datos cambie, y esto normalmente ocurre cuando añades, quitas o renombras campos o relaciones en las entidades; o bien cuando creas nuevas entidades. El caso de la colección de productos es una excepción, porque la columna necesaria para esta relación ya estaba en la tabla de productos. De todas formas, no te preocupes mucho por esto, ante la duda, simplemente actualiza el esquema, no hace daño.

Name	<input checked="" type="checkbox"/> Tom DeMarco and Timothy Lister	
Products		
<input type="button" value="+"/>	<input type="button" value="X"/>	
<input type="button" value="P"/>	<input type="button" value="E"/>	
<input type="button" value="G"/>	<input type="button" value="D"/>	
Number	Description	Price
<input type="text"/> 	<input type="text"/> 	<input type="text"/>
1	starts	=
<input type="text"/> 	1	31.00
<input type="text"/> 	3	35.00
1	There are 2 objects in list	

Figura 5.16 Interfaz de usuario para Author con una colección products

Esta sección ha mostrado el código completo y los pasos para hacer cambios y ver el resultado de una manera muy interactiva. Tus ojos han visto como OpenXava es una herramienta ágil, ideal para hacer desarrollo ágil.

5.4 Resumen

Este capítulo te ha enseñado como usar simples clases de Java para crear una aplicación Web. Con solo escribir clases Java para definir tu dominio, obtienes una aplicación lista para usar. También, has aprendido como refinar la interfaz de usuario por defecto usando algunas anotaciones de OpenXava.

Efectivamente tienes una aplicación funcional con poco esfuerzo. Aunque esta aplicación “tal cual” puede servir como utilidad de mantenimiento o un prototipo, todavía necesitas añadir validaciones, lógica de negocio, comportamiento de la interfaz de usuario, seguridad y así por el estilo para convertir estas entidades que has escrito en una aplicación de gestión lista para tus usuarios.

Aprenderás estos temas avanzados en los siguientes capítulos.

*Pruebas
automáticas*

capítulo 6

Las pruebas son la parte más importante del desarrollo de software. No importa cuan bonita, rápida o tecnológicamente avanzada sea tu aplicación, si falla, darás una impresión muy pobre.

Hacer pruebas manuales, es decir, abrir el navegador y ejecutar la aplicación exactamente como lo haría un usuario final, no es viable; porque el problema real no está en el código que acabas de escribir, sino en el código que ya estaba ahí. Normalmente pruebas el código que acabas de escribir, pero no pruebas todo el código que ya existe en tu aplicación. Y sabes muy bien que cuando tocas cualquier parte de tu aplicación puedes romper cualquier otra parte inadvertidamente.

Necesitas poder hacer cualquier cambio en tu código con la tranquilidad de que no vas a romper tu aplicación. Una forma de conseguirlo, es usando pruebas automáticas. Vamos a hacer pruebas automáticas usando JUnit.

6.1 JUnit

JUnit⁷ es un herramienta muy popular para hacer pruebas automáticas. Esta herramienta está integrada con Eclipse, por tanto no necesitas descargarla para poder usarla. OpenXava extiende las capacidades de JUnit para permitir probar un módulo de OpenXava exactamente de la misma forma que lo haría un usuario final. De hecho, OpenXava usa HtmlUnit⁸, un software que simula un navegador real (incluyendo JavaScript) desde Java. Todo está disponible desde la clase de OpenXava `ModuleTestBase`, que te permite automatizar las pruebas que tú harías a mano usando un navegador de verdad de una forma simple.

La mejor manera de entender como funcionan las pruebas en OpenXava es verlo en acción.

6.2 *ModuleTestBase* para probar módulos

Para crear una prueba para un módulo de OpenXava extendemos de la clase `ModuleTestBase` del paquete `org.openxava.tests`. Ésta clase te permite conectar con un módulo OpenXava como un navegador real, y tiene muchos métodos útiles para probar tu módulo. Creemos la prueba para tu módulo `Customer`.

6.2.1 *El código para la prueba*

Crea un paquete nuevo llamado `org.openxava.invoicing.tests` y dentro

⁷ <http://www.junit.org>

⁸ <http://htmlunit.sourceforge.net>

81 Capítulo 6: Pruebas automáticas

de él una nueva clase llamada CustomerTest con el código del listado 6.1.

Listado 6.1 Prueba automática JUnit para el módulo Customer

```
package org.openxava.invoicing.tests;

import org.openxava.tests.*;

public class CustomerTest extends ModuleTestCase { // Ha de extender de
// ModuleTestCase

    public CustomerTest(String testName) {
        super(testName,
              "Invoicing", // Indicamos el nombre de aplicación (Invoicing)
              "Customer"); // y nombre de módulo (Customer)
    }

    public void testCreateReadUpdateDelete() throws Exception { // Los métodos
// de prueba han de empezar por 'test'

        // Crear
        execute("CRUD.new"); // Pulsa el botón 'New'
        setValue("number", "77"); // Teclea 77 como valor para el campo 'number'
        setValue("name", "JUNIT Customer"); // Pone valor en el campo 'name'
        setValue("address.street", "JUNIT Street"); // Fíjate en la notación del punto
// para acceder al miembro de la referencia
        setValue("address.zipCode", "77555"); // Etc
        setValue("address.city", "The JUNIT city"); // Etc
        setValue("address.state", "The JUNIT state"); // Etc

        execute("CRUD.save"); // Pulsa el botón 'Save'
        assertNoErrors(); // Verifica que la aplicación no muestra errores
        assertEquals("number", ""); // Verifica que el campo 'number' está vacío
        assertEquals("name", ""); // Verifica que el campo 'name' está vacío
        assertEquals("address.street", ""); // Etc
        assertEquals("address.zipCode", ""); // Etc
        assertEquals("address.city", ""); // Etc
        assertEquals("address.state", ""); // Etc

        // Leer
        setValue("number", "77"); // Pone 77 como valor para el campo 'number'
        execute("CRUD.refresh"); // Pulsa el botón 'Refresh'
        assertEquals("number", "77"); // Verifica que el campo 'number' tiene un 77
        assertEquals("name", "JUNIT Customer"); // y 'name' tiene 'JUNIT Customer'
        assertEquals("address.street", "JUNIT Street"); // Etc
        assertEquals("address.zipCode", "77555"); // Etc
        assertEquals("address.city", "The JUNIT city"); // Etc
        assertEquals("address.state", "The JUNIT state"); // Etc

        // Actualizar
        setValue("name", "JUNIT Customer MODIFIED"); // Cambia el valor del
// campo 'name'
        execute("CRUD.save"); // Pulsa el botón 'Search'
        assertNoErrors(); // Verifica que la aplicación no muestra errores
        assertEquals("number", ""); // Verifica que el campo 'number' está vacío
        assertEquals("name", ""); // Verifica que el campo 'name' está vacío

        // Verifica si se ha modificado
        setValue("number", "77"); // Pone 77 como valor para el campo 'number'
        execute("CRUD.refresh"); // Pulsa en el botón 'Refresh'
        assertEquals("number", "77"); // Verifica que el campo 'number' tiene un 77
        assertEquals("name", "JUNIT Customer MODIFIED"); // y 'name'
```

```
// tiene 'JUNIT Customer MODIFIED'

// Borrar
execute("CRUD.delete"); // Pulsar en el botón 'Delete'
assertMessage("Customer deleted successfully"); // Verifica que el mensaje
                                                // 'Customer deleted successfully' se muestra al usuario
}

}
```

Esta prueba crea un nuevo cliente, lo busca, lo modifica y al final lo borra. Aquí ves como puedes usar métodos como `execute()` o `setValue()` para simular las acciones del usuario, y métodos como `assertValue()`, `assertNoErrors()` o `assertMessage()` para verificar el estado de la interfaz de usuario. Tu prueba actúa como las manos y los ojos del usuario. Mira la figura 6.1.

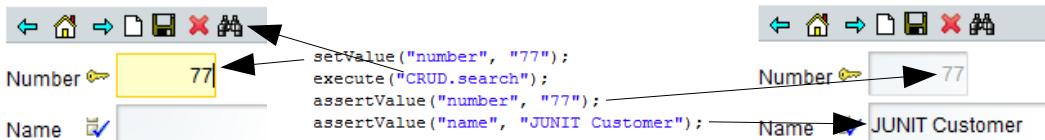


Figura 6.1 Cada línea de la prueba simula una acción del usuario o verifica el estado de la interfaz de usuario

En `execute()` tienes que especificar el nombre calificado de la acción, esto quiere decir `NombreControlador.nombreAccion`. ¿Cómo puede saber el nombre de la acción? Pasea tu ratón sobre el vínculo de la acción, y verás en la barra inferior de tu navegador un código JavaScript que incluye el nombre calificado de la acción. Tal como muestra la figura 6.2.

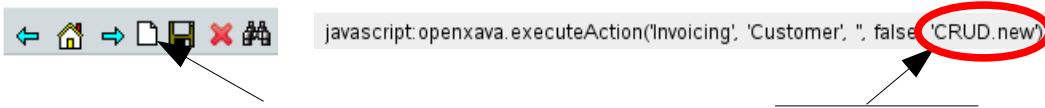


Figura 6.2 La barra inferior del navegador tiene el nombre calificado de la acción

Ahora ya sabes como crear una prueba para probar las operaciones de mantenimiento básicas de un módulo. No es necesario escribir una prueba demasiado exhaustiva al principio. Simplemente prueba las cosas básicas, aquellas cosas que normalmente probarías con un navegador. Tu prueba crecerá de forma natural a medida que tu aplicación crezca y los usuarios vayan encontrando fallos.

Aprendamos como ejecutar tu prueba desde Eclipse.

6.2.2 Ejecutar las pruebas desde Eclipse

JUnit está integrado dentro de Eclipse, por eso ejecutar tus prueba en Eclipse es más fácil que quitarle un caramelito a un niño. Pon el ratón sobre tu clase de prueba, y con el botón derecho escoge *Run As > JUnit Test*. Como muestra la figura 6.3.

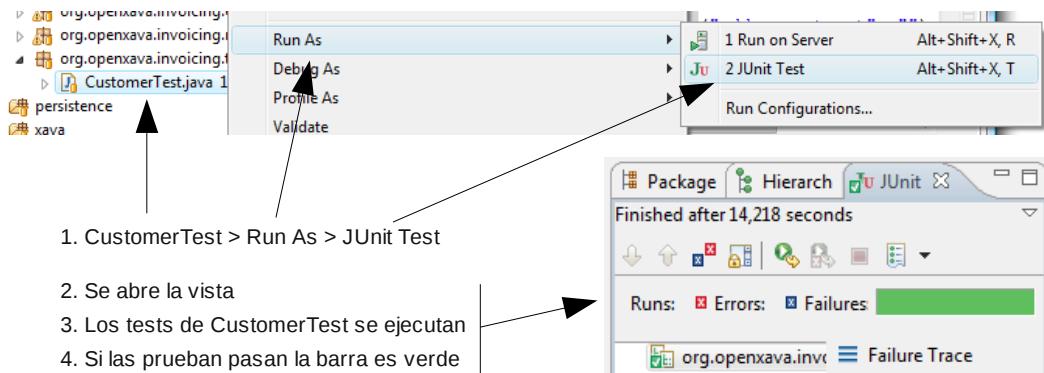


Figura 6.3 Executar pruebas JUnit desde Eclipse

Si la prueba no es satisfactoria la barra sale roja. Puedes probarlo. Edita *CustomerTest* y comenta la línea que da valor al campo *name* field. Mira el listado 6.2.

Listado 6.2 Modificar la prueba para que falle

```
...
setValue("number", "77");
// setValue("name", "JUNIT Customer"); // Comenta esta línea
setValue("address.street", "JUNIT Street");
...
```

Ahora, reejecuta la prueba. Ya que *name* es una propiedad requerida, un mensaje de error será mostrado al usuario, y el objeto no se grabará. Míralo en la figura 6.4.

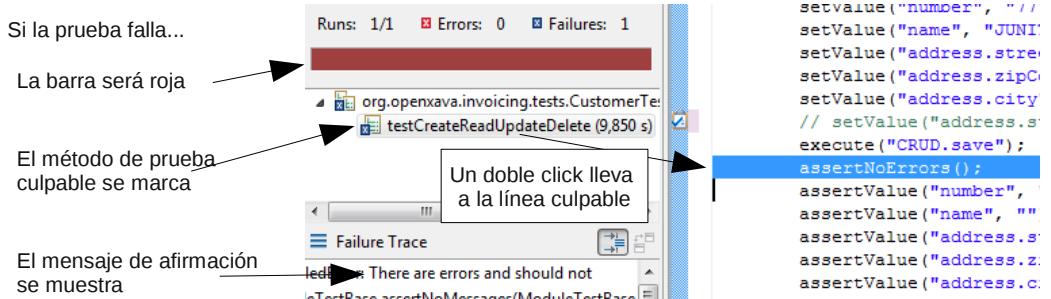


Figura 6.4 Prueba fallida ejecutada desde Eclipse

El *assert* culpable es `assertNoErrors()`, el cual además de fallar muestra en la consola los errores mostrados al usuario. Por eso, en la consola de ejecución de tu prueba verás el mensaje del listado 6.3.

Listado 6.3 Mensaje en consola cuando assertNoErrors() falla

```
16-jul-2009 18:03 org.openxava.tests.ModuleTestBase assertNoMessages
SEVERE: Error unexpected: Value for Name in Customer is required
```

El problema es claro. El cliente no se ha grabado porque el nombre es obligatorio, y éste no se ha especificado.

Has aprendido como se comporta la prueba cuando falla. Ahora, puedes descomentar la línea culpable y volver a ejecutar la prueba para verificar que todo sigue en su sitio.

6.3 Añadir el controlador JDBC al path de Java

Seguramente te habrás dado cuenta de una horrible traza de error en la consola al ejecutar tu prueba, algo así como la que hay en el listado 6.4.

Listado 6.4 Excepción “JDBC Driver not found” al ejecutar la prueba junit

```
javax.persistence.PersistenceException:
[PersistenceUnit: junit] Unable to build EntityManagerFactory
at ...
Caused by:
org.hibernate.HibernateException: JDBC Driver class not found:
org.postgresql.Driver
at ...
```

A pesar de la excepción la prueba funciona bien. OpenXava trata de conectarse a la base de datos para obtener metadatos, si no lo consigue intenta conseguir esos metadatos de otras formas, por eso la prueba funciona.

Vamos a añadir el controlador JDBC de PostgreSQL al *path* de Java de tu proyecto, de esta forma evitaremos esta horripilante traza de error, y aún más importante, tu prueba podrá acceder directamente a la base de datos. Esto es muy importante, porque es un caso común querer usar JPA o JDBC directamente desde una prueba JUnit.

Para añadir el controlador JDBC de PostgreSQL JDBC al *Java Build Path* de tu proyecto sigue las instrucciones de la figura 6.5.

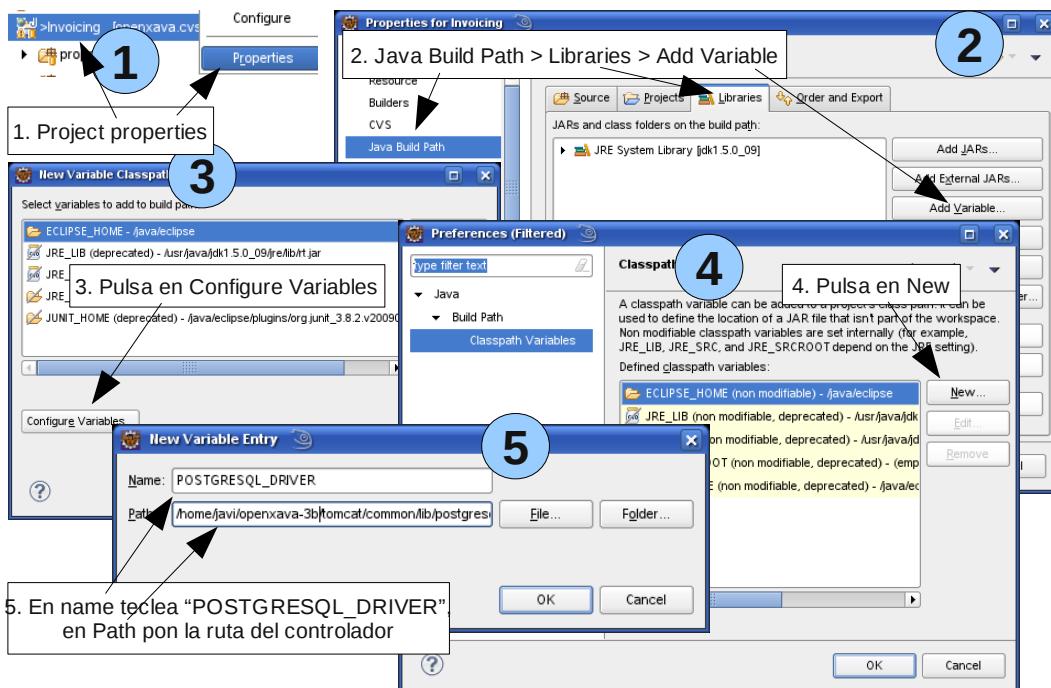


Figura 6.5 Añadir controlador JDBC de PostgreSQL al Java Build Path

Para finalizar cierra todos los diálogos pulsando en OK. En este caso hemos usado una variable *classpath* para añadir la librería al proyecto.

A partir de ahora las pruebas JUnit pueden conectarse a tu base de datos PostgreSQL.

6.4 Crear datos de prueba usando JPA

En tu primera prueba, CustomerTest, la prueba misma empieza creando los datos que van a ser usados en el resto de la prueba. Este es un buen enfoque, especialmente si quieras probar la entrada de datos también. Pero a veces te interesa probar solo un pequeño caso que falla, o simplemente tu módulo no permite entrada de datos. En cualquier caso puedes crear los datos que necesites para probar usando JPA desde tu prueba.

6.4.1 Los métodos *setUp()* y *tearDown()*

Vamos a usar ProductTest para aprender como usar JPA para crear datos de prueba. Crearemos algunos productos antes de ejecutar cada prueba y los borraremos después. Veamos el código de ProductTest en el listado 6.5.

Listado 6.5 Crear y borrar los datos de pruebas para cada caso de prueba

```

package org.openxava.invoicing.tests;

import java.math.*;

import org.openxava.invoicing.model.*;
import org.openxava.tests.*;
import static org.openxava.jpa.XPersistence.*;

public class ProductTest extends ModuleTestCase {

    private Author author; // Declaramos la entidades a crear
    private Category category; // como miembros de instancia para
    private Product product1; // que estén disponibles en todos los métodos de prueba
    private Product product2; // y puedan ser borradas al final de cada prueba

    public ProductTest(String testName) {
        super(testName, "Invoicing", "Product");
    }

    protected void setUp() throws Exception { // setUp() se ejecuta siempre
        // antes de cada prueba
        createProducts(); // Crea los datos usados en las pruebas
        super.setUp(); // Es necesario porque ModuleTestCase lo usa para inicializarse
    }

    protected void tearDown() throws Exception { // tearDown() se ejecuta
        // siempre después de cada prueba
        super.tearDown(); // Necesario, ModuleTestCase cierra recursos aquí
        removeProducts(); // Se borran los datos usado para pruebas
    }

    public void testRemoveFromList() throws Exception { ... }

    public void testChangePrice() throws Exception { ... }

    private void createProducts() { ... }

    private void removeProducts() { ... }

}

```

Aquí estamos sobrescribiendo los métodos `setUp()` y `tearDown()`. Estos métodos son métodos de JUnit que son ejecutados justo antes y después de ejecutar cada método de prueba. Creamos los datos de prueba antes de ejecutar cada prueba, y borramos los datos después de cada prueba. Así, cada prueba puede contar con unos datos concretos para ejecutarse. No importa si otras pruebas borran o modifican datos, o el orden de ejecución de las pruebas. Siempre, al principio de cada método de prueba tenemos todos los datos listos para usar.

6.4.2 Crear datos con JPA

El método `createProducts()` es el responsable de crear los datos de prueba

87 Capítulo 6: Pruebas automáticas

usando JPA. Examinémoslo en el listado 6.6.

Listado 6.6 Crear datos de prueba usando JPA

```
private void createProducts() {  
    // Crear objetos Java  
    author = new Author(); // Se crean objetos de Java convencionales  
    author.setName("JUNIT Author"); // Usamos setters como se suele hacer con Java  
    category = new Category();  
    category.setDescription("JUNIT Category");  
  
    product1 = new Product();  
    product1.setNumber(900000001);  
    product1.setDescription("JUNIT Product 1");  
    product1.setAuthor(author);  
    product1.setCategory(category);  
    product1.setPrice(new BigDecimal("10"));  
  
    product2 = new Product();  
    product2.setNumber(900000002);  
    product2.setDescription("JUNIT Product 2");  
    product2.setAuthor(author);  
    product2.setCategory(category);  
    product2.setPrice(new BigDecimal("20"));  
  
    // Marcar los objetos como persistentes  
    getManager().persist(author); // getManager() es de XPersistence  
    getManager().persist(category); // persist() marca el objeto como persistente  
    getManager().persist(product1); // para que se grabe en la base de datos  
    getManager().persist(product2);  
  
    // Confirma los cambios en la base de datos  
    commit(); // commit() es de XPersistence. Graba todos los objetos en la base de datos  
              // y confirma la transacción  
}
```

Como puedes ver, primero creas los objetos al estilo convencional de Java. Fíjate que los asignamos a miembros de instancia, así puedes usarlos dentro de la prueba. Entonces, los marcas como persistentes, usando el método `persist()` de JPA EntityManager. Para obtener el PersistenceManager solo has de escribir `getManager()` porque tienes un *import* estático arriba (listado 6.7).

Listado 6.7 Static import para facilitar el uso de `getManager()` y `commit()`

```
import static org.openxava.jpa.XPersistence.*;  
...  
getManager().persist(author);  
// Gracias al static import de XPersistence es lo mismo que  
XPersistence.getManager().persist(author);  
...  
commit();  
// Gracias al static import de XPersistence es lo mismo que  
XPersistence.commit();
```

Para finalizar, `commit()` (también de XPersistence) graba todos los objetos a la base de datos y entonces confirma la transacción. Después de eso, los datos ya

están en la base de datos listos para ser usados por tu prueba.

6.4.3 Borrar datos con JPA

Después de que se ejecute la prueba borraremos los datos de prueba para dejar la base de datos limpia. Esto se hace en el método `removeProducts()`. Lo puedes ver en el listado 6.8.

Listado 6.8 `removeProducts()/remove()` borran los datos de prueba

```
private void removeProducts() { // Llamado desde tearDown()
    // por tanto ejecutado después de cada prueba
    remove(product1, product2, author, category); // remove() borra
    commit(); // Confirma los datos en la base de datos, en este caso borrando datos
}

private void remove(Object ... entities) { // Usamos argumentos varargs
    for (Object entity: entities) { // Iteramos por todos los argumentos
        getManager().remove(getManager().merge(entity)); // Borrar(1)
    }
}
```

Es un simple bucle por todas las entidades usadas en la prueba, borrándolas. Para borrar una entidad con JPA has de usar el método `remove()`, pero en este caso has de usar el método `merge()` también (1). Esto es porque no puedes borrar una entidad desasociada (*detached entity*). Al usar `commit()` en `createProducts()` todas la entidades grabadas pasaron a ser entidades desasociadas, porque continúan siendo objetos Java válidos pero el contexto persistente (*persistent context*, la unión entre las entidades y la base de datos) se perdió en el `commit()`, por eso tienes que reasociarlas al nuevo contexto persistente. Este concepto es fácil de entender con el código del listado 6.9.

Listado 6.9 Usar `merge()` para reasociar instancias desasociadas

```
getManager().persist(author); // author está asociado al contexto persistente actual
commit(); // El contexto persistente actual se termina, y author pasa a estar desasociado

getManager().remove(author); // Falla porque author está desasociado

author = getManager().merge(author); // Reasocia author al contexto actual
getManager().remove(author); // Funciona
```

A parte de este detalle curioso sobre el `merge()`, el código para borrar es bastante sencillo.

6.4.4 Filtrar datos desde modo lista en una prueba

Ahora que ya sabes como crear y borrar datos para las pruebas, examinemos los métodos de prueba para tu módulo Product. El primero es `testRemoveFromList()` que selecciona una fila en el modo lista y pulsa en el

89 Capítulo 6: Pruebas automáticas

botón “Borrar seleccionados”. Veamos su código en el listado 6.10.

Listado 6.10 Probando el modo lista filtrando los de datos

```
public void testRemoveFromList() throws Exception {  
    setConditionValues( // Establece los valores para filtrar los datos  
        new String[] { "", "JUNIT" } );  
    setConditionComparators( // Pone los comparadores para filtrar los datos  
        new String[] { "=", "contains_comparator" } );  
    execute("List.filter"); // Pulsa el botón para filtrar  
    assertListRowCount(2); // Verifica que hay 2 filas  
    checkRow(1); // Seleccionamos la fila 1 (que resulta ser la segunda)  
    execute("CRUD.deleteSelected"); // Pulsa en el botón para borrar  
    assertListRowCount(1); // Verifica que ahora solo hay una fila  
}
```

Aquí filtramos en modo lista todos los productos que contienen la palabra “JUNIT” (recuerda que has creado dos de estos en el método `createProducts()`), entonces verificamos que hay dos filas, seleccionamos el segundo producto y lo borramos, verificando al final que la lista se queda con un solo producto.

Has aprendido como seleccionar una fila (usando `checkRow()`) y como verificar el número de filas (usando `assertListRowCount()`). Quizás la parte más intrincada es usar `setConditionValues()` y `setConditionComparators()`. Ambos métodos reciben un *array* de cadenas con valores y comparadores para la condición, tal como muestra la figura 6.6. Los valores son asignados al filtro de la lista secuencialmente (de izquierda a derecha). No es necesario que especifiques todos los valores. El método `setConditionValues()` admite cualquier cadena mientras que `setConditionComparators()` admite los siguientes valores posibles: `starts_comparator`, `contains_comparator`, `=`, `<>`, `>=`, `<=`, `>` y `<`.

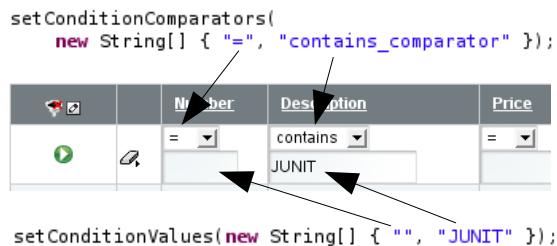


Figura 6.6 Filtrar la lista en una prueba

6.4.5 Usar instancias de entidad dentro de una prueba

La prueba que queda, `testChangePrice()`, simplemente escoge un producto y cambia su precio. Vamos a usar en él una entidad creada en `createProducts()`. El código está en el listado 6.11.

Listado 6.11 Probar usando un valor de una entidad creada para la prueba

```
public void testChangePrice() throws Exception {  
    // Buscar product1  
    execute("CRUD.new");  
    setValue("number", Integer.toString(product1.getNumber())); // (1)
```

```

execute("CRUD.refresh");
assertValue("price", "10.00");

// Cambiar el precio
setValue("price", "12.00");
execute("CRUD.save");
assertNoErrors();
assertValue("price", "");

// Verificar
setValue("number", Integer.toString(product1.getNumber())); // (1)
execute("CRUD.refresh");
assertValue("price", "12.00");
}

```

Lo único nuevo en esta prueba es que para dar valor al número usado para buscar el producto, lo obtenemos de `product1.getNumber()` (1). Recuerda que `product1` es una variable de instancia de la prueba a la que se asigna valor en `createProducts()`, el cual es llamado desde `setUp()`, es decir se ejecuta antes de cada prueba.

Ya tienes la prueba para `Product` y al mismo tiempo has aprendido como probar usando datos de prueba creados mediante JPA.

6.5 Usar datos ya existentes para probar

A veces puedes simplificar la prueba usando una base de datos que contenga los datos necesarios para la prueba. Si no quieres probar la creación de datos desde el módulo, y no borras datos en la prueba, ésta puede ser una buena opción.

Por ejemplo, puedes probar `Author` y `Category` con una prueba tan simple como la del listado 6.12.

Listado 6.12 Probar confiando en datos ya existentes en la base de datos

```

public class AuthorTest extends ModuleTestCase {

    public AuthorTest(String testName) {
        super(testName, "Invoicing", "Author");
    }

    public void testReadAuthor() throws Exception {
        assertEqualsInList(0, 0, "JAVIER CORCOBADO"); // El primer Author en la
                                                       // lista es JAVIER CORCOBADO
        execute("Mode.detailAndFirst"); // Al cambiar a modo detalle
                                       // se visualiza el primero objeto de la lista
        assertEquals("name", "JAVIER CORCOBADO");
        assertEqualsRowCount("products", 2); // Tiene 2 productos
        assertEqualsInCollection("products", 0, // Fila 0 de products
                               "number", "2"); // tiene "2" en la columna "number"
        assertEqualsInCollection("products", 0,

```

91 Capítulo 6: Pruebas automáticas

```
        "description", "Arco iris de lágrimas");
    assertEqualsInCollection("products", 1, "number", "3");
    assertEqualsInCollection("products", 1,
                           "description", "Ritmo de sangre");
}
}
```

Esta prueba verifica que el primer autor en la lista es “JAVIER CORCOBADO”, entonces va al detalle y confirma que tiene una colección llamada *products* con 2 productos: “Arco iris de lágrimas” y “Ritmo de sangre”. De paso, has aprendido como usar los métodos `assertEqualsInList()`, `assertEqualsInCollection()` y `assertCollectionRowCount()`.

Podemos usar la misma técnica para probar el módulo *Category*. Veámoslo en el listado 6.13.

Listado 6.13 Probar sólo el modo lista con datos existentes

```
public class CategoryTest extends ModuleTestBase {

    public CategoryTest(String testName) {
        super(testName, "Invoicing", "Category");
    }

    public void testCategoriesInList() throws Exception {
        assertEqualsInList(0, 0, "MUSIC"); // Fila 0 columna 0 contiene "MUSIC"
        assertEqualsInList(1, 0, "BOOKS"); // Fila 1 columna 0 contiene "BOOKS"
        assertEqualsInList(2, 0, "SOFTWARE"); // Fila 2 columna 0 contiene "SOFTWARE"
    }
}
```

En este caso solo verificamos que en la lista las tres primeras categorías son “MUSIC”, “BOOKS” y “SOFTWARE”.

Puedes ver como la técnica de usar datos preexistentes de una base de datos de prueba te permite crear pruebas más simples. Empezar con una prueba simple e ir complicándolo bajo demanda es una buena idea.

6.6 Probar colecciones

Es el momento de enfrentarnos a la prueba del módulo principal de tu aplicación, *InvoiceTest*. Por ahora la funcionalidad del módulo *Invoice* es limitada, solo puedes añadir, borrar y modificar facturas. Aun así, esta es la prueba más extensa; además contiene una colección, por tanto aprenderás como probar las colecciones.

6.6.1 Dividir la prueba en varios métodos

El listado 6.14 muestra el código de InvoiceTest.

Listado 6.14 La prueba para crear una factura está dividida en varios métodos

```
package org.openxava.invoicing.tests;

import java.text.*;
import java.util.*;
import javax.persistence.*;
import org.openxava.tests.*;
import org.openxava.util.*;

import static org.openxava.jpa.XPersistence.*; // Para usar JPA

public class InvoiceTest extends ModuleTestCase {

    private String number; // Para almacenar el número de la factura que probamos

    public InvoiceTest(String testName) {
        super(testName, "Invoicing", "Invoice");
    }

    public void testCreateInvoice() throws Exception { // El método de prueba
        verifyDefaultValues();
        chooseCustomer();
        addDetails();
        setOtherProperties();
        save();
        verifyCreated();
        remove();
    }

    private void verifyDefaultValues() throws Exception { ... }

    private void chooseCustomer() throws Exception { ... }

    private void addDetails() throws Exception { ... }

    private void setOtherProperties() throws Exception { ... }

    private void save() throws Exception { ... }

    private void verifyCreated() throws Exception { ... }

    private void remove() throws Exception { ... }

    private String getCurrentYear() { ... }

    private String getCurrentDate() { ... }

    private String getNumber() { ... }
}
```

El único método de prueba de esta clase es `testCreate()`, pero dado que es bastante extenso, es mejor dividirlo en varios métodos más pequeños. De hecho,

93 Capítulo 6: Pruebas automáticas

es una buena práctica OO⁹ escribir métodos cortos.

Ya que el método es corto puedes ver con un solo golpe de vista que es lo que hace. En este caso verifica los valores por defecto para una factura nueva, escoge un cliente, añade las líneas de detalle, añade otras propiedades, graba la factura, verifica que ha sido guardada correctamente y al final la borra. Entremos en los detalles de cada uno de estos pasos.

6.6.2 Verificar valores por defecto

Lo primero es verificar que los valores por defecto para una factura nueva son calculados correctamente. Esto se hace en el método `verifyDefaultValues()`. Está en el listado 6.15.

Listado 6.15 Verificar los valores por defecto al crear una nueva factura

```
private void verifyDefaultValues() throws Exception {
    execute("CRUD.new");
    assertEquals("year", getCurrentYear());
    assertEquals("number", getNumber());
    assertEquals("date", getCurrentDate());
}
```

Cuando el usuario pulsa en “Nuevo”, los campos año, número y fecha tienen que llenarse con datos válidos. El método `verifyDefaultValues()` precisamente comprueba esto. Usa varios métodos de utilidad para calcular los valores esperados. Los puedes ver en el listado 6.16.

Listado 6.16 Métodos de utilidad usados por `verifyDefaultValues()`

```
private String getCurrentYear() { // Año actual en formato cadena
    return new SimpleDateFormat("yyyy").format(new Date()); // La forma típica
                                                               // de hacerlo con Java
}

private String getCurrentDate() { // Fecha actual como una cadena en formato corto
    return DateFormat.getDateInstance( // La forma típica de hacerlo con Java
        DateFormat.SHORT).format(new Date());
}

private String getNumber() { // El número de factura para una factura nueva
    if (number == null) { // Usamos inicialización vaga
        Query query = getManager(). // Una consulta JPA para obtener el último número
            createQuery(
                "select max(i.number) from Invoice i where i.year = :year"
            );
        query.setParameter("year",
            Dates.getYear(new Date())); // Dates es una utilidad de OpenXava
        Integer lastNumber = (Integer) query.getSingleResult();
        if (lastNumber == null) lastNumber = 0;
        number = Integer.toString(lastNumber + 1); // Añadimos 1 al
                                                       // último número de factura
    }
}
```

9 Orientada a Objetos

```
    }  
    return number;  
}
```

Los métodos `The getCurrentYear()` y `getCurrentDate()` usan técnicas clásicas de Java para formatear la fecha como una cadena.

El método `getNumber()` es un poco más complejo: usa JPA para calcular el último número de factura del año en curso y después devuelve este valor más uno. Dado que acceder a la base de datos es más pesado que un simple cálculo Java, usamos una inicialización vaga. Una inicialización vaga retrasa el cálculo hasta la primera vez que se necesita, y después lo almacena para futuros usos. Esto lo hacemos guardando el valor en el campo `number`.

Fíjate en el uso de la clase `Dates` para extraer el año de la fecha. `Dates` es una clase de utilidad que puedes encontrar en `org.openxava.util`.

6.6.3 Entrada de datos

Ahora es el momento de `chooseCustomer()` de la factura. Mira el código en listado 6.17.

Listado 6.17 Probar escogiendo un cliente para una factura

```
private void chooseCustomer() throws Exception {
    setValue("customer.number", "1");
    assertValue("customer.name", "FRANCISCO JAVIER PANIZA LUCAS");
}
```

Al introducir el número de cliente el nombre del cliente se rellena con un valor apropiado. Con esto asociamos el cliente 1 con la factura actual.

Y ahora viene la parte más peliaguda de la prueba: añadir las líneas de detalle. Tienes el código de `addDetails()` en el listado 6.18.

Listado 6.18 Probar añadiendo elementos a la colección

```
private void addDetails() throws Exception {
    // Añadir una línea de detalle
    assertCollectionRowCount("details", 0); // La colección esta vacía
    execute("Collection.new", // Pulta en el botón para añadir un nuevo elemento
        "viewObject=xava_view_details"); // viewObject es necesario para determinar
                                         // a que colección nos referimos
    setValue("product.number", "1");
    assertEquals("product.description",
        "Peopleware: Productive Projects and Teams");
    setValue("quantity", "2");
    execute("Collection.saveAndStay"); // Graba el elemento de la colección
                                     // sin cerrar el diálogo
    assertNoErrors(); // No hay errores al grabar el detalle
    // Añadir otro detalle
```

95 Capítulo 6: Pruebas automáticas

```
    setValue("product.number", "2");
    assertEquals("product.description", "Arco iris de lágrimas");
    setValue("quantity", "1");
    execute("Collection.save"); // Graba el elemento de la colección cerrando el diálogo
    assertNoErrors();
    assertEquals("details", 2); // Ahora tenemos 2 filas
}
```

Probar una colección es exactamente igual que probar cualquier otra parte de tu aplicación, solo has de seguir los mismos pasos que un usuario haría con el navegador. Nota que has de usar `viewObject=xava_view_details` como argumento para algunas acciones de la colección.

Ahora que tenemos los detalles añadidos, vamos a llenar los datos restantes y grabar la factura. Los datos restantes se establecen en el método `setOtherProperties()` (listado 6.19).

Listado 6.19 Poner valor a las propiedades restantes antes de grabar la factura

```
private void setOtherProperties() throws Exception {
    setValue("remarks", "This is a JUNIT test");
}
```

Aquí ponemos valor al campo `remarks`. Y ahora estamos listos para grabar la factura. Lo puedes ver en listado 6.20.

Listado 6.20 Grabar la factura desde la prueba JUnit

```
private void save() throws Exception {
    execute("CRUD.save");
    assertNoErrors();

    assertEquals("customer.number", "");
    assertEquals("details", 0);
    assertEquals("remarks", "");
}
```

Simplemente pulsa en “Save”, entonces verifica que no ha habido errores y la vista se ha limpiado.

6.6.4 Verificar los datos

Ahora, buscamos la factura recién creada para verificar que ha sido grabada correctamente. Esto se hace en el método `verifyCreated()` que puedes ver en listado 6.21.

Listado 6.21 Verificar que los datos de la factura han sido grabado correctamente

```
private void verifyCreated() throws Exception {
    setValue("year", getCurrentYear()); // El año actual en el campo año
    setValue("number", getNumber()); // El número de la factura usada en la prueba
    execute("CRUD.refresh"); // Carga la factura desde la DB
```

```
// En el resto de la prueba confirmamos que los valores son los correctos
assertValue("year", getCurrentYear());
assertValue("number", getNumber());
assertValue("date", getCurrentDate());

assertValue("customer.number", "1");
assertValue("customer.name", "FRANCISCO JAVIER PANIZA LUCAS");

assertCollectionRowCount("details", 2);

// Fila 0
assertValueInCollection("details", 0, "product.number", "1");
assertValueInCollection("details", 0, "product.description",
    "Peopleware: Productive Projects and Teams");
assertValueInCollection("details", 0, "quantity", "2");

// Fila 1
assertValueInCollection("details", 1, "product.number", "2");
assertValueInCollection("details", 1, "product.description",
    "Arco iris de lágrimas");
assertValueInCollection("details", 1, "quantity", "1");

assertValue("remarks", "This is a JUNIT test");
}
```

Después de buscar la factura creada verificamos que los valores que hemos grabado están ahí. Si la prueba llega a este punto tu módulo Invoice funciona bien. Solo nos queda borrar la factura creada para que la prueba se pueda ejecutar la siguiente vez. Hacemos esto en el método remove() (listado 6.22).

Listado 6.22 Borrar la factura usada en la prueba

```
private void remove() throws Exception {
    execute("CRUD.delete");
    assertNoErrors();
}
```

Simplemente presiona en “Borrar” y verifica no se han producido errores.

¡Enhорabuena! Has completado tu InvoiceTest.

6.7 Suite

Tienes 5 casos de prueba que velan por tu código, preservando la calidad de tu aplicación. Cuando termines alguna mejora o corrección en tu aplicación ejecuta todas tus pruebas unitarias para verificar que la funcionalidad existente no se ha roto.

Tradicionalmente, para ejecutar todos las pruebas de tu aplicación deberías crear una suite de pruebas, y ejecutarla. Una suite de pruebas es una clase que agrega todas tus pruebas JUnit para que puedas ejecutarlas todas de un golpe.

97 Capítulo 6: Pruebas automáticas

Afortunadamente, si trabajas con Eclipse no necesitas escribir una clase de suite, Eclipse te permite ejecutar todas las pruebas de tu aplicación automáticamente, como muestra la figura 6.7.

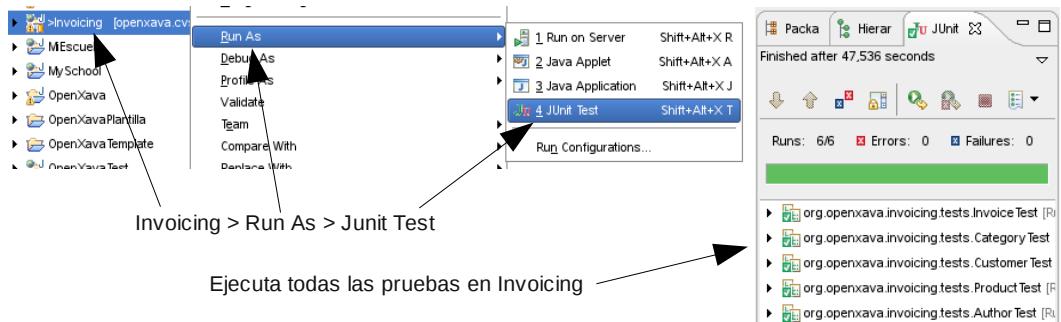


Figura 6.7 Ejecuta todas las prueba del proyecto a la vez

Es decir, si ejecutas *Run As > JUnit Test* en el proyecto, se ejecutarán todas sus pruebas JUnit.

6.8 Resumen

Has automatizado las pruebas de toda la funcionalidad actual de tu aplicación. Puede parecer que este código de prueba es mucho más largo y aburrido que el código real de la aplicación. Pero recuerda, el código de prueba es el tesoro más valioso que tienes. Quizás ahora no me creas, pero trata de hacer pruebas JUnit y una vez te hayan salvado la vida, ya no podrás desarrollar sin pruebas automáticas nunca más.

¿Qué probar? No hagas pruebas exhaustivas al principio. Es mejor probar poco que no probar nada. Si tratas de hacer pruebas muy exhaustivas acabarás no haciendo pruebas en absoluto. Empieza haciendo algunas pruebas JUnit para tu código, y con cada nueva característica o nuevo arreglo añade nuevas pruebas. Al final, tendrás una suite de pruebas muy completa. En resumen, prueba poco, pero prueba siempre.

Sí, hacer pruebas automáticas es una tarea continua. Y para predicar con el ejemplo a partir de ahora escribiremos todas las pruebas para el código que desarrollemos en el resto del libro. De esta manera aprenderás más trucos sobre las pruebas JUnit en los siguientes capítulos.

Herencia

capítulo 7

La herencia es una forma práctica de reutilizar el código en el mundo de la orientación a objetos. Usar herencia con JPA y OpenXava es tan fácil como hacerlo con puro Java. Vamos a usar la herencia para quitar el código repetitivo y aburrido, como la definición de los UUID. También, añadiremos una nueva entidad, Order, y usaremos la herencia para hacerlo con muy poco código. Además, aprenderás cuán práctico es usar herencia incluso para código de pruebas.

7.1 Heredar de una superclase mapeada

Las clases Author, Category, Detail e Invoice tienen algo de código en común. Este código es la definición del campo oid (listado 7.1) y es exactamente el mismo para todas estas clases. Ya sabes que copiar y pegar es un pecado mortal, por eso tenemos que buscar una forma de quitar este código repetido, y así evitar ir al infierno.

Listado 7.1 Propiedad oid: código común para Author, Category, Detail e Invoice

```
@Id @GeneratedValue(generator="system-uuid") @Hidden
@GenericGenerator(name="system-uuid", strategy = "uuid")
@Column(length=32)
private String oid;

public String getOid() {
    return oid;
}

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

Una solución elegante en este caso es usar herencia. JPA permite varias formas de herencia. Una de ellas es heredar de una superclase mapeada. Una superclase mapeada es una clase Java con anotaciones de mapeo JPA, pero no es una entidad en sí. Su único objetivo es ser usada como clase base para definir entidades. Usemos una, y verás su utilidad rápidamente.

Primero, movemos el código común a una clase marcada como @MappedSuperclass. La llamamos Identifiable¹⁰. La puedes ver en listado 7.2.

Listado 7.2 Identifiable: una superclase mapeada con generación de UUID

```
package org.openxava.invoicing.model;

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

¹⁰ A partir de OpenXava 4.0 la superclase Identifiable está incluida en OpenXava

101 Capítulo 7: Herencia

```
@MappedSuperclass // Marcada como una superclase mapeada en vez de como una entidad
public class Identifiable {

    @Id @GeneratedValue(generator="system-uuid") @Hidden
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid; // La definición de propiedad incluye anotaciones de OX y JPA

    public String getOid() {
        return oid;
    }

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

}
```

Ahora puedes definir las entidades Author, Category, Detail e Invoice de una manera más sucinta. Para ver un ejemplo tienes el nuevo código para Category en el listado 7.3.

Listado 7.3 Entidad Category refactorizada para extender de Identifiable

```
@Entity
public class Category extends Identifiable { // Extiende de Identifiable
    // por tanto no necesita tener una propiedad id

    @Column(length=50)
    private String description;

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

    public String getDescription() {
        return description;
    }

}
```

La refactorización es extremadamente simple. Category ahora desciende de Identifiable y hemos quitado el campo oid y los métodos getOid() y setOid(). De esta forma, no solo tu código es más corto, sino también más elegante, porque estás declarando tu clase como identifiable (el qué, no el cómo), y has quitado de tu clase de negocio un código que era un tanto técnico.

Ahora, puedes aplicar esta misma refactorización a las entidades Author, Detail e Invoice. Además, a partir de ahora extenderás la mayoría de tus entidades de la superclase mapeada Identifiable.

Has aprendido, pues, que una superclase mapeada es una clase normal y corriente con anotaciones de mapeo JPA que puedes usar como clase base para

tus entidades. También has aprendido como usar una superclase mapeada para simplificar tu código.

7.2 Herencia de entidades

Una entidad puede heredar de otra entidad. Esta herencia de entidades es una herramienta muy útil para simplificar tu modelo. Vamos a usarla para añadir una nueva entidad, Order, a tu aplicación Invoicing.

7.2.1 Nueva entidad Order

Queremos añadir un nuevo concepto a la aplicación Invoicing: pedido (*order*). Mientras que una factura es algo que quieras cobrar a tu cliente, un pedido es algo que tu cliente te ha solicitado. Estos dos conceptos están fuertemente unidos, porque cobrarás por las cosas que tu cliente te ha pedido, y tú le has servido.

Sería interesante poder tratar pedidos en tu aplicación, y asociar estos pedidos con sus correspondientes facturas. Tal como muestra el diagrama UML de la figura 7.1 y el código Java del listado 7.4.

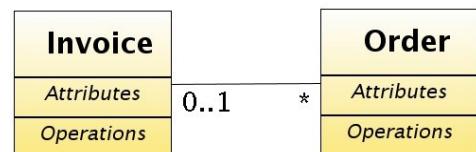


Figura 7.1 UML de Order e Invoice

Listado 7.4 Relación entre Invoice y Order

```

@Entity
public class Invoice {

    @OneToMany(mappedBy="invoice")
    private Collection<Order> orders;
    ...
}

@Entity
public class Order {

    @ManyToOne // Sin inicialización vaga (lazy fetching) (1)
    private Invoice invoice;
}
  
```

Es decir, una factura tiene varios pedidos, y un pedido puede referenciar a una factura. Fíjate como no usamos inicialización vaga (*lazy fetching*) para la referencia `invoice` (1), esto es por un bug de Hibernate cuando la referencia contiene la relación bidireccional (es decir, es la referencia declarada en el atributo `mappedBy` del `@OneToOne`).

¿Cómo es Order? Bien, tiene un cliente, unas líneas de detalle con producto y cantidad, un año y un número. Algo así como lo que hay en la figura 7.2

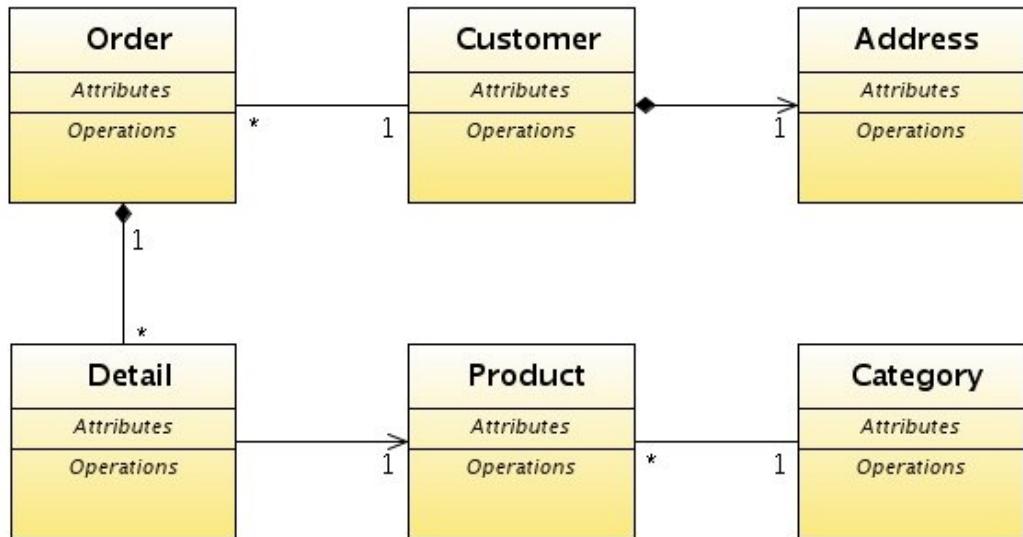


Figura 7.2 Diagrama UML para Order

Curiosamente, este diagrama UML es idéntico al diagrama de **Invoice** (que puedes ver en la figura 5.1 de la sección 5.1). Es decir, para crear tu entidad **Order** puedes copiar y pegar la clase **Invoice**, y asunto zanjado. Pero, ¡espera un momento! ¿“Copiar y pegar” no era un pecado mortal? Hemos de encontrar una forma de reutilizar **Invoice** para **Order**.

7.2.2 CommercialDocument como entidad abstracta

Una manera práctica de reutilizar el código de **Invoice** para **Order** es usando herencia, además es una excusa perfecta para aprender lo fácil que es usar la herencia con JPA y OpenXava.

En la mayoría de las culturas orientadas a objetos has de observar el precepto sagrado *es un*¹¹. Esto significa que no podemos hacer que **Invoice** herede de **Order**, porque una **Invoice** no es un **Order**, y por la misma regla no podemos hacer que **Order** descienda de **Invoice**. La solución para este caso es crear una clase base para ambos, **Order** y **Invoice**. Llamaremos a esta clase **CommercialDocument**.

En la figura 7.3 puedes ver el diagrama UML para **CommercialDocument**, y en el listado 7.5 tienes

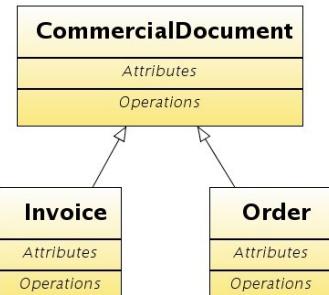


Figura 7.3 UML para CommercialDocument

¹¹ La cultura Eiffel no observa la norma *es un*. Es decir *es un* no es una regla absoluta en el universo OO

la misma idea expresada con Java.

Listado 7.5 Usar herencia para definir Order e Invoice

```
public class CommercialDocument { ... }
public class Order extends CommercialDocument { ... }
public class Invoice extends CommercialDocument { ... }
```

Empecemos a refactorizar el código actual. Primero, renombra (usando *Refactor > Rename*) Invoice como CommercialDocument. Después, edita el código de CommercialDocument para declararla como una clase abstracta, tal como muestra el listado 7.6.

Listado 7.6 CommercialDocument es una clase abstracta

```
abstract public class CommercialDocument ← // Añadimos el modificador abstract
```

Queremos crear instancias de Invoice y Order, pero no queremos crear instancias de CommercialDocument directamente, por eso la declaramos abstracta.

7.2.3 Invoice refactorizada para usar herencia

Ahora, has de crear la entidad Invoice extendiéndola de CommercialDocument. Puedes ver el nuevo código de Invoice en el listado 7.7.

Listado 7.7 La entidad Invoice ahora extiende de CommercialDocument

```
@Entity
public class Invoice extends CommercialDocument {
}
```

Invoice tiene ahora un código bastante breve, de hecho el cuerpo de la clase está, por ahora, vacío.

Este nuevo código necesita un esquema de base de datos ligeramente diferente, ahora las facturas y los pedidos se almacenarán en la misma tabla (la tabla CommercialDocument) usando una columna discriminador. Por tanto has de borrar las viejas tablas ejecutando las sentencias SQL en el listado 7.8.

Listado 7.8 Setencia SQL para borrar las viejas tablas de Invoice

```
drop table detail;
drop table invoice;
```

Puedes ejecutar estas sentencias SQL desde la perspectiva de Eclipse *Database Development*, tal como se explica en el capítulo 4 (sección 4.7).

Después de borrar las tablas viejas, ejecuta la tarea ant updateSchema para regenerar todas las tablas necesarias. Ya puedes ejecutar el módulo Invoice y

105 Capítulo 7: Herencia

verlo funcionando en tu navegador. Lanza también InvoiceTest. Tiene que salirte verde.

7.2.4 Crear Order usando herencia

Gracias a CommercialDocument el código para Order es más sencillo que el mecanismo de un sonajero. Míralo en el listado 7.9.

Listado 7.9 Entidad Order que extiende de CommercialDocument

```
@Entity  
public class Order extends CommercialDocument {  
}
```

Después de escribir la clase Order del listado 7.9, aunque de momento esté vacía, ya puedes usar el módulo Order desde tu navegador. Sí, a partir de ahora crear una nueva entidad con una estructura parecida a Invoice, es decir cualquier entidad para un documento comercial, es simple y rápido. Un buen uso de la herencia es una forma elegante de tener un código más simple.

El módulo Order funciona perfectamente, pero tiene un pequeño problema. El nuevo número de pedido lo calcula a partir del último número de factura, no de pedido. Esto es así porque el calculador para el siguiente número lee de la entidad Invoice. Una solución obvia es mover la definición de la propiedad number de CommercialDocument a Invoice y Order. Aunque, no lo vamos a hacer así, porque en el capítulo 8 refinaremos la forma de calcular el número de documento, de momento simplemente haremos un pequeño ajuste en el código actual para que no falle. Edita la clase NextNumberForYearCalculator y en la consulta cambia “Invoice” por “CommercialDocument”, dejando el método calculate() como en el listado 7.10.

Listado 7.10 NextNumberForYearCalculator.calculate() con CommercialDocument

```
public Object calculate() throws Exception {  
    Query query = XPersistence.getManager()  
        .createQuery()  
        "select max(i.number) from " +  
        "CommercialDocument i " + // CommercialDocument en vez de Invoice  
        "where i.year = :year");  
    query.setParameter("year", year);  
    Integer lastNumber = (Integer) query.getSingleResult();  
    return lastNumber == null?1:lastNumber + 1;  
}
```

Ahora buscamos el número máximo de cualquier tipo de documento comercial del año para calcular el nuevo número, por lo tanto la numeración es compartida para todos los tipos de documentos. Esto lo mejoraremos en el capítulo 8 para separar la numeración para facturas y pedidos; y para tener un mejor soporte de

entornos multiusuario.

7.2.5 Convención de nombres y herencia

Fíjate que no has necesitado cambiar el nombre de ninguna propiedad de Invoice para hacer la refactorización. Esto es por el siguiente principio práctico: *No uses el nombre de clase en los nombres de miembro*, por ejemplo, dentro de una clase Account no uses la palabra “Account” en ningún método o propiedad. Mira el listado 7.11.

Listado 7.11 No uses el nombre de la clase en los nombres de miembros

```
public class Account { // No usaremos Account en los nombres de los miembros

    private int accountNumber; // Mal, porque usa "account"
    private int number; // Bien, no usa "account"

    public void cancelAccount() { } ← // Mal, porque usa "Account"
    public void cancel() { } // Bien, no usa "account"
    ...
}
```

Usando esta nomenclatura puedes refactorizar la clase Account en una jerarquía sin renombrar sus miembros, y además puedes escribir código polimórfico con Account.

7.2.6 Asociar Order con Invoice

Hasta ahora, Order e Invoice son exactamente iguales. Vamos a hacerles sus primeras extensiones, que va a ser asociar Order con Invoice, como muestra la figura 7.4. Solo necesitas añadir una referencia desde Order a Invoice. Tienes el código para esto en listado 7.12.

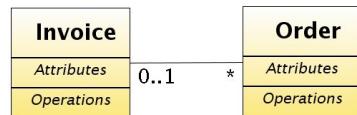


Figura 7.4 Asociación entre Invoice y Order

Listado 7.12 Código íntegro de Order con una referencia a Invoice

```
@Entity
public class Order extends CommercialDocument {

    @ManyToOne
    private Invoice invoice; // Añadida referencia a Invoice

    public Invoice getInvoice() {
        return invoice;
    }

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

Por otra parte en Invoice añadimos una colección de entidades Order. Lo puedes ver en el listado 7.13.

Listado 7.13 Código íntegro de Invoice con una colección de entidades Order

```
@Entity  
public class Invoice extends CommercialDocument {  
  
    @OneToMany(mappedBy="invoice")  
    private Collection<Order> orders; // Añadimos colección de entidades Order  
  
    public Collection<Order> getOrders() {  
        return orders;  
    }  
  
    public void setOrders(Collection<Order> orders) {  
        this.orders = orders;  
    }  
}
```

Después de escribir este código tan simple, ya puedes probar estas, recién creadas, relaciones. Primero, actualiza el esquema de la base de datos, y después abre tu navegador y explora los módulos Order e Invoice. Fíjate como al final de la interfaz de usuario de Order tienes una referencia a Invoice. El usuario puede usar esta referencia para asociar una factura al pedido actual. Por otra parte, si exploras el módulo Invoice, verás una colección de pedidos al final. El usuario puede usarla para añadir pedidos a la factura actual.

Intenta añadir pedidos a la factura, y asociar una factura a un pedido. Funciona, aunque la interfaz de usuario es un poco fea, de momento.

7.3 Herencia de vistas

La herencia no solo sirve para reutilizar código Java y mapeos, sino también para reutilizar la definición de la interfaz de usuario, las definiciones @View. Esta sección muestra como funciona la herencia de vistas.

7.3.1 El atributo `extendsView`

Tanto Order como Invoice usan una interfaz de usuario generada por defecto con todos sus miembros uno por cada línea. Nota como la @View que hemos declarado en CommercialDocument no se ha heredado. Es decir, si no defines una vista para la entidad se genera una por defecto, la @View de la entidad padre no se usa. Tal como muestra el listado 7.14.

Listado 7.14 Las vistas no se heredan por defecto

```
@View(members = "a, b, c;") // Esta vista se usa para visualizar Parent, pero no para Child
public class Parent { ... }

public class Child extends Parent { ... } // Child se visualiza usando la vista
                                         // generada automáticamente, no la vista de Parent
```

Generalmente la vista de la entidad padre “tal cual” no es demasiado útil porque no contiene todas las propiedades de la entidad actual. Por tanto este comportamiento suele venir bien como comportamiento por defecto.

Aunque, en una entidad no trivial necesitas refinar la interfaz de usuario y quizás sea útil heredar (en lugar de copiar y pegar) la vista del padre. Puedes hacer esto usando el atributo `extendsView` en `@View`. Mira el listado 7.15.

Listado 7.15 Usar herencia de vista por medio de extendsView

```
@View(members = "a, b, c;") // Esta vista sin nombre es la vista DEFAULT
public class Parent { ... }

@Views({
    @View(name="A" members = "d", // Añade d a la vista heredada
          extendsView = "super.DEFAULT"), // Extienda la vista por defecto de Parent
    @View(name="B" members = "a, b, c; d") // La vista B es igual a la vista A
})
public class Child extends Parent { ... } // Child se visualiza usando la vista
                                         // generada automáticamente, no la vista de Parent
```

Usando `extendsView` los miembros a mostrar serán aquellos en la vista que extendemos más aquellos en `members` de la actual.

Vamos a usar esta característica para definir las vistas para `CommercialDocument`, `Order` e `Invoice`.

7.3.2 Vista para `Invoice` usando herencia

Dado que la `@View` de `CommercialDocument` no se hereda, la interfaz de usuario actual para `Invoice` es bastante fea: todos los miembros, uno por línea. Vamos a definir una interfaz de usuario mejor. Una vista parecida a la del capítulo 5, pero añadiendo una pestaña para pedidos. Queremos algo como lo que muestra la figura 7.5.

109 Capítulo 7: Herencia

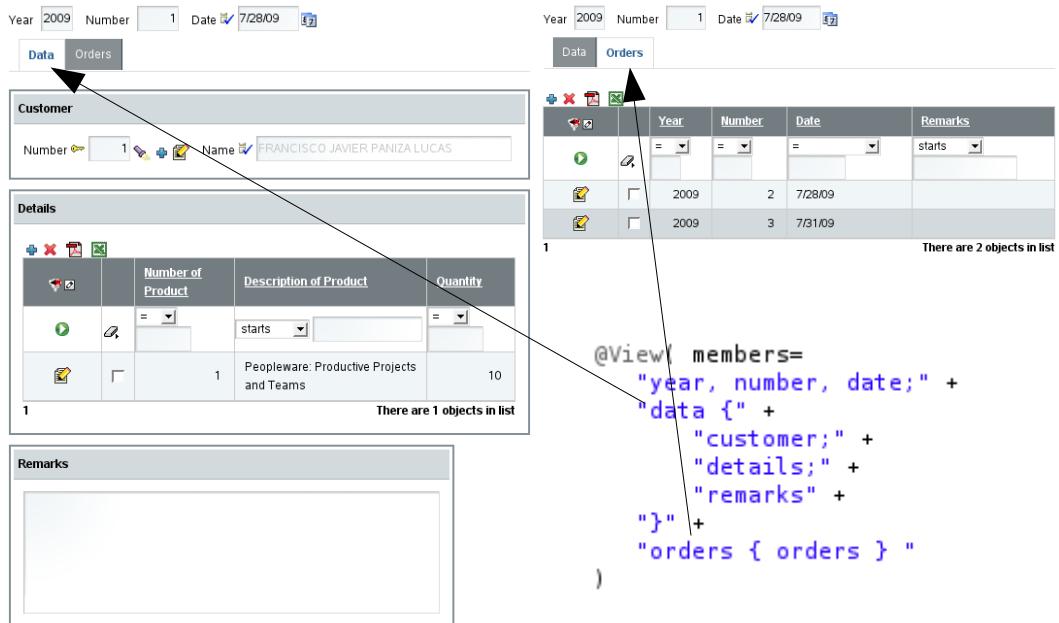


Figura 7.5 Interfaz de usuario para Invoice con orders

Nota que ponemos todos los miembros de la parte de `CommercialDocument` de `Invoice` en la cabecera y la primera pestaña (`data`), y la colección de pedidos en la otra pestaña. El listado 7.16 muestra la forma de definir esto sin herencia.

Listado 7.16 Vista para Invoice sin herencia de vistas

```
@View( members=
    "year, number, date;" +
    "data {" +
        "customer;" +
        "details;" +
        "remarks" +
    "}" +
    "orders { orders } "
)
public class Invoice extends CommercialDocument {
```

Puedes notar como todos los miembros, excepto la parte de `orders`, son comunes para todos los `CommercialDocument`. Por lo tanto, vamos a mover esta parte común a `CommercialDocument` y redefinir la vista usando herencia.

Quita el viejo `@View` de `CommercialDocument`, y escribe el que hay en el listado 7.17.

Listado 7.17 Vista para CommercialDocumment

```
@View(members=
    "year, number, date," + // Los miembros para la cabecera en una línea
    "data {" + // Una pestaña 'data' para los datos principales del documento
```

```

    "customer;" +
    "details;" +
    "remarks" +
}
abstract public class CommercialDocument extends Identifiable {

```

Esta vista indica como distribuir los datos comunes para todos los documentos comerciales. Ahora podemos redefinir la vista de Invoice a partir de esta. Mira el listado 7.18.

Listado 7.18 Definición de la vista de Invoice usando herencia de vistas

```

@View(extendsView="super.DEFAULT", // Extiende de la vista de CommercialDocument
      members="orders { orders }" // Añadimos orders dentro de una pestaña
)
public class Invoice extends CommercialDocument {

```

De esta forma declarar la vista para Invoice es más corto, es más, la distribución común para Order, Invoice y todos los demás posibles CommercialDocument está en un único sitio, por tanto, si añades una nueva propiedad a CommercialDocument solo has de tocar la vista de CommercialDocument.

7.3.3 Vista para Order usando herencia

Ahora que tienes una vista adecuada para CommercialDocument, declarar una vista para Order es lo más fácil del mundo. La figura 7.6 muestra la vista que



Figura 7.6 Interfaz de usuario para Order con una referencia a Invoice

111 Capítulo 7: Herencia

queremos. Una vista con toda la información del pedido, y su factura asociada en otra pestaña.

Para obtener este resultado, puedes definir la vista de Order extendiendo la vista por defecto de CommercialDocument, y añadiendo la referencia a factura, tal como muestra el listado 7.19.

Listado 7.19 Definición de vista de Order usando herencia de vistas

```
@View(extendsView="super.DEFAULT", // Extiende de la vista de CommercialDocument  
       members="invoice { invoice } " // Añadimos invoice dentro de una pestaña  
)  
public class Order extends CommercialDocument {
```

Con esto conseguimos toda la información de CommercialDocument más una pestaña con la factura.

7.3.4 Usar @ReferenceView y @CollectionView para refinar vistas

Queremos que cuando un pedido sea editado desde la interfaz de usuario de Invoice la vista a usar sea simple, sin cliente ni factura, porque estos datos son redundantes en este caso. Mira la figura 7.7.



Figura 7.7 Vista para editar Order desde Invoice tiene que ser más simple

Para obtener este resultado define una vista más simple en Order (listado 7.20), y referenciala desde la colección orders en Invoice (listado 7.21).

Listado 7.20 Vista NoCustomerNoInvoice en Order para ser usada desde Invoice

```
@Views({ // Para declarar más de una vista  
    @View( extendsView="super.DEFAULT", // La vista por defecto  
          members="invoice { invoice } "  
)  
    @View( name="NoCustomerNoInvoice", // Una vista llamada NoCustomerNoInvoice
```

```

    members=
    "year, number, date;" +
    "details;" +
    "remarks"
)
})
public class Order extends CommercialDocument {

```

Esta nueva vista definida en Order llamada NoCustomerNoInvoice puede ser referenciada desde Invoice para visualizar o editar elementos individuales de la colección orders usando @CollectionView. Míralo en el listado 7.21.

Listado 7.21 Usar la vista NoCustomerNoInvoice de Order desde Invoice

```

@OneToMany(mappedBy="invoice")
@CollectionView("NoCustomerNoInvoice") // Esta vista se usa para visualizar orders
private Collection<Order> orders;

```

Y tan solo con este código la colección orders usará una vista más apropiada de Invoice para editar elementos individuales (figura 7.7).

Además, queremos que desde la interfaz de usuario de Order la factura no muestre el cliente y los pedidos, porque son datos redundantes en este caso. Para conseguirlo, vamos a definir una vista más simple en Invoice (listado 7.22), y referenciarla desde la referencia invoice en Order (listado 7.23).

Listado 7.22 Nueva vista NoCustomerNoOrders en Invoice para usarse en Order

```

@Views({
    // Para declarar más de una vista
    @View(  extendsView="super.DEFAULT",   // La vista por defecto
            members="orders { orders } "
    ),
    @View(  name="NoCustomerNoOrders",      // Una vista llamada NoCustomerNoOrders
            members=
            "year, number, date;" +
            "details;" +
            "remarks"
    )
})
public class Invoice extends CommercialDocument {

```

Esta nueva vista definida en Invoice llamada NoCustomerNoOrders puede ser referenciada desde Order para visualizar la referencia Invoice usando @ReferenceView. Lo puedes ver en el listado 7.23.

Listado 7.23 Usar la vista NoCustomerNoOrders de Invoice desde Order

```

public class Order extends CommercialDocument {

    @ManyToOne
    @ReferenceView("NoCustomerNoOrders") // Esta vista se usa para visualizar invoice
    private Invoice invoice;

    ...
}

```

```
}
```

Ahora la referencia `invoice` será visualizada apropiadamente desde `Order` y así tendrás la interfaz de usuario deseada.

7.4 Herencia en las pruebas JUnit

`Invoice` ha sido refactorizada para usar herencia, y también hemos usado herencia para añadir una nueva entidad, `Order`. Además, esta entidad `Order` tiene relación con `Invoice`. Lo cual es una nueva funcionalidad, por ende has de probar todas estas nuevas características.

Dado que `Invoice` y `Order` tienen bastantes cosas en común (la parte de `CommercialDocument`) podemos refactorizar las pruebas para usar herencia, y así eludir el dañino “copiar y pegar” también en tu código de prueba.

7.4.1 Crear una prueba de módulo abstracta

Si examinas la prueba para crear una factura, en el método `testCreate()` de `InvoiceTest` (sección 6.6). Puedes notar que probar la creación de una factura es exactamente igual que probar la creación de un pedido. Porque, ambos tienen año, número, fecha, cliente, detalles y observaciones. Por tanto, aquí la herencia es una buena herramienta para la reutilización de código.

Vamos a renombrar `InvoiceTest` como `CommercialDocumentTest`, y entonces crearemos `InvoiceTest` y `OrderTest` a partir de él. Puedes ver el diagrama UML de esta idea en la figura 7.8.

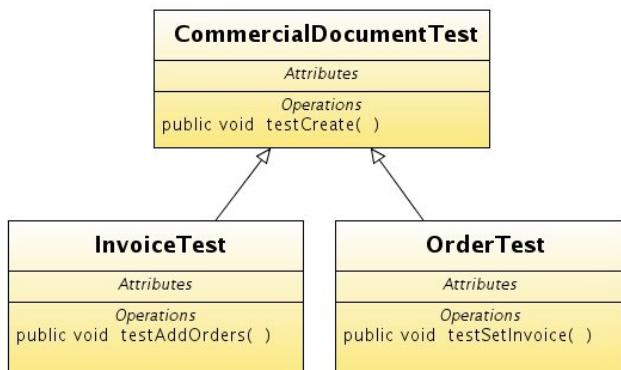


Figura 7.8 Jerarquía de `CommercialDocumentTest`

Primero renombra tu actual clase `InvoiceTest` a `CommercialDocumentTest`, y después haz los cambios indicados en el listado 7.24.

Listado 7.24 `CommercialDocumentTest` creado renombrando el viejo `InvoiceTest`

```
abstract public class CommercialDocumentTest // Añade abstract a la definición de clase
  extends ModuleTestBase {
```

```

private String number;

public CommercialDocumentTest(
    String testName,
    String moduleName) // moduleName añadido como argumento en el constructor
{
    super(testName, "Invoicing", moduleName); // Envía el moduleName
}

public void testCreate() throws Exception { ... } // Como el original

private void addDetails() throws Exception {
    // Añadir una línea de detalle
    assertCollectionRowCount("details", 0);
    execute("Collection.new",
        "viewObject=xava_view_section0_details"); // Cambia xava_view_details
                                                // por xava_view_section0_details
    // El resto del método como el original
    ...
}

private String getNumber() {
    if (number == null) {
        Query query = getManager().
            createQuery(
                "select max(i.number) from " +
                "CommercialDocument i " + // Invoice cambiada por CommercialDocument
                "i where i.year = :year");
        query.setParameter("year",
            Dates.getYear(new Date()));
        Integer lastNumber = (Integer)
            query.getSingleResult();
        if (lastNumber == null) lastNumber = 0;
        number = Integer.toString(lastNumber + 1);
    }
    return number;
}

private void remove() throws Exception { ... } // Como el original

private void verifyCreated() throws Exception { ... } // Como el original

private void save() throws Exception { ... } // Como el original

private void setOtherProperties()
    throws Exception { ... } // Como el original

private void chooseCustomer() throws Exception { ... } // Como el original

private void verifyDefaultValues()
    throws Exception { ... } // Como el original

private String getCurrentYear() { ... } // Como el original

private String getCurrentDate() { ... } // Como el original
}

```

Como ves en el listado 7.24 has tenido que hacer unos pocos cambios para

115 Capítulo 7: Herencia

adaptar CommercialDocumentTest. Primero, la has declarado abstracta, de esta forma esta clase no es ejecutada por Eclipse como una prueba JUnit, es solo válida como clase base para crear pruebas, pero ella misma no es una prueba.

Otro cambio importante nos lo encontramos en el constructor, donde ahora tienes moduleName en vez de “Invoice”, así puedes usar esta prueba para Order, Invoice o cualquier otro módulo que quieras. Los otros cambios son simples detalles: has de usar xava_view_section0_details para viewObject cuando llamas a las acciones de la colección, porque ahora la colección details está en la primera pestaña (section0), y has de cambiar “Invoice” por “ComercialDocument” en la consulta para obtener el siguiente número.

Ahora ya tienes una clase base lista para crear los módulos de prueba para Order e Invoice. Hagámoslo sin más dilación.

7.4.2 Prueba base abstracta para crear pruebas de módulo concretas

Crear tu primera versión para OrderTest e InvoiceTest es simplemente extender de CommercialDocumentTest. Nada más. Mira InvoiceTest en el listado 7.25.

Listado 7.25 InvoiceTest usando herencia

```
public class InvoiceTest extends CommercialDocumentTest {  
  
    public InvoiceTest(String testName) {  
        super(testName, "Invoice");  
    }  
}
```

Y OrderTest en el listado 7.26.

Listado 7.26 OrderTest usando herencia

```
public class OrderTest extends CommercialDocumentTest {  
  
    public OrderTest(String testName) {  
        super(testName, "Order");  
    }  
}
```

Ejecuta estas dos prueba y verás como testCreate(), heredado de CommercialDocumentTest, se ejecuta en ambos casos, contra su módulo correspondiente. Con esto estamos probando el comportamiento común para Order e Invoice. Probemos ahora la funcionalidad particular de cada uno.

7.4.3 Añadir nuevas pruebas a las pruebas de módulo extendidas

Hasta ahora hemos probado como crear una factura y un pedido. En esta sección probaremos como añadir pedidos a una factura en el módulo Invoice, y como establecer la factura a un pedido en el módulo Order.

Para probar como añadir un pedido a una factura añade el método `testAddOrdersMethod()` del listado 7.27 a `InvoiceTest`.

Listado 7.27 Probar añadir pedidos a una factura en `InvoiceTest`

```
public void testAddOrders() throws Exception {
    assertListNotEmpty(); // Esta prueba confía en que ya existen facturas
    execute("List.orderBy", "property=number"); // Para usar siempre el mismo pedido
    execute("Mode.detailAndFirst"); // Va al modo detalle editando la primera factura
    execute("Sections.change", "activeSection=1"); // Cambia a la pestaña 1
    assertCollectionRowCount("orders", 0); // Esta factura no tiene pedidos
    execute("Collection.add", "viewObject=xava_view_section1_orders"); // Pulsar el botón para añadir un nuevo pedido, esto te lleva
        // a la lista de pedidos
    execute("AddToCollection.add", "row=0"); // Escoge el primer pedido de la lista
    assertCollectionRowCount("orders", 1); // El pedido se ha añadido a la factura
    checkRowCollection("orders", 0); // Marca el pedido, para borrarlo
    execute("Collection.removeSelected", "viewObject=xava_view_section1_orders");
    assertCollectionRowCount("orders", 0); // El pedido ha sido borrado
}
```

En este caso asumimos que hay al menos una factura, y que la primera factura de la lista no tiene pedidos. Antes de ejecutar esta prueba, si no tienes facturas todavía, crea una sin pedidos, o si ya tienes facturas, asegúrate de que la primera no tiene pedidos.

Para probar como asignar una factura a un pedido añade el método `testSetInvoice()` del listado 7.28 a `OrderTest`.

Listado 7.28 Probar asignar una factura a un pedido en `OrderTest`

```
public void testSetInvoice() throws Exception {
    assertListNotEmpty(); // Esta prueba confía en que existen pedidos
    execute("Mode.detailAndFirst"); // Va a modo detalle editando la primera factura
    execute("Sections.change", "activeSection=1"); // Cambia a la pestaña 1
    assertEquals("invoice.number", ""); // Este pedido todavía no tiene
    assertEquals("invoice.year", ""); // una factura asignada
    execute("Reference.search", "keyProperty=invoice.year"); // Pulsar en el botón para buscar la factura, esto te
        // lleva a la lista de facturas
    String year = getValueInList(0, "year"); // Memoriza el año y el número de
    String number = getValueInList(0, "number"); // la primera factura de la lista
    execute("ReferenceSearch.choose", "row=0"); // Escoge la primera factura
    assertEquals("invoice.year", year); // Al volver al detalle del pedido verificamos
    assertEquals("invoice.number", number); // que la factura ha sido seleccionada
}
```

En este caso asumimos que hay al menos un pedido, y el primer pedido de la lista no tiene factura. Antes de ejecutar esta prueba, si no tienes pedidos, crea uno sin factura, o si ya tienes pedidos, asegúrate de que el primero no tiene factura.

Con esto ya tienes tus pruebas listas. Ejecútalas, y obtendrás el resultado de la figura 7.9. Fíjate que la prueba base `CommercialDocumentTest` no se muestra porque es abstracta. Y `testCreate()` de `CommercialDocumentTest` se ejecuta para `InvoiceTest` y `OrderTest`.

No solo has adaptado tu código de pruebas al nuevo código de Invoicing, sino que también has aprendido como usar herencia en el mismo código de pruebas.

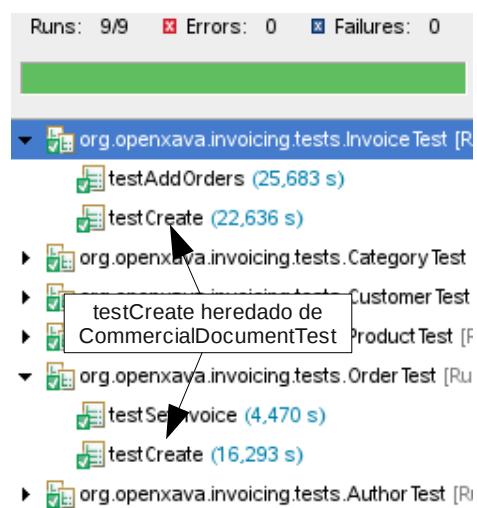


Figura 7.9 Pruebas de Invoicing

7.5 Resumen

Este capítulo te ha mostrado algunos ejemplos prácticos sobre como usar herencia con Java y JPA para simplificar tu código. Hemos usado la configuración por defecto de JPA para la herencia, aunque puedes refining el comportamiento de JPA para la herencia con anotaciones como `@Inheritance`, `@DiscriminatorColumn`, `@DiscriminatorValue`, etc. Para aprender más acerca de la herencia en JPA puedes leer la documentación de la sección 2.3.

OpenXava, al menos en la versión 3.1.x, solo soporta la estrategia de tabla única de JPA, este quiere decir que todos los datos de una jerarquía se almacenan en la misma tabla.

*Lógica
de negocio
básica*

capítulo 8

Has convertido tu modelo del dominio en una aplicación web plenamente funcional. Esta aplicación ya es bastante útil de por sí, aunque aún puedes hacerle muchas mejoras. Transformemos pues tu aplicación en algo más serio, y de paso, aprendamos algunas cosas interesantes sobre OpenXava.

Empezaremos por añadir algo de lógica de negocio a tus entidades para hacer de tu aplicación algo más que un simple gestor de base de datos.

8.1 Propiedades calculadas

Quizás la lógica de negocio más simple que puedes añadir a tu aplicación es una propiedad calculada. Las propiedades que has usado hasta ahora son persistentes, es decir, cada propiedad se almacena en una columna de una tabla de la base de datos. Una propiedad calculada es una propiedad que no almacena su valor en la base de datos, sino que se calcula cada vez que se accede a la propiedad. Observa la diferencia entre una propiedad persistente y una calculada en el listado 8.1.

Listado 8.1 Diferencia entre una propiedad persistente y una calculada

```
// Propiedad persistente
private int quantity; // Tiene un campo, por tanto es persistente
public int getQuantity() { // Un getter para devolver el valor del campo
    return quantity;
}
public void setQuantity(int quantity) { // Cambia el valor del campo
    this.quantity = quantity;
}

// Propiedad calculada
public int getAmount() { // No tiene campo ni setter, sólo un getter
    return quantity * price; // con un cálculo
}
```

Las propiedades calculadas son reconocidas automáticamente por OpenXava. Puedes usarlas en vistas, listas tabulares o cualquier otra parte de tu código.

Vamos a usar propiedades calculadas para añadir el elemento “económico” a nuestra aplicación Invoicing. Porque, tenemos líneas de detalle, productos, cantidades. Pero, ¿qué pasa con el dinero?

8.1.1 Propiedad calculada simple

El primer paso será añadir una propiedad de importe a Detail. Lo que queremos es que cuando el usuario elija un producto y teclea la cantidad el importe de la línea sea recalculado y mostrado al usuario, tal como muestra la figura 8.1.

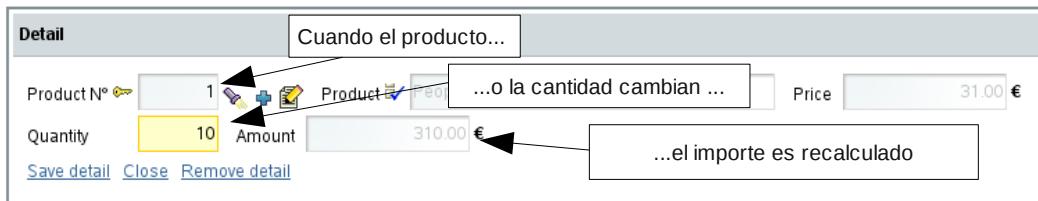


Figura 8.1 Comportamiento de la propiedad calculada amount en Detail

Añadir esta funcionalidad a tu actual código es prácticamente añadir una propiedad calculada a Detail. Simplemente añade el código del listado 8.2 al código de Detail.

Listado 8.2 Propiedad calculada amount en la clase Detail

```
@Stereotype("MONEY")
@Depends("product.number, quantity") // Cuando el usuario cambie producto o cantidad
public BigDecimal getAmount() {           // esta propiedad se recalculará y se redibujará
    return new BigDecimal(quantity).multiply(product.getPrice());
}
```

Es tan solo poner el cálculo en `getAmount()` y usar `@Depends` para indicar a OpenXava que la propiedad `amount` depende de `product.number` y `quantity`, así cada vez que el usuario cambia alguno de estos valores la propiedad se recalculará.

Ahora has de añadir esta nueva propiedad a la vista de Detail (listado 8.3).

Listado 8.3 Añadir la propiedad amount a la vista de Detail

```
@View(members="product; quantity, amount") // amount añadida
public class Detail extends Identifiable {
```

El único detalle que nos queda es modificar la vista Simple de Product para mostrar el precio. Puedes verlo en el listado 8.4.

Listado 8.4 Añadir la propiedad price a la vista de Product

```
@View(name="Simple", members="number, description, price") // price añadido
public class Product {
```

Nada más. Tan solo necesitas añadir el *getter* y modificar las vistas. Ahora puedes probar los módulos Invoice y Order para ver la propiedad `amount` en acción.

8.1.2 Usar `@DefaultValueCalculator`

La forma en que calculamos el importe de la línea de detalle no es la mejor. Tiene, al menos, dos inconvenientes. El primero es que el usuario puede querer tener la posibilidad de cambiar el precio unitario. Y segundo, si el precio de un producto cambia los importes de todas las facturas cambian también, y esto no es

bueno.

Para evitar estos inconvenientes lo mejor es almacenar el precio de cada producto en cada línea de detalle. Añadamos pues una propiedad persistente `pricePerUnit` a la entidad `Detail`, y calculemos su valor desde `price` de `Product` usando un `@DefaultValueCalculator`. De tal forma que consigamos el efecto que puedes ver en la figura 8.2.



Figura 8.2 Entidad Detail con `pricePerUnit` como propiedad persistente

El primer paso es obviamente añadir la propiedad `pricePerUnit`. Añade el código del listado 8.5 a tu entidad `Detail`.

Listado 8.5 La propiedad `pricePerUnit` de la entidad `Detail`

```

@DefaultValueCalculator(
    value=PricePerUnitCalculator.class, // Esta clase calcula el valor inicial
    properties=@PropertyValue(
        name="productNumber", // La propiedad productNumber del calculador...
        from="product.number") // ... se llena con el valor de product.number de la entidad
)
@stereotype("MONEY")
private BigDecimal pricePerUnit; // Una propiedad persistente convencional...

public BigDecimal getPricePerUnit() { // ... con sus getter y setter
    return pricePerUnit==null?
        BigDecimal.ZERO:pricePerUnit; // Así nunca devuelve nulo
}

public void setPricePerUnit(BigDecimal pricePerUnit) {
    this.pricePerUnit = pricePerUnit;
}

```

`PricePerUnitCalculator` contiene la lógica para calcular el valor inicial. Simplemente lee el precio del producto. Observa el código de este calculador en el listado 8.6.

Listado 8.6 `PricePerUnitCalculator` calcula el valor por defecto para `pricePerUnit`

```

package org.openxava.invoicing.calculators; // En el paquete calculators

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

import static org.openxava.jpa.XPersistence.*; // Para usar getManager()

public class PricePerUnitCalculator implements ICalculator {

    private int productNumber; // En calculate() contendrá el número de producto

```

```

public Object calculate() throws Exception {
    Product product = getManager() // getManager() de XPersistence
        .find(Product.class, productNumber); // Busca el producto
    return product.getPrice(); - Returns its price
}

public void setProductNumber(int productNumber) {
    this.productNumber = productNumber;
}

public int getProductNumber() {
    return productNumber;
}

}

```

De esta forma cuando el usuario escoge un producto el campo de precio unitario se rellena con el precio del producto, pero, dado que es una propiedad persistente, el usuario puede cambiar este valor. Y si en el futuro el precio del producto cambiara este precio unitario de la línea de detalle no cambiaría.

Esto implica que has de adaptar la propiedad calculada amount (listado 8.7).

Listado 8.7 La propiedad amount de Detail adaptada para usar pricePerUnit

```

@Stereotype("MONEY")
@Depends("pricePerUnit, quantity") // pricePerUnit en vez de product.number
public BigDecimal getAmount() {
    return new BigDecimal(quantity)
        .multiply(getPricePerUnit()); // getPricePerUnit() en vez de product.getPrice()
}

```

Ahora getAmount() usa productPerUnit como fuente en lugar de product.price.

Finalmente, hemos de modificar la vista de Detail para que muestre la nueva propiedad (listado 8.8).

Listado 8.8 La vista de Product ahora incluye pricePerUnit

```

@View(members="product; quantity, pricePerUnit, amount") // pricePerUnit añadida
public class Detail extends Identifiable {

```

Y la vista Simple de Product para que no muestre el precio (listado 8.9).

Listado 8.9 Vista Simple de Product ahora no incluye price

```

@View(name="Simple", members="number, description") // price quitado
public class Product {

```

Es decir, hemos arreglado la interfaz de usuario para que muestre la propiedad persistente (y modificable por el usuario) pricePerUnit, en vez del precio del producto.

También sería bonito ver estas nuevas propiedades en la colección. Para hacerlo, edita la entidad `CommercialDocument` y modifica la lista de propiedades a mostrar, como muestra el listado 8.10.

Listado 8.10 Añadir `pricePerUnit` y `amount` a `details` en `ComercialDocument`

```
@ListProperties(
    "product.number, product.description, " +
    "quantity, pricePerUnit, amount") // pricePerUnit y amount añadidos
private Collection<Detail> details = new ArrayList<Detail>();
```

Actualiza el esquema de la base de datos, prueba los módulos `Order` e `Invoice` y podrás observar el nuevo comportamiento al añadir líneas de detalle.

8.1.3 Propiedades calculadas dependientes de una colección

También queremos añadir importes a `Order` e `Invoice`. Tener IVA, importe base e importe total es indispensable. Para hacerlo solo necesitas añadir unas pocas propiedades calculadas. La figura 8.3 muestra la interfaz de usuario para estas propiedades.

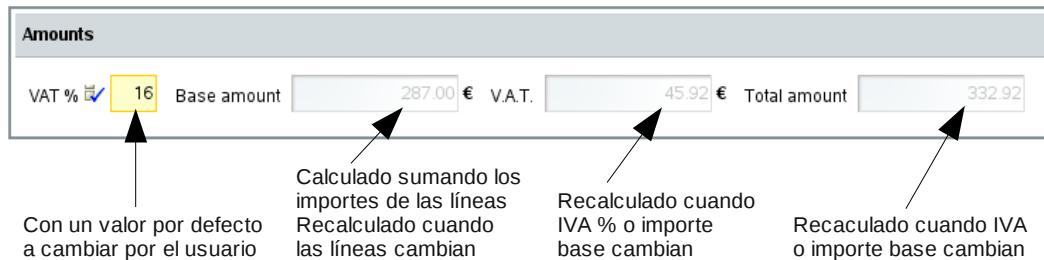


Figura 8.3 Propiedades calculadas en `CommercialDocument`

Empecemos con `baseAmount`. El listado 8.11 muestra su implementación.

Listado 8.11 Propiedad calculada `baseAmount` en `CommercialDocument`

```
@Stereotype("MONEY")
public BigDecimal getBaseAmount() {
    BigDecimal result = new BigDecimal("0.00");
    for (Detail detail: getDetails()) { // Iteramos por todas la líneas de detalle
        result = result.add(detail.getAmount()); // Acumulamos el importe
    }
    return result;
}
```

La implementación es simple, se trata de sumar los importes de todas las líneas.

La siguiente propiedad a añadir es `vatPercentage` que se usará para calcular el IVA. El listado 8.12 te muestra el código para esta propiedad.

125 Capítulo 8: Lógica de negocio básica

Listado 8.12 La propiedad persistente vatPercentage en CommercialDocument

```
@Digits(integerDigits=2, fractionalDigits=0) // Para indicar su tamaño  
@Required  
private BigDecimal vatPercentage;  
  
public BigDecimal getVatPercentage() {  
    return vatPercentage==null?  
        BigDecimal.ZERO:vatPercentage; // Así nunca devuelve nulo  
}  
  
public void setVatPercentage(BigDecimal vatPercentage) {  
    this.vatPercentage = vatPercentage;  
}
```

Puedes ver como vatPercentage es una propiedad persistente convencional. En este caso usamos @Digits (una anotación del entorno de validación Hibernate Validator) como una alternativa a @Column para especificar el tamaño.

Continuaremos añadiendo la propiedad vat. La puedes ver en el listado 8.13.

Listado 8.13 La propiedad calculada vat en CommercialDocument

```
@Stereotype("MONEY")  
@Depends("vatPercentage") // Cuando vatPercentage cambia vat se recalcula y revisualiza  
public BigDecimal getVat() {  
    return getBaseAmount() // baseAmount * vatPercentage / 100  
        .multiply(getVatPercentage())  
        .divide(new BigDecimal("100"));  
}
```

Es un cálculo simple. Usamos @Depends para recalcular y revisualizar vat cada vez que el usuario modifique vatPercentage.

Solo nos queda totalAmount por añadir. Puedes ver su código en el listado 8.14.

Listado 8.14 La propiedad calculada totalAmount en CommercialDocument

```
@Stereotype("MONEY")  
@Depends("baseAmount, vat") // Cuando baseAmount o vat cambian totalAmount se  
public BigDecimal getTotalAmount() { // recalcula y revisualiza  
    return getBaseAmount().add(getVat()); // baseAmount + vat  
}
```

Una vez más un cálculo simple, y una vez más usamos @Depends.

Ahora que ya has escrito las propiedades para los importes de CommercialDocument, tienes que modificar la vista para que estas nuevas propiedades se muestren (listado 8.15).

Listado 8.15 Añadir propiedades de importes a la vista de CommercialDocument

```
@View(members=  
    "year, number, date;" +  
    "data {" +
```

```

    "customer;" +
    "details;" +
    "amounts [ " + // Los corchetes indican un grupo, visualizado dentro de un marco
      " vatPercentage, baseAmount, vat, totalAmount" +
    "];" +
    "remarks" +
  "}";
}
abstract public class CommercialDocument extends Identifiable {

```

Añadimos vatPercentage, baseAmount, vat, totalAmount entre details y remarks, pero entre unos corchetes con un nombre (amounts). Esto significa que estas propiedades se mostrarán dentro de un marco llamado “amount”. Esto es un grupo.

Ahora, puedes actualizar el esquema de la base de datos (por causa de vatPercentage) y probar tu aplicación. Debería funcionar casi como en la susodicha figura 8.3. “Casi” porque vatPercentage todavía no tiene un valor por defecto. Lo añadiremos en la siguiente sección.

8.1.4 Valor por defecto desde un archivo de propiedades

Es conveniente para el usuario tener el campo vatPercentage lleno por defecto con un valor adecuado. Puedes usar un calculador (@DefaultValueCalculator) que devuelva un valor fijo, en este caso cambiar el valor por defecto implica cambiar el código fuente. O puedes leer el valor por defecto de una base de datos (usando JPA desde tu calculador), en este caso cambiar el valor por defecto implica actualizar la base de datos.

Otra opción es tener estos valores de configuración en un archivo de propiedades, un archivo plano con pares clave=valor. En este caso cambiar el valor por defecto de vatPercentage es tan simple como editar un archivo plano con un editor de texto.

Implementemos la opción del archivo de propiedades. Crea un archivo llamado *invoicing.properties* en la carpeta *Invoicing/properties* con el contenido del listado 8.16.

Listado 8.16 El contenido del archivo *invoicing.properties*

```
defaultVatPercentage=18
```

Aunque puedes usar la clase *java.util.Properties* de Java para leer este archivo preferimos usar una clase propia para leer estas propiedades. Vamos a llamar a esta clase *InvoicingPreferences* y la pondremos en un nuevo paquete llamado *org.openxava.invoicing.util*. Tienes el código en el listado 8.17.

Listado 8.17 InvoicingPreferences para leer el archivo invoicing.properties

```

package org.openxava.invoicing.util; // En el paquete util

import java.io.*;
import java.math.*;
import java.util.*;

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

public class InvoicingPreferences {

    private final static String
        FILE_PROPERTIES="invoicing.properties";
    private static Log log =
        LogFactory.getLog(InvoicingPreferences.class);
    private static Properties properties; // Almacenamos las propiedades aquí

    private static Properties getProperties() {
        if (properties == null) { // Usamos inicialización vaga
            PropertiesReader reader = // PropertiesReader es una clase de OpenXava
                new PropertiesReader(
                    InvoicingPreferences.class,
                    FILE_PROPERTIES);
            try {
                properties = reader.get();
            }
            catch (IOException ex) {
                log.error(
                    XavaResources.getString( // Para leer un mensaje i18n
                        "properties_file_error",
                        FILE_PROPERTIES),
                    ex);
                properties = new Properties();
            }
        }
        return properties;
    }

    public static BigDecimal getDefaultVatPercentage() { // El único método público
        return new BigDecimal(
            getProperties().getProperty("defaultVatPercentage"));
    }

}

```

Como puedes ver InvoicingPreferences es una clase con un método estático, `getDefaultVatPercentage()`. La ventaja de usar esta clase en lugar de leer directamente del archivo de propiedades es que si cambias la forma en que se obtienen las preferencias, por ejemplo leyendo de una base de datos o de un directorio LDAP, solo has de cambiar esta clase en toda tu aplicación.

Puedes usar esta clase desde el calculador por defecto para la propiedad `vatPercentage`. El listado 8.18 contiene el código del calculador.

Listado 8.18 Calculador para el valor por defecto del porcentaje de IVA

```
package org.openxava.invoicing.calculators; // En el paquete calculators

import org.openxava.calculators.*; // Para usar ICalculator
import org.openxava.invoicing.util.*; // Para usar InvoicingPreferences

public class VatPercentageCalculator implements ICalculator {

    public Object calculate() throws Exception {
        return InvoicingPreferences.getDefaultVatPercentage();
    }

}
```

Como ves, simplemente devuelve `defaultValuePercentage` de `InvoicingPreferences`. Ahora, ya puedes usar este calculador en la definición de la propiedad `vatPercentage` en `CommercialDocument`. Mira en el listado 8.19.

Listado 8.19 Calculador por defecto para vatPercentage de CommercialDocument

```
@DefaultValueCalculator(VatPercentageCalculator.class)
private BigDecimal vatPercentage;
```

Con este código cuando el usuario pulsa para crear una nueva factura, el campo `vatPercentage` se llenará con 16, o cualquier otro valor que hayas puesto en `invoicing.properties`.

8.2 Métodos de retrollamada JPA

Otra forma práctica de añadir lógica de negocio a tu modelo es mediante los métodos de retrollamada JPA. Un método de retrollamada se llama en un momento específico del ciclo de vida de la entidad como objeto persistente. Es decir, puedes especificar cierta lógica a ejecutar al grabar, leer, borrar o modificar una entidad.

En esta sección veremos algunas aplicaciones prácticas de los métodos de retrollamada JPA.

8.2.1 Cálculo de valor por defecto multiusuario

Hasta ahora estamos calculando el número para `Invoice` y `Order` usando `@DefaultValueCalculator`. Éste calcula el valor por defecto en el momento que el usuario pulsa para crear una nueva `Invoice` o `Order`. Por tanto, si varios usuarios pulsan en el botón “nuevo” al mismo tiempo todos ellos obtendrán el mismo número. Esto no es apto para aplicaciones multiusuario. La forma correcta de generar un número único es generándolo justo en el momento de grabar.

129 Capítulo 8: Lógica de negocio básica

Vamos a implementar la generación del número usando métodos de retrollamada JPA. JPA permite marcar cualquier método de tu clase para ser ejecutado en cualquier momento de su ciclo de vida. Indicaremos que justo antes de grabar un `CommercialDocument` calcule su número. De paso mejoraremos el cálculo para tener una numeración diferente para `Order` e `Invoice`.

Edita la entidad `CommercialDocument` y añade el método `calculateNumber()` que tienes en el listado 8.20.

Listado 8.20 Método `@PrePersist calculateNumber()` de `CommercialDocument`

```
@PrePersist // Ejecutado justo antes de grabar el objeto por primera vez
public void calculateNumber() throws Exception {
    Query query = XPersistence.getManager()
        .createQuery("select max(i.number) from " +
            getClass().getSimpleName() + // De esta forma es válido para Invoice y Order
            " i where i.year = :year");
    query.setParameter("year", year);
    Integer lastNumber = (Integer) query.getSingleResult();
    this.number = lastNumber == null?1:lastNumber + 1;
}
```

El código del listado 8.20 es el mismo que el de `NextNumberForYearCalculator` pero usando `getClass().getSimpleName()` en lugar de “`CommercialDocument`”. El método `getSimpleName()` devuelve el nombre de la clase sin paquete, es decir, precisamente el nombre de entidad. Será “`Order`” para `Order` e “`Invoice`” para `Invoice`. Así podemos obtener una numeración diferente para `Order` e `Invoice`.

La especificación JPA establece que no puedes usar el API JPA dentro de un método de retrollamada. Por tanto, el método de arriba no es legal desde un punto de vista estricto. Pero, Hibernate (la implementación de JPA que OpenXava usa por defecto) te permite usar en `@PrePersist`. Y dado que usar JPA es la forma más fácil de hacer este cálculo, nosotros lo usamos.

Ahora borra la clase `NextNumberForYearCalculator` de tu proyecto, y modifica la propiedad `number` de `CommercialDocument` para que no la use (listado 8.21).

Listado 8.21 `number` de `CommercialDocument` sin `@DefaultValueCalculator`

```
@Column(length=6)
@DefaultValueCalculator(value=NextNumberForYearCalculator.class, // Quita esto
    properties=@PropertyValue(name="year"))
+
@ReadOnly // El usuario no puede modificar el valor
private int number;
```

Nota que, además de quitar `@DefaultValueCalculator`, hemos añadido la anotación `@ReadOnly`. Esto significa que el usuario no puede introducir ni modificar este número. Esta es la forma correcta de hacerlo ahora dado que el

número es generado al grabar el objeto, por lo que el valor que tecleará el usuario sería sobrescrito siempre.

Prueba ahora el módulo de Invoice u Order, verás como el número está vacío y no es editable, y cuando añades la primera línea de detalle, lo cual graba el documento contenedor, el número de documento se calcula y se actualiza en la interfaz de usuario.

8.2.2 Sincronizar propiedades persistentes y calculadas

La forma en que calculamos el IVA, el importe base y el importe total es natural y práctica. Usamos propiedades calculadas que calculan, usando Java puro, los valores cada vez que son llamadas. Esto está bien porque cada vez que el usuario añade, modifica o borra una línea de detalle en la interfaz de usuario, el IVA, el importe base y el importe total se recalculan con datos frescos instantáneamente.

Pero, las propiedades calculadas tienen algunos inconvenientes. Por ejemplo, si quieres hacer un proceso masivo o un informe de todas las facturas cuyo importe total esté entre ciertos rangos. En estos casos, si tienes una base de datos demasiado grande el proceso puede ser lentísimo, porque has de instanciar todas las facturas para calcular su importe total. Una solución para este problema es tener una propiedad persistente, por tanto una columna en la base de datos para el importe de la factura o pedido; así el rendimiento es bastante mayor.

En nuestro caso mantendremos nuestras actuales propiedades calculadas, pero vamos a añadir una nueva, llamada amount, que contendrá el mismo valor que totalAmount, pero amount será persistente con su correspondiente columna en la base de datos. Lo complicado aquí es mantener sincronizado el valor de la propiedad amount. Vamos a usar métodos de retrollamada JPA para conseguirlo.

El primer paso es añadir la propiedad amount a CommercialDocument. Nada más fácil, puedes verlo en el listado 8.22.

Listado 8.22 Propiedad persistente amount en CommercialDocument

```
@Stereotype("MONEY")
private BigDecimal amount;

public BigDecimal getAmount() {
    return amount;
}

public void setAmount(BigDecimal amount) {
    this.amount = amount;
}
```

Modifiquemos la entidad Detail para que cada vez que un detalle sea

131 Capítulo 8: Lógica de negocio básica

añadido, quitado o modificado la propiedad `amount` de su `CommercialDocument` contenedor sea recalculada correctamente. Añade los tres métodos de retrollamada del listado 8.23 a tu entidad `Detail`.

Listado 8.23 Métodos de retrollamada en Detail recalculan el importe del padre

```
@PrePersist // Al grabar el detalle por primera vez
private void onPersist() {
    getParent().getDetails().add(this); // Para tener la colección sincronizada
    getParent().recalculateAmount();
}

@PreUpdate // Cada vez que el detalle se modifica
private void onUpdate() {
    getParent().recalculateAmount();
}

@PreRemove // Al borrar el detalle
private void onRemove() {
    getParent().getDetails().remove(this); // Para tener la colección sincronizada
    getParent().recalculateAmount();
}
```

Llamamos al método `recalculateAmount()` del `CommercialDocument` padre cada vez que se añade, quita o modifica un detalle. Veamos este método en el listado 8.24.

Listado 8.24 recalculateAmount() sincroniza totalAmount y amount

```
public void recalculateAmount() {
    setAmount(getTotalAmount());
}
```

Como puedes ver movemos la propiedad calculada, `totalAmount`, a la persistente, `amount`.

Prueba el módulo `Invoice` u `Order` con este código y verás que cuando una línea de detalle es añadida, borrada o modificada la columna en la base de datos para `amount` se actualiza correctamente. Pero, si tratas de borrar la `Invoice` u `Order` obtendrás una `java.util.ConcurrentModificationException`. Esto es porque estamos usando *cascade ALL*, y cuando un `CommercialDocument` se borra sus detalles son borrados automáticamente, y en este caso el método de retrollamada `@PreRemove (onRemove())` falla. Hemos de hacer un pequeño ajuste para evitar este desgradable efecto. El ajuste es simple: no ejecutar el método `onRemove()` de `Detail` cuando el `CommercialDocument` contenedor está siendo borrado. Para programar esto añade el código del listado 8.25 a tu clase `CommercialDocument`.

Listado 8.25 Código en CommercialDocument para indicar que se está borrando

```
@Transient // No se almacena en la tabla de la base de datos
private boolean removing = false; // Indica si JPA está borrando el documento ahora
```

```

boolean isRemoving() { // Acceso paquete, no es accesible desde fuera
    return removing;
}

@PreRemove // Cuando el documento va a ser borrado marcamos removing como true
private void markRemoving() {
    this.removing = true;
}

@PostRemove // Cuando el documento ha sido borrado marcamos removing como false
private void unmarkRemoving() {
    this.removing = false;
}

```

Aquí ves como hemos añadido una propiedad transitoria `removing`, que es `true` solo cuando el `CommercialDocument` está siendo borrado. Podemos usar esta propiedad desde `Detail` de la forma que ves en el listado 8.26.

Listado 8.26 Método `onRemoved()` de `Detail` refinado

```

@PreRemove
private void onRemove() {
    if (getParent().isRemoving()) return; // Añadimos esta línea para evitar excepciones
    getParent().getDetails().remove(this);
    getParent().recalculateAmount();
}

```

Es decir, si el contenedor está siendo borrado, no ejecutamos la lógica de `onRemove()`. Después de este pequeño ajuste, `Invoice` y `Order` tienen su propiedad `amount` siempre sincronizada, y lista para ser usada en un proceso masivo.

8.3 Lógica desde la base de datos (@Formula)

Idealmente escribirás toda tu lógica de negocio en Java, dentro de tus entidades. Sin embargo, hay ocasiones que esto no es lo más conveniente. Imagina que tienes una propiedad calculada en `CommercialDocument`, digamos `estimatedProfit`, como la del listado 8.27.

Listado 8.27 `estimatedProfit` como propiedad calculada en `CommercialDocument`

```

@stereotype("MONEY")
public BigDecimal getEstimatedProfit() {
    return getAmount().multiply(new BigDecimal("0.10"));
}

```

Si necesitas realizar un proceso con todas las facturas con un `estimatedProfit` mayor de 1000, has de escribir algo parecido al listado 8.28.

Listado 8.28 Seleccionar objetos según una propiedad calculada

```

Query query = getManager()
    .createQuery("from Invoice"); // Sin condición en la consulta
for (Object o: query.getResultList()) { // Itera por todos los objetos
    Invoice i = (Invoice) o;
    if (i.getEstimatedProfit() // Pregunta a cada objeto
        .compareTo(new BigDecimal("1000")) > 0) {
        i.doSomething();
    }
}

```

No puedes usar una condición en la consulta para discriminar por `estimatedProfit`, porque `estimatedProfit` no está en la base de datos, solo está en el objeto Java, por tanto tienes que instanciar cada objeto para preguntar por su `estimatedProfit`. A veces esto es una buena opción, pero si tienes una cantidad inmensa de facturas, y solo unas cuantas tienen el `estimatedProfit` mayor de 1000, entonces el proceso será muy ineficiente. ¿Qué alternativa tenemos?

Nuestra alternativa es usar la anotación `@Formula`. `@Formula` es una extensión de Hibernate al JPA estándar, que te permite mapear tu propiedad contra un estamento SQL. Puedes definir `estimatedProfit` con `@Formula` como muestra el listado 8.29.

Listado 8.29 `estimatedProfit` con `@Formula` en `CommercialDocument`

```

@org.hibernate.annotations.Formula("AMOUNT * 0.10") // El cálculo usando SQL
@stereotype("MONEY")
private BigDecimal estimatedProfit; // Un campo, como con las propiedades persistentes

public BigDecimal getEstimatedProfit() { // Sólo el getter es necesario
    return estimatedProfit;
}

```

Esto indica que cuando un `CommercialDocument` se lea de la base de datos, el campo `estimatedProfit` se llenará con el cálculo de `@Formula`, un cálculo que por cierto hace la base de datos. Lo más útil de las propiedades `@Formula` es que puedes usarlas en las condiciones, por tanto puedes reescribir el anterior proceso como muestra el listado 8.30.

Listado 8.30 Seleccionar objetos según una propiedad `@Formula`

```

Query query = getManager()
    .createQuery("from Invoice i where " +
        "i.estimatedProfit > :estimatedProfit"); // Podemos usar una condición
query.setParameter("estimatedProfit", new BigDecimal(1000));
for (Object o: query.getResultList()) { // Iteramos solo por los objetos seleccionados
    Invoice i = (Invoice) o;
    i.doSomething();
}

```

De esta forma pones el peso del cálculo de `estimatedProfit` y la selección

de los registros en el servidor de base de datos, y no el servidor Java.

Este hecho también tiene efecto en modo lista, porque el usuario no puede filtrar ni ordenar por propiedades calculadas, pero sí por propiedades con @Formula (figura 8.4).

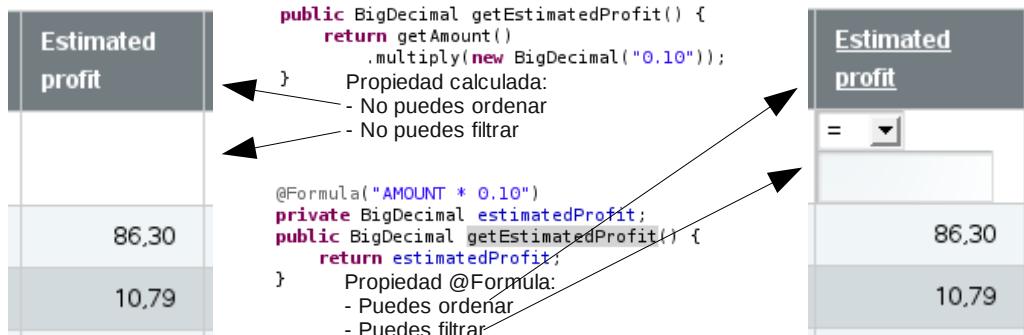


Figura 8.4 Modo lista para propiedades calculadas y propiedades @Formula

@Formula es una buena opción para mejorar el rendimiento en algunos casos. De todas formas, generalmente es mejor usar propiedades calculadas y escribir así tu lógica en Java. La ventaja de las propiedades calculadas sobre @Formula es que tu código no es dependiente de la base de datos. Además, con las propiedades calculadas puedes reejecutar el cálculo sin tener que leer el objeto de la base de datos, por tanto puedes usar @Depends.

8.4 Pruebas JUnit

Antes de ir al siguiente capítulo, vamos a escribir el código JUnit para éste. Recuerda, el código no está terminado si no tiene pruebas JUnit. Puedes escribir las pruebas antes, durante o después del código principal. Pero siempre has de escribirlas.

El código de prueba mostrado aquí no es solo para darte un buen ejemplo, sino también para enseñarte maneras de probar diferentes casos en tu aplicación OpenXava.

8.4.1 Modificar la prueba existente

Crear una nueva prueba para cada nuevo caso parece una buena idea desde un punto de vista estructural, pero en la mayoría de los casos no es práctico, porque de esa forma tu código de prueba crecerá muy rápido, y con el tiempo, ejecutar todas las pruebas supondrá muchísimo tiempo.

El enfoque más pragmático es modificar el código de prueba existente para cubrir todos los nuevos casos que hemos desarrollado. Hagámoslo de esta forma.

135 Capítulo 8: Lógica de negocio básica

En nuestro caso, todo el código de este capítulo aplica a `CommercialDocument`, por tanto vamos a modificar el método `testCreate()` de `CommercialDocumentTest` para ajustarlo a la nueva funcionalidad. Dejamos el método `testCreate()` tal como muestra el listado 8.31.

Listado 8.31 El método `testCreate()` de `CommercialDocumentTest` modificado

```
public void testCreate() throws Exception {
    calculateNumber(); // Añadido para calcular primero el siguiente número de documento
    verifyDefaultValues();
    chooseCustomer();
    addDetails();
    setOtherProperties();
    save();
    verifyAmountAndEstimatedProfit(); // Prueba el método de retrollamada y @Formula
    verifyCreated();
    remove();
}
```

Como ves, añadimos una nueva línea al principio para calcular el siguiente número de documento, y una llamada al nuevo método `verifyAmountAndEstimatedProfit()`.

Ahora nos conviene más calcular el siguiente número de documento al principio para usarlo en el resto de la prueba. Para hacer esto, cambia el viejo método `getNumber()` por los dos métodos mostrados en el listado 8.32.

Listado 8.32 Un método para calcular el siguiente número y otro para leerlo

```
private void calculateNumber() {
    Query query = getManager().
        createQuery(
            "select max(i.number) from " +
            model + // Cambiamos CommercialDocument por una variable
            " i where i.year = :year");
    query.setParameter("year", Dates.getYear(new Date()));
    Integer lastNumber = (Integer) query.getSingleResult();
    if (lastNumber == null) lastNumber = 0;
    number = Integer.toString(lastNumber + 1);
}

private String getNumber() {
    return number;
}
```

Anteriormente, teníamos solo `getNumber()` que calculaba y devolvía el número, ahora tenemos un método para calcular (`calculateNumber()`), y otro para devolver el resultado (`getNumber()`). Puedes notar que la lógica del cálculo tiene un pequeño cambio, en vez de usar “`CommercialDocument`” como fuente de la consulta usamos `model`, una variable. Esto es así porque ahora la numeración para facturas y pedidos está separada. Llenamos esta variable, un campo de la clase de prueba, en el constructor, tal como muestra el listado 8.33.

Listado 8.33 El constructor de la prueba almacena el nombre del modelo

```
private String model; // Nombre del modelo para la condición. Puede ser "Invoice" u "Order"

public CommercialDocumentTest(String testName, String moduleName) {
    super(testName, "Invoicing", moduleName);
    this.model = moduleName; // El nombre del módulo coincide con el del modelo
}
```

En este caso el nombre de módulo, “Invoice” u “Order”, coincide con el nombre de modelo, “Invoice” u “Order”, así que la forma más fácil de obtener el nombre de modelo es desde el nombre de módulo.

Veamos el código que prueba la nueva funcionalidad.

8.4.2 Verificar valores por defecto y propiedades calculadas

En este capítulo hemos hecho algunas modificaciones en los valores por defecto. Primero, el valor por defecto para `number` ya no se calcula mediante un `@DefaultValueCalculator` en su lugar usamos un método de retrollamada JPA. Segundo, tenemos una nueva propiedad, `vatPercentage`, cuyo valor inicial se calcula leyendo de un archivo de propiedades. Para probar estos casos hemos de modificar el método `verifyDefaultValues()` como ves en el listado 8.34.

Listado 8.34 El método verifyDefaultValues() adaptado

```
private void verifyDefaultValues() throws Exception {
    execute("CRUD.new");
    assertEquals("year", getCurrentYear());
    assertEquals("number", getNumber()); // Ahora el número no tiene valor inicial
    assertEquals("number", ""); // al crear un documento nuevo (sección 8.2.1)
    assertEquals("date", getCurrentDate());
    assertEquals("vatPercentage", "18"); // Valor de archivo de propiedades (sección 8.1.4)
}
```

Comprobamos el cálculo del valor por defecto de `vatPercentage` y verificamos que `number` no tiene valor inicial, porque ahora `number` no se calcula hasta el momento de grabar el documento (sección 8.2.1). Cuando el documento (factura o pedido) se grabe verificaremos que `number` se calcula. El documento se graba justo cuando la primera línea de detalle se añade. También cuando la línea se añade podemos verificar el cálculo de `amount` de `Detail` (la propiedad calculada simple, sección 8.1.1), el valor por defecto para `pricePerUnit` (`@DefaultValueCalculator`, sección 8.1.2) y las propiedades de importes del documento (propiedades calculadas que dependen de una colección, sección 8.1.3). Probamos todo esto haciendo unas ligeras modificaciones en el ya existente método `addDetails()` (listado 8.35).

Listado 8.35 addDetails() prueba propiedades calculadas y valores por defecto

```

private void addDetails() throws Exception {
    // Añadir una línea de detalle
    assertCollectionRowCount("details", 0);
    execute("Collection.new",
        "viewObject=xava_view_section0_details");
    setValue("product.number", "1");
    assertEquals("product.description",
        "Peopleware: Productive Projects and Teams");
    assertEquals("pricePerUnit", // @DefaultValueCalculator, sección 8.1.2
        "31.00");
    setValue("quantity", "2");
    assertEquals("amount", "62.00"); // Propiedad calculada, sección 8.1.1
    execute("Collection.save");
    assertNoErrors();
    assertEquals("number", 1);

    // Al grabar el primer detalle se graba el documento,
    // entonces verificamos que el número se calcula
    assertEquals("number", getNumber()); // Valor por defecto multiusuario, sección 8.2.1

    // Verificar propiedades calculadas del documento
    assertEquals("baseAmount", "62.00"); // Propiedades calculadas
    assertEquals("vat", "11.16"); // que dependen de una colección,
    assertEquals("totalAmount", "73.16"); // sección 8.1.3

    // Añadir otro detalle
    execute("Collection.new",
        "viewObject=xava_view_section0_details");
    setValue("product.number", "2");
    assertEquals("product.description",
        "Arco iris de lágrimas");
    assertEquals("pricePerUnit", // @DefaultValueCalculator, sección 8.1.2
        "15.00");
    setValue("pricePerUnit", "10.00"); // Modificar el valor por defecto
    setValue("quantity", "1");
    assertEquals("amount", "10.00"); // Propiedad calculada, sección 8.1.1
    execute("Collection.save");
    assertNoErrors();
    assertEquals("number", 2);

    // Verificar propiedades calculadas del documento
    assertEquals("baseAmount", "72.00"); // Propiedades calculadas
    assertEquals("vat", "12.96"); // que dependen de una colección,
    assertEquals("totalAmount", "84.96"); // sección 8.1.3
}

```

Como ves, con estas modificaciones sencillas probamos la mayoría de nuestro nuevo código. Nos quedan sólo las propiedades amount y estimatedProfit. Las cuales probaremos en la siguiente sección.

8.4.3 Sincronización entre propiedad persistente y calculada / @Formula

En la sección 8.2.2 usamos un método de retrollamada de JPA en CommercialDocument para tener una propiedad persistente, amount, sincronizada con una calculada, totalAmount. La propiedad amount solo se muestra en modo

lista.

En la sección 8.3 hemos creado una propiedad que usa `@Formula`, `estimatedProfit`. Esta propiedad se muestra solo en modo lista.

Obviamente, la forma más simple de probarlo es yendo a modo lista y verificando que los valores para estas dos propiedades son los esperados. En `testCreate()` llamamos a `verifyAmountAndEstimatedProfit()`. Veamos su código en el listado 8.36.

Listado 8.36 El nuevo método verifyAmountAndEstimatedProfit()

```
private void verifyAmountAndEstimatedProfit() throws Exception {
    execute("Mode.list"); // Cambia a modo lista
    setConditionValues(new String [] { // Filtra para ver en la lista solamente
        getCurrentYear(), getNumber() // el documento que acabamos de crear
    });
    execute("List.filter"); // Hace el filtro
    assertEqualsInList(0, 0, getCurrentYear()); // Verifica que el filtro
    assertEqualsInList(0, 1, getNumber()); // ha funcionado
    assertEqualsInList(0, "amount", "84.96"); // Confirma el importe
    assertEqualsInList(0, "estimatedProfit", "8.50"); // Confirma el beneficio estimado
    execute("Mode.detailAndFirst"); // Va a modo detalle
}
```

Dado que ahora vamos a modo lista y después volvemos a detalles, hemos de hacer una pequeña modificación en el método `verifyCreated()`, que es ejecutado justo después de `verifyAmountAndEstimatedProfit()`. Veamos la modificación en el listado 8.37.

Listado 8.37 verifyCreated() ahora no busca el documento recién creado

```
private void verifyCreated() throws Exception {
    setValue("year", getCurrentYear()); // Borramos estas líneas
    setValue("number", getNumber()); // para buscar el documento
    execute("CRUD.search"); // porque ya lo hemos buscado desde el modo lista

    // El resto de la prueba...
    ...
}
```

Quitamos estas líneas porque ahora no es necesario buscar el documento recién creado. Ahora en el método `verifyAmountAndEstimatedProfit()` vamos a modo lista y escogemos el documento, por tanto ya estamos editando el documento.

¡Enhorabuena! Ahora tus pruebas ya están sincronizadas con tu código. Es un buen momento para ejecutar todas las pruebas de tu aplicación.

8.5 Resumen

En este capítulo has aprendido algunas formas comunes de añadir lógica de negocio a tus entidades. No hay duda sobre la utilidad de las propiedades calculadas, los métodos de retrollamada o @Formula. Sin embargo, todavía tenemos muchas otras formas de añadir lógica a tu aplicación OpenXava, que vamos a aprender a usar.

En futuros capítulos verás como añadir validación, modificar el funcionamiento estándar del módulo y añadir tu propia lógica de negocio, entre otras formas de añadir lógica personalizada a tu aplicación.

Validación avanzada

capítulo 9

De momento solo hemos hecho validaciones básicas usando la anotación @Required. A veces es necesario escribir nuestra propia lógica de validación. Ya aprendiste como funciona la validación en el capítulo 3. Aquí vamos a añadir validaciones con lógica propia a tu aplicación.

9.1 Alternativas de validación

Vamos a refinar tu código para que el usuario no pueda asignar pedidos a una factura si los pedidos no han sido servidos todavía. Es decir, solo los pedidos servidos pueden asociarse a una factura. Aprovecharemos la oportunidad para explorar diferentes formas de hacer esta validación.

9.1.1 Añadir la propiedad delivered a Order

Para hacer esto, lo primero es añadir una nueva propiedad a la entidad Order. La propiedad delivered (listado 9.1).

Listado 9.1 Nueva propiedad delivered en la entidad Order

```
private boolean delivered;

public boolean isDelivered() {
    return delivered;
}

public void setDelivered(boolean delivered) {
    this.delivered = delivered;
}
```

Ahora actualiza el esquema de la base de datos y ejecuta las sentencias SQL del listado 9.2 contra la base de datos. Puedes hacerlo con la perspectiva de base de datos de Eclipse (ver sección 4.7).

Listado 9.2 Poner a false la columna delivered de la tabla CommercialDocument

```
update CommercialDocument
set delivered = false
```

Además es necesario añadir la propiedad delivered a la vista. Modifica la vista Order como muestra el listado 9.3.

Listado 9.3 Vista de Order modificada para incluir la propiedad delivered

```
@Views({
    @View( extendsView="super.DEFAULT",
        members="delivered; invoice { invoice }" // delivered añadida
    ),
    ...
})
public class Order extends CommercialDocument {
```

143 Capítulo 9: Validación avanzada

Ahora tienes una nueva propiedad `delivered` que el usuario puede marcar para indicar que el pedido ha sido servido. Ejecuta el nuevo código y marca algunos de los pedidos existentes como servidos.

9.1.2 Validar con `@EntityValidator`

En tu aplicación actual el usuario puede añadir cualquier pedido que le plazca a una factura usando el módulo `Invoice`, y puede asignar una factura a cualquier pedido desde el módulo `Order`. Vamos a restringir esto. Solo los pedidos servidos podrán añadirse a una factura.

La primera alternativa que usaremos para implementar esta validación es mediante `@EntityValidator`. Esta anotación te permite asignar a tu entidad una clase con la lógica de validación deseada. Anotemos tu entidad `Order` tal como muestra el listado 9.4.

Listado 9.4 `@EntityValidator` para la entidad `Order`

```
@EntityValidator(  
    value=DeliveredToBeInInvoiceValidator.class, // Clase con la lógica de validación  
    properties= {  
        @PropertyValue(name="year"), // El contenido de estas propiedades  
        @PropertyValue(name="number"), // se mueve desde la entidad Order  
        @PropertyValue(name="invoice"), // al validador antes de  
        @PropertyValue(name="delivered") // ejecutar la validación  
    }  
)  
public class Order extends CommercialDocument {
```

Cada vez que un objeto `Order` se crea o modifica un objeto del tipo `DeliveredToBeInInvoiceValidator` es creado, entonces las propiedades `year`, `number`, `invoice` y `delivered` se llenan con las propiedades del mismo nombre del objeto `Order`. Después de eso, el método `validate()` del validador se ejecuta. Puedes ver el código del validador en el listado 9.5.

Listado 9.5 Validador para que el pedido esté entregado para estar en una factura

```
package org.openxava.invoicing.validators; // En el paquete 'validators'  
  
import org.openxava.invoicing.model.*;  
import org.openxava.util.*;  
import org.openxava.validators.*;  
  
public class DeliveredToBeInInvoiceValidator  
    implements IValidator { // Ha de implementar IValidator  
  
    private int year; // Propiedades a ser inyectada desde Order  
    private int number;  
    private boolean delivered;  
    private Invoice invoice;  
  
    public void validate(Messages errors) // La lógica de validación  
        throws Exception
```

```

{
    if (invoice == null) return;
    if (!delivered) {
        errors.add( // Al añadir mensajes a errores la validación fallará
            "order_must_be_delivered", // Un id del archivo i18n
            year, number); // Argumentos para el mensaje
    }
}

// Getters y setters para year, number, delivered y invoice
...
}

```

La lógica de validación es extremadamente fácil, si una factura está presente y este pedido no ha sido servido añadimos un mensaje de error, por tanto la validación fallará. Has de añadir el mensaje de error en el archivo *Invoicing/i18n/Invoicing-messages_en.properties*. Tal como muestra el listado 9.6.

Listado 9.6 Internationalización del error en Invoicing-messages_en.properties

```
# Messages for the Invoicing application
order_must_be_delivered=Order {0}/{1} must be delivered in order to be added to
an Invoice
```

Ahora puedes intentar añadir pedidos a una factura con la aplicación, verás como los pedidos no servidos son rechazados. Como se ve en la figura 9.1.

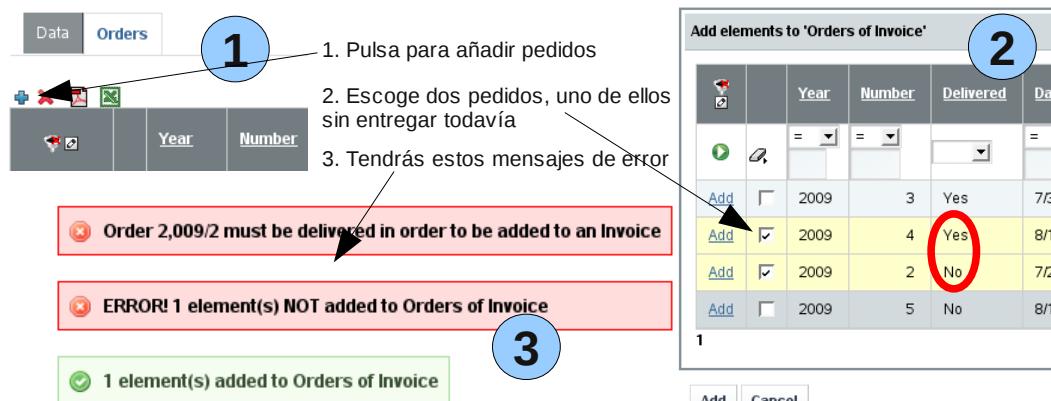


Figura 9.1 Añadir pedidos no entregados produce errores de validación

Ya tienes tu validación hecha con `@EntityValidator`. No es difícil, pero es un poco verboso, porque necesitas escribir una clase nueva solo para añadir 2 líneas de lógica. Aprendamos otras formas de hacer esta misma validación.

9.1.3 Validar con métodos de retrollamada JPA

Vamos a probar otra forma más sencilla de hacer esta validación, simplemente moviendo la lógica de validación desde la clase validador a la misma entidad Order, en este caso a un método @PreUpdate.

Lo primero es eliminar la clase DeliveredToBeInInvoiceValidator de tu proyecto. También quita la anotación @EntityValidator de tu entidad Order (listado 9.7).

Listado 9.7 Quitar la anotación @EntityValidator de la entidad Order

```
@EntityValidator(value=DeliveredToBeInInvoiceValidator.class,
    properties= { // Quita la anotación @EntityValidator
        @PropertyValue(name="year"),
        @PropertyValue(name="number"),
        @PropertyValue(name="invoice"),
        @PropertyValue(name="delivered")
    }
)
public class Order extends CommercialDocument {
```

Acabamos de eliminar la validación. Ahora, vamos a añadirla de nuevo, pero ahora dentro de la misma clase Order. Escribe el método validate() del listado 9.8 en tu clase Order.

Listado 9.8 Método de retrollamada JPA para validar en la entidad Order

```
@PreUpdate // Justo antes de actualizar la base de datos
private void validate() throws Exception {
    if (invoice != null && !isDelivered()) { // La lógica de validación
        throw new InvalidStateException( // La excepción de validación del
            new InvalidValue[] { // marco de validación Hibernate Validator
                new InvalidValue(
                    "Order must be delivered",
                    getClass(), "delivered",
                    true, this)
            }
        );
    }
}
```

Antes de grabar un pedido esta validación se ejecutará, si falla una InvalidStateException será lanzada. Esta excepción es del marco de validación Hibernate Validator, de esta forma OpenXava sabe que es una excepción de validación. Lo engorroso de esta solución es que InvalidStateException requiere un array de objetos InvalidValue. Lo bueno es que con solo un método dentro de tu entidad tienes la validación hecha.

9.1.4 Validar en el setter

Otra alternativa para hacer tu validación es poner tu lógica de validación

dentro del método `setter`. Es un enfoque simple y llano. Para probarlo, quita el método `validate()` de tu entidad `Order`, y modifica el método `setInvoice()` de la forma que ve en listado 9.9.

Listado 9.9 Validación dentro de un método `setter` de `invoice` en `Order`

```
public void setInvoice(Invoice invoice) {
    if (invoice != null && !isDelivered()) { // La lógica de validación
        throw new InvalidStateException( // La excepción de Hibernate Validator
            new InvalidValue[] {
                new InvalidValue(
                    "Order must be delivered",
                    getClass(), "delivered",
                    true, this)
            }
        );
    }
    this.invoice = invoice; // La asignación típica del setter
}
```

Esto funciona exactamente como las dos opciones anteriores. Es parecida a la alternativa del `@PrePersist`, solo que no depende de JPA, es una implementación básica de Java.

9.1.5 Validar con `Hibernate Validator`

Como opción final vamos a hacer la más breve. Consiste en poner tu lógica de validación dentro de un método booleano anotado con la anotación de Hibernate Validator `@AssertTrue`.

Para implementar esta alternativa primero quita la lógica de validación del método `setInvoice()`. Después, añade `isDeliveredToBeInInvoice()` del listado 9.10 a tu entidad `Order`.

Listado 9.10 Validar `Order` usando una anotación `@AssertTrue`

```
@AssertTrue // Antes de grabar confirma que el método devuelve true, si no lanza una excepción
private boolean isDeliveredToBeInInvoice() {
    return invoice == null || isDelivered(); // La lógica de validación
}
```

Esta es la forma más simple de validar, porque solo anotamos el método con la validación, y es `Hibernate Validator` el responsable de llamar este método al grabar, y lanzar la `InvalidStateException` correspondiente si la validación no pasa.

9.1.6 Validar al borrar con `@RemoveValidator`

Las validaciones que hemos visto hasta ahora se hacen cuando la entidad se modifica, pero a veces es útil hacer la validación justo al borrar la entidad, y usar

147 Capítulo 9: Validación avanzada

la validación para vetar el borrado de la misma.

Vamos a modificar la aplicación para impedir que un usuario borre un pedido si éste tiene una factura asociada. Para hacer esto anota tu entidad Order con @RemoveValidator, como muestra el listado 9.11.

Listado 9.11 @RemoveValidator para la entidad Order

```
@RemoveValidator(OrderRemoveValidator.class) // La clase con la validación
public class Order extends CommercialDocument {
```

Ahora, antes de borrar un pedido la lógica de OrderRemoveValidator se ejecuta, y si la validación falla el pedido no se borra. Veamos el código de este validador en el listado 9.12.

Listado 9.12 Validador para validar si un pedido puede borrarse

```
package org.openxava.invoicing.validators; // En el paquete 'validators'

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

public class OrderRemoveValidator
    implements IRemoveValidator { // Ha de implementar IRemoveValidator

    private Order order;

    public void setEntity(Object entity) throws Exception // La entidad a borrar se injectará
        // con este método antes de la validación
    {
        this.order = (Order) entity;
    }

    public void validate(Messages errors) throws Exception // La lógica de validación
        // Añadiendo mensajes
    {
        if (order.getInvoice() != null) {
            errors.add("cannot_delete_order_with_invoice"); // Añadiendo mensajes
        }
    }
}
```

La lógica de validación está en el método validate(). Antes de llamarlo la entidad a validar es inyectada usando setEntity(). Si se añaden mensajes al objeto errors la validación fallará y la entidad no se borrará. Has de añadir el mensaje de error en el archivo *Invoicing/i18n/Invoicing-messages_en.properties*. Véase el listado 9.13.

Listado 9.13 Internacionalización del error en Invoicing-messages_en.properties

```
cannot_delete_order_with_invoice=Order with invoice cannot be deleted
```

Ahora si intentas borrar un pedido con una factura asociada obtendrás un

mensaje de error y el borrado no se producirá.

Puedes ver que usar un `@RemoveValidator` no es difícil, pero es un poco verboso. Has de escribir una clase nueva solo para añadir un simple “if”. Examinemos una alternativa más breve.

9.1.7 Validar al borrar con un método de retrollamada JPA

Vamos a probar otra forma más simple de hacer esta validación al borrar, moviendo la lógica de validación desde la clase validador a la misma entidad `Order`, en este caso en un método `@PreRemove`.

El primer paso es eliminar la clase `OrderRemoveValidator` de tu proyecto. Además quita la anotación `@RemoveValidator` de tu entidad `Order` (listado 9.14).

Listado 9.14 Quitar la anotación `@RemoveValidator` de la entidad `Order`

```
@RemoveValidator(OrderRemoveValidator.class) // Quitamos @RemoveValidator
public class Order extends CommercialDocument {
```

Hemos quitado la validación. Añadámosla otra vez, pero ahora dentro de la misma clase `Order`. Añade el método `validateOnRemove()` del listado 9.15 a la clase `Order`.

Listado 9.15 Método de retrollamada JPA para validar al borrar en `Order`

```
@PreRemove // Justo antes de borrar la entidad
private void validateOnRemove() {
    if (invoice != null) { // La lógica de validación
        throw new IllegalStateException( // Lanza una excepción runtime
            XavaResources.getString( // Para obtener un mensaje de texto
                "cannot_delete_order_with_invoice"));
    }
}
```

Antes de borrar un pedido esta validación se efectuará, si falla se lanzará una `IllegalStateException`. Puedes lanzar cualquier excepción *runtime* para abortar el borrado. Tan solo con un método dentro de la entidad tienes la validación hecha.

9.1.8 ¿Cuál es la mejor forma de validar?

Has aprendido varias formas de hacer la validación sobre tus clases del modelo. ¿Cuál de ellas es la mejor? Todas ellas son opciones válidas. Depende de tus circunstancias y preferencias personales. Si tienes una validación que no es trivial y es reutilizable en varios puntos de tu aplicación, entonces usar un `@EntityValidator` y `@RemoveValidator` es una buena opción. Por otra parte, si quieres usar tu modelo fuera de OpenXava y sin JPA, entonces el uso de la

validación en los *setters* es mejor.

En nuestro caso particular hemos optado por `@AssertTrue` para la validación “el pedido ha de estar servido para estar en una factura” y por `@PreRemove` para la validación al borrar. Ya que son las alternativas más simples que funcionan.

9.2 Crear tu propia anotación de Hibernate Validator

Las técnicas mencionadas hasta ahora son muy útiles para la mayoría de las validaciones de tus aplicaciones. Sin embargo, a veces te encuentras con algunas validaciones que son muy genéricas y quieres usarlas una y otra vez. En este caso definir tu propia anotación de Hibernate Validator puede ser una buena opción. Definir un Hibernate Validator es más largo y engorroso que lo que hemos visto hasta ahora, pero usarlo y reusarlo es simple, tan solo añadir una anotación a tu propiedad o clase.

Vamos a aprender como crear un Hibernate Validator.

9.2.1 Usar un Hibernate Validator en tu entidad

Usar un Hibernate Validator es superfácil. Simplemente anota tu propiedad, como ves en el listado 9.16.

Listado 9.16 Usar una anotación de Hibernate Validator para nuestra propiedad

```
@ISBN // Esta anotación indica que esta propiedad tiene que validarse como un ISBN
private String isbn;
```

Solo con añadir `@ISBN` a tu propiedad, y ésta será validada justo antes de que la entidad se grabe en la base de datos, ¡genial! El problema es que `@ISBN` no está incluida como un validador predefinido en el marco de validación Hibernate Validator. Esto no es un gran problema, si quieres una anotación `@ISBN`, hazla tú mismo. De hecho, vamos a crear la anotación de validación `@ISBN` en esta sección.

Antes de nada, añadimos una nueva propiedad `isbn` a `Product`. Edita tu clase `Product` y añádele el código del listado 9.17.

Listado 9.17 Nueva propiedad `isbn` en `Producto`

```
@Column(length=10)
private String isbn;

public String getIsbn() {
    return isbn;
}

public void setIsbn(String isbn) {
    this.isbn = isbn;
}
```

{}

Actualiza el esquema de tu base de datos, y ejecuta el módulo Product con tu navegador. Sí, la propiedad isbn ya está ahí. Ahora, puedes añadir la validación.

9.2.2 Definir tu propia anotación ISBN

Creemos la anotación @ISBN. Primero, crea un paquete en tu proyecto llamado org.openxava.invoicing.annotations, entonces sigue las instrucciones de la figura 9.2 para crear una nueva anotación llamada ISBN.

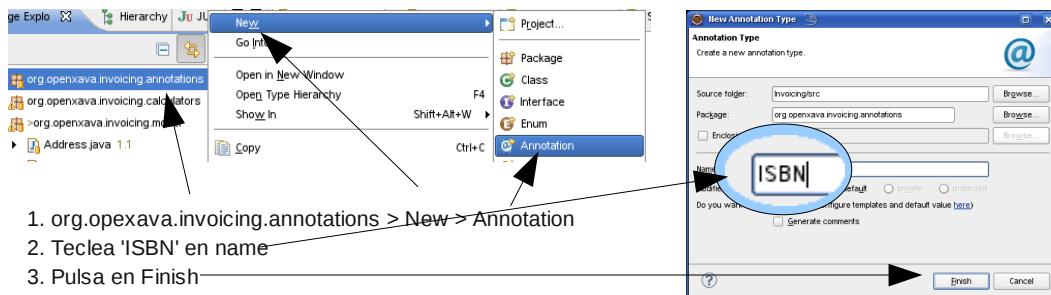


Figura 9.2 Crear la nueva anotación ISBN con Eclipse

Edita el código de tu recién creada anotación ISBN y déjala como la del listado 9.18.

Listado 9.18 Código para la anotación ISBN

```
package org.openxava.invoicing.annotations; // En el paquete annotations

import java.lang.annotation.*;
import org.hibernate.validator.*;
import org.openxava.invoicing.validators.*;

@ValidatorClass(ISBNValidator.class) // Esta clase contiene la lógica de validación
@Target({ ElementType.FIELD, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
public @interface ISBN { // Una definición de anotación Java convencional

    String message() default "ISBN does not exist"; // El mensaje si la validación falla
}
```

Como puedes ver, es una definición de anotación normal y corriente, solo que usas @ValidatorClass para indicar la clase con la lógica de validación. Escribamos la clase ISBNValidator.

9.2.3 Usar Apache Commons Validator para implementar la lógica

Vamos a escribir la clase ISBNValidator con la lógica de validación para un ISBN. En lugar de escribir nosotros mismos la lógica para validar un ISBN

151 Capítulo 9: Validación avanzada

usaremos el proyecto Commons Validator¹² de Apache. Commons Validator contiene algoritmos de validación para direcciones de correo electrónico, fechas, URL y así por el estilo. El *commons-validator.jar* se incluye por defecto en los proyectos OpenXava, por tanto lo puedes usar sin ninguna configuración adicional.

El código para ISBNValidator está en el listado 9.19.

Listado 9.19 Versión inicial de ISBNValidator

```
package org.openxava.invoicing.validators; // En el paquete validators

import org.hibernate.validator.*;
import org.openxava.util.*;
import org.openxava.invoicing.annotations.*;

public class ISBNValidator
    implements Validator<ISBN> { // Tiene que implementar Validator<ISBN>

    private static org.apache.commons.validator.ISBNValidator
        validator = // De Commons Validator
            new org.apache.commons.validator.ISBNValidator();

    public void initialize(ISBN isbn) {
    }

    public boolean isValid(Object value) { // Contiene la lógica de validación
        if (Is.empty(value)) return true;
        return validator
            .isValid(value.toString()); // Usa Commons Validator
    }
}
```

Como ves, la clase validador tiene que implementar Validator del paquete *org.hibernate.validator*. Esto fuerza a tu validador a implementar *initialize()* e *isValid()*. El método *isValid()* contiene la lógica de validación. Fíjate que si el elemento a validar está vacío asumimos que es válido, porque validar si un valor está presente es responsabilidad de otras anotaciones, como *@Required*, y no de *@ISBN*.

En este caso la lógica de validación es sencillísima, porque nos limitamos a llamar al validador ISBN de Apache Commons Validator.

@ISBN está listo para usar. Para hacerlo anota tu propiedad *isbn* con él. Puedes ver cómo en el listado 9.20.

Listado 9.20 La propiedad isbn anotada con @ISBN

```
@Column(length=10) @ISBN
private String isbn;
```

12 <http://commons.apache.org/validator/>

Ahora, puedes probar tu módulo, y verificar que el ISBN que introduces se valida correctamente. Enhorabuena, has escrito tu primer Hibernate Validator. No ha sido tan difícil: una anotación, una clase.

Este @ISBN es suficientemente bueno para usarlo en la vida real, sin embargo, vamos a mejorarlo un poco más, y así tendremos la posibilidad de experimentar con algunas posibilidades interesantes.

9.2.4 Llamar a un servicio web REST para validar el ISBN

Aunque la mayoría de los validadores tienen una lógica simple, puedes crear validadores con una lógica compleja si lo necesitas. Por ejemplo, en el caso de nuestro ISBN, queremos, no solo verificar el formato correcto, sino también comprobar que existe de verdad un libro con ese ISBN. Una forma de hacer esto es usando servicios web.

Como seguramente ya sepas, un servicio web es una funcionalidad que reside en un servidor web y que tú puedes llamar desde tu programa. La forma tradicional de desarrollar servicios web es mediante los estándares WS-*, como SOAP, UDDI, etc. Aunque, hoy en día está surgiendo una forma más simple de desarrollar servicios, REST. REST consiste básicamente en usar la ya existente “forma de trabajar” de internet para comunicación entre programas. Llamar a un servicio REST consiste en usar una URL web convencional para obtener un recurso de un servidor web. Este recurso usualmente contiene datos en formato XML, HTML, JSON, etc. En otras palabras, los programas usan internet de la misma manera que lo hacen los usuarios con sus navegadores.

Hay bastantes sitios con servicios web SOAP y REST para consultar el ISBN de un libro, pero no suele ser gratis. Por tanto, vamos a usar una alternativa más barata, que va a ser llamar a un sitio web convencional para hacer la búsqueda del ISBN, y examinar después la página resultado para determinar si la búsqueda ha funcionado. Algo así como un servicio web pseudo-REST.

Para llamar a la página web usaremos el marco de trabajo HtmlUnit¹³. Aunque el principal cometido de este marco de trabajo sea crear pruebas para tus aplicaciones web, puedes usarlo para leer cualquier página web. Lo usaremos porque es más fácil que otras librerías con este propósito, como por ejemplo Apache Commons HttpClient. Observa lo simple que es leer una página web con HtmlUnit en el listado 9.21.

Listado 9.21 Leer una página web usando HtmlUnit

```
 WebClient client = new WebClient();
 HtmlPage page = (HtmlPage) client.getPage("http://www.openxava.org/");
```

13 <http://htmlunit.sourceforge.net/>

Después de esto, puedes usar el objeto page para manipular la página leída.

OpenXava usa HtmlUnit como marco subyacente para las pruebas, por tanto ya está incluido en OpenXava, pero no se incluye por defecto en las aplicaciones OpenXava, así que tienes que incluirlo tú mismo en tu aplicación. Para hacerlo, copia los archivos *htmlunit.jar*, *commons-httpclient.jar*, *commons-codec.jar*, *htmlunit-core-js.jar*, *commons-lang.jar*, *xercesImpl.jar*, *xalan.jar*, *cssparser.jar*, *sac.jar* y *nekohtml.jar*, desde la carpeta *OpenXava/lib* a la carpeta *Invoicing/web/WEB-INF/lib*. Después de copiar estos archivos refresca el proyecto Invoicing pulsando F5.

Modifiquemos ISBNValidator para que haga uso de este servicio REST. Puedes ver el resultado en el listado 9.22.

Listado 9.22 ISBNValidator que usa un servicio web REST

```
package org.openxava.invoicing.validators;

import org.apache.commons.logging.*;
import org.hibernate.validator.*;
import org.openxava.util.*;
import org.openxava.invoicing.annotations.*;

import com.gargoylesoftware.htmlunit.*; // Para usar HtmlUnit
import com.gargoylesoftware.htmlunit.html.*; // Para usar HtmlUnit

public class ISBNValidator implements Validator<ISBN> {

    private static Log log = LogFactory.getLog(ISBNValidator.class);
    private static org.apache.commons.validator.ISBNValidator
        validator =
            new org.apache.commons.validator.ISBNValidator();

    public void initialize(ISBN isbn) {
    }

    public boolean isValid(Object value) {
        if (Is.empty(value)) return true;
        if (!validator.isValid(value.toString())) return false;
        return isbnExists(value); // Aquí hacemos la llamada REST
    }

    private boolean isbnExists(Object isbn) {
        try {
            WebClient client = new WebClient();
            HtmlPage page = (HtmlPage) client.getPage( // Llamamos a
                "http://www.bookfinder4u.com/" + // bookdiner4u
                "IsbnSearch.aspx?isbn=" + // con una URL para buscar
                isbn + "&mode=direct"); // por ISBN
            return page.asText() // Comprueba si la página resultante contiene
                .indexOf("ISBN: " + isbn) >= 0; // el ISBN buscado
        }
        catch (Exception ex) {
            log.warn("Impossible to connect to bookfinder4u" +
                "to validate the ISBN. Validation fails", ex);
            return false; // Si hay algún error asumimos que la validación ha fallado
        }
    }
}
```

```

    }
}
}
```

Simplemente buscamos una página usando como argumento en la URL el ISBN, si la página resultante contiene el ISBN buscado quiere decir que la búsqueda ha sido satisfactoria, si no es que la búsqueda ha fallado. El método `page.asText()` devuelve el contenido de la página HTML sin las marcas HTML, es decir, con solo la información textual.

Puedes usar este truco con cualquier sitio que te permita hacer búsquedas, así puedes consultar virtualmente millones de sitios web desde tu aplicación. En un servicio REST más puro el resultado hubiera sido un documento XML en vez de uno HTML, pero hubieras tenido que pasar por caja.

Prueba ahora tu aplicación y verás como si introduces un ISBN no existente la validación falla.

9.2.5 Añadir atributos a tu anotación

Creas una anotación Hibernate Validator cuando quieres reutilizar la validación varias veces, usualmente en varios proyectos. En este caso, necesitas hacer tu validación adaptable, para que sea reutilizable de verdad. Por ejemplo, en el proyecto actual buscar en www.bookfinder4u.com el ISBN es conveniente, pero en otro proyecto, o incluso en otra entidad de tu actual proyecto, puede que no quieras hacer esa búsqueda. Necesitas hacer tu anotación más flexible.

La forma de añadir esta flexibilidad a tu anotación de validación es mediante los atributos. Por ejemplo, podemos añadir un atributo de búsqueda booleano a nuestra anotación ISBN para poder escoger si queremos buscar el ISBN en internet para validar o no. Para hacerlo, simplemente añade el atributo `search` al código de la anotación ISBN, tal como muestra el listado 9.23.

Listado 9.23 Anotación ISBN con el atributo search

```

public @interface ISBN {
    boolean search() default true; // Para (des)activar la búsqueda web al validar
    String message() default "ISBN does not exist";
}
```

Este nuevo atributo `search` puede leerse de la clase validador. Míralo en el listado 9.24.

Listado 9.24 ISBNValidator con el atributo search

```
public class ISBNValidator implements Validator<ISBN> {

    ...

    private boolean search; // Almacena la opción search

    public void initialize(ISBN isbn) { // Lee los atributos de la anotación
        this.search = isbn.search();
    }

    public boolean isValid(Object value) {
        if (Is.empty(value)) return true;
        if (!validator.isValid(value.toString())) return false;
        return search?isbnExists(value):true; ← Usa search
    }

    ...

}
```

Aquí ves la utilidad del método `initialize()`, que lee la anotación para inicializar el validador. En este caso simplemente almacenamos el valor de `isbn.search()` para preguntar por él en `isValid()`.

Ahora puedes escoger si quieres llamar a nuestro servicio pseudo-REST o no para hacer la validación ISBN. Véase el listado 9.25.

Listado 9.25 Usar @ISBN con el atributo search

```
@ISBN(search=false) // En este caso no se hace un búsqueda en la web para validar el ISBN
private String isbn;
```

Usando esta técnica puedes añadir cualquier atributo que necesites para dar más flexibilidad a tu anotación ISBN.

¡Enhorabuena! Has aprendido como crear tu propia anotación de Hibernate Validator, y de paso a usar la útil herramienta HtmlUnit.

9.3 Pruebas JUnit

Nuestra meta no es desarrollar una ingente cantidad de código, sino crear software de calidad. Al final, si creas software de calidad acabarás escribiendo más cantidad de software, porque podrás dedicar más tiempo a hacer cosas nuevas y excitantes, y menos depurando legiones de *bugs*. Y tú sabes que la única forma de conseguir calidad es mediante las pruebas automáticas, por tanto actualicemos nuestro código de prueba.

9.3.1 Probar la validación al añadir a una colección

Recuerda que hemos refinado tu código para que el usuario no pueda asignar

pedidos a una factura si los pedidos no están servidos. Después de esto, tu actual `testAddOrders()` de `InvoiceTest` puede fallar, porque trata de añadir el primer pedido, y es posible que ese primer pedido no esté marcado como servido.

Modifiquemos la prueba para que funcione y también para comprobar la nueva funcionalidad de validación. Mira el listado 9.26.

Listado 9.26 `testAddOrders()` de `InvoiceTest` prueba la validación en los pedidos

```
public void testAddOrders() throws Exception {
    assertListNotEmpty();
    execute("List.orderBy", "property=number");
    execute("Mode.detailAndFirst");
    execute("Sections.change", "activeSection=1");
    assertCollectionRowCount("orders", 0);
    execute("Collection.add",
        "viewObject=xava_view_section1_orders");
    execute("AddToCollection.add", "row=0"); // Ahora no seleccionamos al azar

    checkFirstOrderWithDeliveredEquals("Yes"); // Selecciona un pedido entregado
    checkFirstOrderWithDeliveredEquals("No"); // Selecciona uno no entregado
    execute("AddToCollection.add"); // Tratamos de añadir ambos
    assertError( // Un error, porque el pedido no entregado no se puede añadir
        "ERROR! 1 element(s) NOT added to Orders of Invoice");
    assertMessage(// Un mensaje de confirmación, porque el pedido entregado ha sido añadido
        "1 element(s) added to Orders of Invoice");

    assertCollectionRowCount("orders", 1);
    checkRowCollection("orders", 0);
    execute("Collection.removeSelected",
        "viewObject=xava_view_section1_orders");
    assertCollectionRowCount("orders", 0);
}
```

Hemos modificado la parte de la selección de pedidos a añadir, antes seleccionábamos el primero, no importaba si estaba servido o no. Ahora seleccionamos un pedido servido y otro no servido, de esta forma comprobamos que el pedido servido se añade y el no servido es rechazado.

La pieza que nos falta es la forma de seleccionar los pedidos. Esto es el trabajo del método `checkFirstOrderWithDeliveredEquals()`. Veámoslo en el listado 9.27.

Listado 9.27 Método para marcar los pedidos seleccionándolos por 'delivered'

```
private void checkFirstOrderWithDeliveredEquals(String value)
    throws Exception
{
    int c = getListRowCount(); // El total de filas visualizadas en la lista
    for (int i=0; i<c; i++) {
        if (value.equals(
            getValueInList(i, 2))) // 2 es la columna 'delivered'
        {
            checkRow(i);
            return;
        }
    }
}
```

```

        }
        fail("Must be at least one row with delivered=" + value);
    }
}

```

Aquí ves una buena técnica para hacer un bucle sobre los elementos visualizados de una lista para seleccionarlos y coger algunos datos, o cualquier otra cosa que quieras hacer con los datos de la lista.

9.3.2 Probar validación al asignar una referencia y al borrar

Desde el módulo Invoice el usuario no pueda asignar pedidos a una factura si los pedidos no están servidos, por lo tanto, desde el módulo Order el usuario tampoco debe poder asignar una factura a un pedido si éste no está servido. Es decir, hemos de probar también la otra parte de la asociación. Lo haremos modificando el actual `testSetInvoice()` de `OrderTest`.

Además, aprovecharemos este caso para probar la validación al borrar que vimos en las secciones 9.1.6 y 9.1.7. Allí modificamos la aplicación para impedir que un usuario borrara un pedido si éste tenía una factura asociada. Ahora probaremos este hecho.

Todo esto está en el `testSetInvoice()` mejorado que puedes ver en el listado 9.28.

Listado 9.28 `testSetInvoice()` prueba las validaciones al grabar y al borrar

```

public void testSetInvoice() throws Exception {
    assertListNotEmpty();
    execute("List.orderBy", "property=number"); // Establece el orden de la lista
    execute("Mode.detailAndFirst");
    assertEquals("delivered", "false"); // El pedido no ha de estar entregado
    execute("Sections.change", "activeSection=1");
    assertEquals("invoice.number", "");
    assertEquals("invoice.year", "");
    execute("Reference.search",
            "keyProperty=invoice.year");
    String year = getValueInList(0, "year");
    String number = getValueInList(0, "number");
    execute("ReferenceSearch.choose", "row=0");
    assertEquals("invoice.year", year);
    assertEquals("invoice.number", number);

    // Los pedidos no entregados no pueden tener factura
    execute("CRUD.save");
    assertEquals(1, assertErrorsCount()); // No podemos grabar porque no ha sido entregado
    setValue("delivered", "true");
    execute("CRUD.save"); // Con delivered=true podemos grabar el pedido
    assertNoErrors();

    // Un pedido con factura no se puede borrar
    execute("Mode.list"); // Vamos al modo lista y
    execute("Mode.detailAndFirst"); // volvemos a detalle para cargar el pedido grabado
    execute("CRUD.delete"); // No podemos borrar porque tiene una factura asociada
}

```

```

assertErrorsCount(1);

// Restaurar los valores originales
setValue("delivered", "false");
setValue("invoice.year", "");
execute("CRUD.save");
assertNoErrors();
}
}

```

La prueba original solo buscaba una factura, ni siquiera intentaba grabarla. Ahora, hemos añadido código al final para probar la grabación de un pedido marcado como servido, y marcado como no servido, de esta forma comprobamos la validación. Después de eso, tratamos de borrar el pedido, el cual tiene una factura, así probamos también la validación al borrar.

9.3.3 Probar el Hibernate Validator propio

Solo nos queda probar tu Hibernate Validator ISBN, el cual usa un servicio REST para hacer la validación. Simplemente hemos de escribir una prueba que trate de asignar un ISBN incorrecto, uno inexistente y uno correcto a un producto, y ver que pasa. Para hacer esto añadimos un método `testISBNValidator()` a `ProductTest`. Lo puedes ver en el listado 9.29.

Listado 9.29 testISBNValidator() de ProductTest prueba el Hibernate Validator

```

public void testISBNValidator() throws Exception {
    // Buscar product1
    execute("CRUD.new");
    setValue("number", Integer.toString(product1.getNumber()));
    execute("CRUD.refresh");
    assertEquals("description", "JUNIT Product 1");
    assertEquals("isbn", "");

    // Con un formato de ISBN incorrecto
    setValue("isbn", "1111");
    execute("CRUD.save"); // Falla por el formato (apache commons validator)
    assertEquals("1111 is not a valid value for Isbn of " +
        "Product: ISBN does not exist");

    // ISBN no existe aunque tiene un formato correcto
    setValue("isbn", "1234367890");
    execute("CRUD.save"); // Falla porque no existe (el servicio REST)
    assertEquals("1234367890 is not a valid value for Isbn of " +
        "Product: ISBN does not exist");

    // ISBN existe
    setValue("isbn", "0932633439");
    execute("CRUD.save"); // No falla
    assertNoErrors();
}
}

```

Seguramente la prueba manual que hacías mientras estabas escribiendo el validador `@ISBN` era parecida a esta. Por eso, si hubieras escrito tu código de

prueba antes que el código de la aplicación¹⁴, lo hubieras podido usar mientras que desarrollabas, lo cual es más eficiente que repetir una y otra vez a mano las pruebas con el navegador.

Fíjate que si usas `@ISBN(search=false)` esta prueba no funciona porque no solo comprueba el formato sino que también hace la búsqueda con el servicio REST. Por tanto, has de usar `@ISBN` sin atributos para anotar la propiedad `isbn` y poder ejecutar esta prueba.

Ahora ejecuta todas las prueba de tu aplicación Invoicing para verificar que todo sigue en su sitio.

9.4 Resumen

En este capítulo has aprendido varias formas de hacer validación en una aplicación OpenXava. Además, ahora estás preparado para encapsular toda la lógica de validación reutilizable en anotaciones usando Hibernate Validator.

La validación es una parte importante de la lógica de tu aplicación, y te ánimo a que la pongas en el modelo, es decir en las entidades; tal y como este capítulo ha mostrado. Aun así, a veces es conveniente poner algo de lógica fuera de las clases del modelo. Aprenderás a hacer esto en los siguientes capítulos.

¹⁴ Ventajas de hacer primero las pruebas:
<http://www.extremeprogramming.org/rules/testfirst.html>

*Refinar el
comportamiento
predefinido*

capítulo 10

Espero que estés muy contento con el código de tu aplicación Invoicing. Es realmente simple, básicamente tienes entidades, clases simples que modelan tu problema. Toda la lógica de negocio está en esas entidades, y OpenXava genera una aplicación con un comportamiento decente a partir de ellas.

No solo de lógica de negocio vive el hombre. Un buen comportamiento también es importante. Seguramente, te habrás encontrado con que o bien tu o bien tu usuario queréis un comportamiento diferente al estándar de OpenXava, al menos para ciertas partes de tu aplicación. Refinar el comportamiento predefinido a veces es necesario si quieres que tu usuario esté cómodo.

El comportamiento de la aplicación viene dado por los controladores. Un controlador es una colección de acciones. Una acción contiene el código a ejecutar cuando el usuario pulsa en un vínculo o botón. Puedes definir tus propios controladores y acciones y asociarlos a tus módulos o entidades, de esta forma refinas la forma en que OpenXava se comporta.

En este capítulo refinaremos los controladores y acciones estándar para poder personalizar el comportamiento de tu aplicación Invoicing.

10.1 Acciones personalizadas

Por defecto, un módulo OpenXava te permite manejar tu entidad de una forma bastante buena: es posible añadir, modificar, borrar, buscar, generar informes PDF y exportar a Excel las entidades. Estas acciones por defecto están contenidas en el controlador `Typical`. Puedes refining o extender el comportamiento de tu módulo definiendo tu propio controlador. Esta sección te enseñará como definir tu propio controlador y escribir tus acciones personalizadas.

10.1.1 Controlador `Typical`

Por defecto el módulo `Invoice` usa las acciones del controlador `Typical`. El controlador `Typical` está definido en `default-controllers.xml` que se encuentra en la carpeta `OpenXava/xava` de tu `workspace`. Una definición de controlador es un fragmento de XML con una lista de acciones. OpenXava aplica por defecto el controlador `Typical` a todos los módulos. Puedes ver su definición en el listado 10.1.

Listado 10.1 Definición del controlador “`Typical`” en `default-controllers.xml`

```
<controller name="Typical">      <!-- "Typical" hereda sus acciones de los controladores -->
    <extends controller="Navigation"/>  <!-- "Navigation", -->
    <extends controller="CRUD"/>        <!-- "CRUD" -->
    <extends controller="Print"/>       <!-- y "Print" -->
</controller>
```

163 Capítulo 10: Refinar el comportamiento predefinido

Aquí puedes ver como se puede definir un controlador a partir de otros controladores. Este es un uso sencillo de la herencia. En este caso el controlador Typical tiene todas las acciones de los controladores Navigation, Print y CRUD. Navigation tiene las acciones para navegar por los registros en modo detalle. Print tiene las acciones para imprimir informes PDF y exportar a Excel, y CRUD tiene las acciones para crear, leer, actualizar y borrar. El listado 10.2 muestra un extracto del controlador CRUD.

Listado 10.2 Definición del controlador “CRUD” en default-controllers.xml

```
<controller name="CRUD">

    <action name="new"
        class="org.openxava.actions.NewAction"
        image="images/new.gif"
        keystroke="Control N">
        <!--
            name="new": Nombre para referenciar la acción desde otras partes
            class="org.openxava.actions.NewAction": La clase con la lógica de la acción
            image="images/new.gif": Imagen a mostrar para esta acción
            keystroke="Control N": Teclas que se pueden pulsar para ejecutar la acción
        -->
        <set property="restoreModel"
            value="true"/>    <!-- La propiedad restoreModel de la acción
                                se pondrá a true antes de ejecutarla -->
    </action>

    <action name="save"
        mode="detail"
        by-default="if-possible"
        class="org.openxava.actions.SaveAction"
        image="images/save.gif"
        keystroke="Control S"/>
        <!--
            mode="detail": Esta acción se mostrará solo en modo detalle
            by-default="if-possible": Esta acción se ejecutará cuando el usuario pulse INTRO
        -->

    <action name="delete" mode="detail"
        confirm="true"
        class="org.openxava.actions.DeleteAction"
        image="images/delete.gif"
        keystroke="Control D"/>
        <!-- confirm="true": Pide confirmación al usuario antes de ejecutar la acción -->

    ...

</controller>
```

Aquí se ve como definir las acciones. Básicamente consiste en vincular un nombre con una clase con la lógica a ejecutar. Además, define una imagen y un atajo de teclado. También vemos como se puede configurar la acción usando `<set />`.

Las acciones se muestran por defecto en modo lista y detalle, aunque puedes, por medio del atributo mode, especificar que sea mostrada solo en modo lista (list) o detalle (detail).

10.1.2 Refinar el controlador para un módulo

Empezaremos refinando la acción para borrar del módulo Invoice. Nuestro objetivo es que cuando el usuario pulse en el botón de borrar, la factura no sea borrada de la base de datos, sino que simplemente se marque como borrada. De esta forma, podemos recuperar las facturas borradas si fuese necesario.

La figura 10.1 muestra las acciones de Typical. Queremos todas estas acciones en nuestro módulo Invoice, con la excepción de que vamos a escribir nuestra propia lógica para la acción de borrar.

Para definir tu propio controlador para Invoice has de crear un archivo llamado *controllers.xml* en la carpeta *xava* de tu proyecto y escribir el código del listado 10.3 en él.

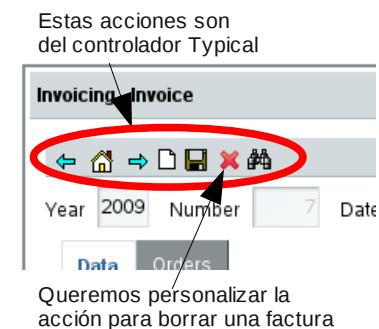


Figura 10.1 Acciones de Typical

Listado 10.3 El archivo controllers.xml con la definición del controlador Order

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE controllers SYSTEM "dtds/controllers.dtd">
<controllers>
    <controller name="Invoice"> <!-- El mismo nombre que la entidad -->
        <extends controller="Typical"/> <!-- Hereda todas las acciones de Typical -->
        <!-- Typical ya tiene una acción 'delete', al usar el mismo nombre la sobreescrivimos -->
        <action name="delete"
            mode="detail" confirm="true"
            class="org.openxava.invoicing.actions.DeleteInvoiceAction"
            image="images/delete.gif"
            keystroke="Control D"/>
    </controller>
</controllers>
```

Para definir un controlador para tu entidad, has de crear un controlador con el mismo nombre que la entidad. Es decir, si existe un controlador llamado “Invoice”, cuando ejecutes el módulo Invoice éste será el controlador a usar en vez de Typical.

Extendemos el controlador Invoice de Typical, así todas las acciones de Typical están disponible en tu módulo Invoice. Cualquier acción que definas en tu controlador Invoice estará disponible como un botón para que el usuario pueda pulsarlo. Aunque en este caso hemos llamado a nuestra acción “delete”, precisamente el nombre de una acción del controlador Typical, de esta forma estamos anulando la acción de Typical. Es decir, solo una acción delete se mostrará al usuario y será la nuestra.

10.1.3 Escribir tu propia acción

Escribamos pues la primera versión de nuestra acción delete. Mírala en el listado 10.4.

Listado 10.4 Primera versión tonta de DeleteInvoiceAction

```
package org.openxava.invoicing.actions; // En el paquete actions

import org.openxava.actions.*; // Para usar ViewBaseAction

public class DeleteInvoiceAction
    extends ViewBaseAction { // ViewBaseAction tiene getView(), addMessage(), etc

    public void execute() throws Exception { // La lógica de la acción
        addMessage( // Añade un mensaje para mostrar al usuario
            "Don't worry! I have cleared only the screen");
        getView().clear(); // getView() devuelve el objeto xava_view
                           // clear() borrar los datos en la interfaz de usuario
    }

}
```

Una acción es una clase simple. Tiene un método execute() con la lógica a hacer cuando el usuario pulse en el botón o vínculo correspondiente. Una acción ha de implementar la interfaz org.openxava.actions.IAction, aunque normalmente es más práctico extender de BaseAction, ViewBaseAction o cualquier otra acción base del paquete org.openxava.actions.

ViewBaseAction tiene una propiedad view que puedes usar desde dentro de execute() mediante getView(). Este objeto del tipo org.openxava.view.View permite manejar la interfaz de usuario, en este caso borramos los datos visualizados usando getView().clear().

También usamos addMessage(). Todos los mensajes añadidos con addMessage() se mostrarán al usuario al final de la ejecución de la acción. Puedes, bien añadir el mensaje a mostrar, o bien un id de una entrada en i18n/Invoicing-messages_en.properties.

La figura 10.2 muestra el comportamiento del módulo Invoice después de añadir la acción de borrar personalizada. Por supuesto, este es un comportamiento

tonto. Añadamos el comportamiento real.

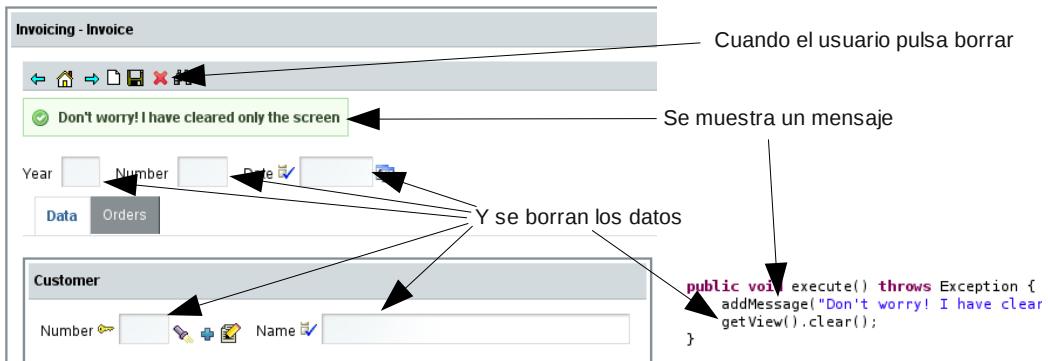


Figura 10.2 Comportamiento de nuestra acción de borrar personalizada

Para marcar como borrada la factura actual sin borrarla realmente, necesitamos añadir una nueva propiedad a Invoice. Llamémosla deleted. Puedes verla en el listado 10.5.

Listado 10.5 Nueva propiedad deleted en Invoice

```
public class Invoice extends CommercialDocument {
    @Hidden // No se mostrará por defecto en las vistas y los tabs
    private boolean deleted;

    public boolean isDeleted() {
        return deleted;
    }

    public void setDeleted(boolean deleted) {
        this.deleted = deleted;
    }

    ...
}
```

Como ves, es una propiedad booleana simple y llana. El único detalle es el uso de la anotación @Hidden. Indica que cuando una vista o lista tabular por defecto sea generada la propiedad deleted no se mostrará; aunque si la pones explícitamente en @View(members=) o @Tab(properties=) sí que se mostrará. Usa esta anotación para marcar aquellas propiedades de uso interno del programador pero que no tienen sentido para el usuario final.

Ahora actualiza el esquema de la base de datos. Despues ejecuta las sentencias SQL del listado 10.6 contra la base de datos. Puedes usar la *Database perspective* del Eclipse para hacerlo (ver la sección 4.7).

Listado 10.6 Poner a false la columna deleted de la tabla CommercialDocument

```
update CommercialDocument
```

```
set deleted = false
```

Recuerda que los datos para la entidades Invoice se almacenan en la tabla CommercialDocument.

Ya estamos preparados para escribir el código real de la acción. Míralo en el listado 10.7.

Listado 10.7 La implementación del método execute() de DeleteInvoiceAction

```
public void execute() throws Exception {
    Invoice invoice = XPersistence.getManager().find(
        Invoice.class,
        getView().getValue("oid")); // Leemos el id de la vista
    invoice.setDeleted(true); // Modificamos el estado de la entidad
    addMessage("object_deleted", "Invoice"); // El mensaje de confirmación de borrado
    getView().clear(); // Borramos la vista
}
```

El efecto visual es el mismo, se ve un mensaje y la vista se borra, pero en este caso hacemos algo de lógica. Buscamos la entidad Invoice asociada con la vista actual y entonces cambiamos el valor de su propiedad deleted. No necesitas hacer nada más, porque OpenXava confirma automáticamente la transacción JPA. Es decir, puedes leer cualquier objeto y modificar su estado en una acción, y cuando la acción finalice los cambios se almacenarán en la base de datos.

Pero hemos dejado algunos cabos sueltos. Si el usuario pulsa en el botón de borrar cuando no hay un objeto seleccionado la instrucción para buscar fallará y un mensaje un tanto técnico e ininteligible se le mostrará a nuestro desamparado usuario. Podemos refinar este caso con un simple “if”. Observa la ligera modificación al método execute() en el listado 10.8.

Listado 10.8 Refina el caso de no tener un objeto en la vista

```
public void execute() throws Exception {
    if (getView().getValue("oid") == null) { // ¿Hay un objeto en la vista?
        addError("no_delete_not_exists");
        return;
    }
    ...
}
```

Si la propiedad oid no tiene valor en la vista significa que la vista no está visualizando un objeto existente en la base de datos, por tanto no hay nada que borrar. En este caso, simplemente muestra un mensaje de error. No necesitas añadir “no_delete_not_exists” a tu archivo de mensajes, porque ya existe en los archivos de mensajes de OpenXava. Echa un vistazo a *OpenXava/i18n/Messages_en.properties* para ver los mensajes predefinidos de OpenXava.

Ahora que ya sabes como escribir tus propias acciones personalizadas, es tiempo de aprender como escribir código genérico.

10.2 Acciones genéricas

El código actual de DeleteInvoiceAction refleja la forma típica de escribir acciones. Es código concreto que accede directamente a entidades concretas para manipularlas.

Pero a veces puedes encontrarte alguna lógica en tu acción susceptible de ser usada y reusada por toda tu aplicación, incluso en todas tus aplicaciones. En este caso, puedes utilizar algunas técnicas para crear código más reutilizable, y así convertir tus acciones personalizadas en acciones genéricas.

Aprendamos estas técnicas para escribir código más genérico en nuestras acciones.

10.2.1 Código genérico con MapFacade

Imagínate que quieras usar tu DeleteInvoiceAction también para pedidos. Es más, imagínate que quieras usarla para cualquier entidad de la aplicación con una propiedad deleted. Es decir, quieres una acción para marcar como borrada, en lugar de borrarla de la base de datos, no solo facturas sino cualquier entidad. En este caso, el código actual de tu acción no es suficiente. Necesitas un código más genérico.

Puedes conseguir una acción más genérica usando la clase de OpenXava llamada MapFacade. MapFacade (del paquete org.openxava.model) te permite manejar el estado de tus entidades usando mapas, esto es conveniente ya que View trabaja con mapas. Además, los mapas son más dinámicos que los objetos y por tanto más apropiados para crear código genérico.

Reescribamos nuestra acción para borrar. Primero, renombremos DeleteInvoiceAction (una acción para borrar objetos de tipo Invoice) como InvoicingDeleteAction (la acción para borrar objetos en la aplicación Invoicing). Esto implica que tienes que cambiar la entrada para la acción en controllers.xml, para cambiar el nombre de la clase. Tal como muestra el listado 10.9.

Listado 10.9 Clase cambiada a InvoicingDeleteAction en controllers.xml

```
<action name="delete" mode="detail" confirm="true"
  class="org.openxava.invoicing.actions.DeleteInvoiceAction"
  class="org.openxava.invoicing.actions.InvoicingDeleteAction"
  image="images/delete.gif"
  keystroke="Control D"/>
```

169 Capítulo 10: Refinar el comportamiento predefinido

Ahora, renombra tu DeleteInvoiceAction como InvoicingDeleteAction y reescribe su código dejándola como en el listado 10.10.

Listado 10.10 InvoicingDeleteAction con código genérico usando MapFacade

```
public class InvoicingDeleteAction extends ViewBaseAction {

    public void execute() throws Exception {
        if (getView()
            .getKeyValuesWithValue() // En lugar de getValue("oid") usamos
            .isEmpty()) // el más genérico getKeyValuesWithValue()
        {
            addError("no_delete_not_exists");
            return;
        }
        Map values = new HashMap(); // Los valores a modificar en la entidad
        values.put("deleted", true); // Asignamos true a la propiedad deleted
        MapFacade.setValues( // Modifica los valores de la entidad indicada
            getModelName(), // Un método de ViewBaseAction
            getView().getKeyValues(), // La clave de la entidad a modificar
            values); // Los valores a cambiar
        resetDescriptionsCache(); // Reinicia los caches para los combos
        addMessage("object_deleted", getModelName());
        getView().clear();
        getView().setEditable(false); // Dejamos la vista como no editable
    }
}
```

Esta acción hace lo mismo que la anterior, pero no tiene ninguna referencia a la entidad Invoice. Por tanto, es genérica, puedes usarla con Order, Author o cualquier otra entidad siempre y cuando tengan un propiedad deleted. El truco está en MapFacade la cual permite modificar una entidad a partir de mapas. Puedes obtener esos mapas directamente de la vista (usando getView().getKeyValues() por ejemplo) o puedes crearlos de una manera genérica, como en el caso del mapa values.

Adicionalmente puedes ver dos pequeñas mejoras sobre la versión antigua. Primero, llamamos a resetDescriptionsCache(), un método de BaseAction. Este método borra el caché usado para los combos. Cuando modificas una entidad, si quieres que los combos reflejen los cambios en la sesión actual has de llamar a este método. Segundo, llamamos a getView().setEditable(false). Esto inhabilita los controles de la vista, para impedir que el usuario rellene datos en la vista. Para crear una nueva entidad el usuario tiene que pulsar el botón “nuevo”.

Ahora tu acción está lista para ser usada por cualquier otra entidad. Podríamos copiar y pegar el controlador Invoice como Order en controllers.xml. De esta forma, nuestra lógica genérica para borrar se usaría para Order. ¡Espera un momento! ¿He dicho “copiar y pegar”? No queremos arder en el fuego eterno del

infierno, ¿verdad? Así que usaremos una forma más automática de insuflar nuestra nueva acción a todos los módulos. Aprendámoslo en la siguiente sección.

10.2.2 Cambiar el controlador por defecto para todos los módulos

Si usas `InvoicingDeleteAction` solo para `Invoice` entonces definirla en el controlador `Invoice` de `controllers.xml` es una buena táctica. Pero, recuerda que hemos mejorado esta acción precisamente para hacerla reutilizable, por tanto reutilicémosla. Vamos a asignar un controlador a todos los módulos de un solo golpe.

El primer paso es cambiar el nombre del controlador de `Invoice` a `Invoicing`. El listado 10.11 muestra el controlador renombrado en `controllers.xml`.

Listado 10.11 El controlador `Invoicing` es el controlador `Invoice` renombrado

```
<controller name="Invoicing">
<controller name="Invoice">

    <extends controller="Typical"/>

    <action name="delete" mode="detail" confirm="true"
        class=
            "org.openxava.invoicing.actions.InvoicingDeleteAction"
        image="images/delete.gif"
        keystroke="Control D">
        <use-object name="xava_view"/>
    </action>

</controller>
```

Como ya sabes, cuando usas el nombre de una entidad, como `Invoice`, como nombre de controlador, ese controlador será usado por defecto en el módulo de esa entidad. Por lo tanto, si cambiamos el nombre del controlador, este controlador no se usará para la entidad. De hecho el controlador `Invoicing` no es usado por ningún módulo, porque no hay ninguna entidad llamada `Invoicing`.

Queremos que el controlador `Invoicing` sea el controlador usado por defecto por todos los módulos de la aplicación. Para hacer esto hemos de modificar el archivo `application.xml` que tienes en la carpeta `xava` de tu aplicación. Dejándolo como el que hay en el listado 10.12.

Listado 10.12 El archivo `application.xml` con `Invoicing` como módulo por defecto

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

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

<application name="Invoicing">

    <!--
```

171 Capítulo 10: Refinar el comportamiento predefinido

```
Se asume un módulo por defecto para cada entidad con el
controlador de <default-module/>
-->

<default-module>
  <controller name="Invoicing" />
</default-module>

</application>
```

De esta forma tan simple todos los módulos de tu aplicación ahora usarán Invoicing en lugar de Typical como controlador por defecto. Trata de ejecutar tu módulo Invoice y verás como la acción se ejecuta al borrar un elemento.

Puedes probar el módulo Order también, pero no funcionará porque no tiene la propiedad deleted. Podríamos añadir la propiedad deleted a Order y funcionaría con nuestro nuevo controlador, pero en vez de “copiar y pegar” la propiedad deleted en todas nuestras entidades, vamos a usar una técnica mejor. Veámoslo en la siguiente sección.

10.2.3 Volvamos un momento al modelo

Tu tarea ahora sería añadir la propiedad deleted a todas las entidades para que la acción InvoiceDeleteAction funcione. Esta es una buena ocasión para usar herencia y así poner el código común en el mismo sitio, en lugar de usar el infame “copiar y pegar”.

Primero quita la propiedad deleted de Invoice como muestra el listado 10.13.

Listado 10.13 Quitamos la propiedad deleted de Invoice

```
public class Invoice extends CommercialDocument {
  ...
  @Hidden
  private boolean deleted;

  public boolean isDeleted() {
    return deleted;
  }

  public void setDeleted(boolean deleted) {
    this.deleted = deleted;
  }
  ...
}
```

Y ahora crea una nueva superclase mapeada llamada Deletable. El listado 10.14 muestra su código.

Listado 10.14 Superclase mapeada Deletable con una propiedad deleted

```
@MappedSuperclass
public class Deletable extends Identifiable {

    @Hidden
    private boolean deleted;

    public boolean isDeleted() {
        return deleted;
    }

    public void setDeleted(boolean deleted) {
        this.deleted = deleted;
    }

}
```

Deletable es una superclase mapeada. Recuerda, una superclase mapeada no es una entidad, es una clase con propiedades, métodos y anotaciones de mapeo para ser usada como superclase para entidades. Deletable extiende de Identifiable, por tanto cualquier entidad que extienda Deletable tendrá las propiedades oid y deleted.

Ahora puedes convertir cualquiera de tus entidades actuales en Deletable, solo has de cambiar Identifiable por Deletable como superclase. El listado 10.15 muestra como hacer esto con CommercialDocument.

Listado 10.15 CommercialDocument ahora extiende de Deletable

```
abstract public class CommercialDocument extends Deletable {
    abstract public class CommercialDocument extends Identifiable {
        ...
    }
}
```

Dado que Invoice y Order son ComercialDocument, ahora puedes usar tu controlador Invocing con la acción InvoiceDeleteAction contra ellos.

Nos queda un sutil detalle. La entidad Order tiene un método @PreRemove para hacer una validación al borrar. Esta validación puede impedir el borrado. Podemos mantener esta validación para nuestro borrado personalizado simplemente sobrescribiendo el método setDeleted() de Order, como muestra el listado 10.16.

Listado 10.16 Llamar explícitamente al método @PreRemove desde setDeleted()

```
public class Order extends CommercialDocument {

    ...

    @PreRemove
    private void validateOnRemove() { // Ahora este método no se ejecuta
        if (invoice != null) { // automáticamente ya que el borrado real no se produce
            throw new IllegalStateException();
        }
    }
}
```

```

        XavaResources.getString(
            "cannot_delete_order_with_invoice"));
    }
}

public void setDeleted(boolean deleted) {
    if (deleted) validateOnRemove(); // Llamamos a la validación explícitamente
    super.setDeleted(deleted);
}

}

```

Con este cambio la validación funciona igual que en el caso de un borrado de verdad, así preservamos el comportamiento original intacto.

10.2.4 Metadatos para un código más genérico

Con tu actual código de Invoice y Order el funcionamiento es bueno. Aunque si tratas de borrar una entidad de cualquier otro módulo, recibirás un feo mensaje de error. La figura 10.3 muestra lo que ocurre cuando intentas borrar un Customer.

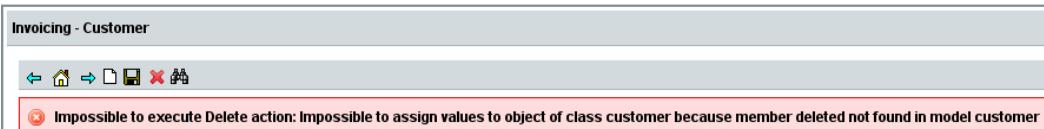


Figura 10.3 Mensaje de error al tratar de borrar una entidad sin propiedad deleted

Sí, si tu entidad no tiene una propiedad deleted, la acción de borrar falla miserablemente. Es verdad que gracias a la clase Deletable puedes añadir la propiedad deleted a todas tus entidades fácilmente, pero puede ser que quieras tener entidades que puedan marcarse como borradas (Deletable) y entidades que sean borradas de verdad de la base de datos. Queremos que la acción funcione bien en todos los casos.

OpenXava almacena metadatos para todas tus entidades, y puedes acceder a estos metadatos desde tu código. Esto te permite, por ejemplo, averiguar si la entidad tiene una propiedad deleted.

El listado 10.17 muestra una modificación en la acción para preguntar si la entidad tiene una propiedad deleted, si no el proceso de borrado no se realiza.

Listado 10.17 Usar metadatos para averiguar si la propiedad 'deleted' existe

```

public void execute() throws Exception {
    if (getView().getKeyValuesWithValue().isEmpty()) {
        addError("no_delete_not_exists");
        return;
    }
}

```

```

if (!getView().getMetaModel() // Metadatos de la entidad actual
    .containsMetaProperty("deleted")) // ¿Tiene una propiedad deleted?
{
    addMessage( // De momento, mostramos un mensaje si la propiedad deleted no está
        "Not deleted, it has no deleted property");
    return;
}

Map values = new HashMap();
values.put("deleted", true);
MapFacade.setValues(
    getModelName(),
    getView().getKeyValues(),
    values);
resetDescriptionsCache();
addMessage("object_deleted", getModelName());
getView().clear();
getView().setEditable(false);
}

```

La clave aquí es `getView().getMetaModel()` que devuelve un objeto `MetaModel` del paquete `org.openxava.model.meta`. Este objeto contiene metadatos sobre la entidad actualmente visualizada en la vista. Puedes preguntar por propiedades, referencias, colecciones, métodos y otra metainformación sobre la entidad. Consulta la API de `MetaModel` para aprender más. En este caso preguntamos si la propiedad `deleted` existe.

De momento solo mostramos un mensaje. Mejorémoslo para borrar de verdad la entidad.

10.2.5 Acciones encadenadas

Queremos que cuando la entidad no tenga una propiedad `deleted` sea borrada de la base de datos de la manera habitual. Nuestra primera opción es escribir nosotros mismos la lógica de borrado, realmente no es una tarea complicada. Sin embargo, es mucho mejor usar la lógica estándar de borrado de OpenXava, así no necesitamos escribir ninguna lógica de borrado y usamos un código más refinado y probado.

Para hacer esto OpenXava provee la posibilidad de encadenar acciones. Es decir, puedes decir que después de tu acción otra acción sea ejecutada. Esto es tan simple como implementar la interfaz `IChainAction` en tu clase. El listado 10.18 muestra `InvoicingDeleteAction` modificada para encadenar con la acción estándar de OpenXava para borrar.

Listado 10.18 InvoicingDeleteAction encadena con la acción de borrar estándar

```

public class InvoicingDeleteAction extends ViewBaseAction
    implements IChainAction { // Encadena con otra acción,
        // indicada en el método getNextAction()

```

```

private String nextAction = null; // Para guardar la siguiente acción a ejecutar

public void execute() throws Exception {
    if (getView().getKeyValuesWithValue().isEmpty()) {
        addError("no_delete_not_exists");
        return;
    }

    if (!getView().getMetaModel()
        .containsMetaProperty("deleted"))
    {
        nextAction = "CRUD.delete"; // "CRUD.delete" se ejecutará cuando esta
        // acción finalice
        return;
    }

    Map values = new HashMap();
    values.put("deleted", true);
    MapFacade.setValues(
        getModelName(),
        getView().getKeyValues(),
        values);
    resetDescriptionsCache();
    addMessage("object_deleted", getModelName());
    getView().clear();
    getView().setEditable(false);
}

public String getNextAction() // Obligatorio por causa de IChainAction
    throws Exception
{
    return nextAction; // Si es nulo no se encadena con ninguna acción
}
}

```

Simplemente devolvemos “CRUD.delete” en `getNextAction()` si queremos que la acción por defecto para borrar de OpenXava se ejecute. Así, escribimos nuestra propia lógica de borrado (en este caso marcar una propiedad con `true`) para algunos casos, y “dejamos pasar” la lógica estándar para los demás.

Ahora puedes usar tu `InvoiceDeleteAction` contra cualquier entidad. Si la entidad tiene una propiedad `deleted` se marcará como borrada, en caso contrario se borrará físicamente de la base de datos.

Este ejemplo te muestra como usar `IChainAction` para refinar la lógica estándar de OpenXava. Otra forma de hacerlo es mediante la herencia. Veamos cómo en la siguiente sección.

10.2.6 Refinar la acción de búsqueda por defecto

`InvoiceDeleteAction` ahora funciona bastante bien, aunque no tiene demasiada utilidad. Es inútil marcar como borrados los objetos, si el resto de la aplicación no es consciente de ello. Es decir, hemos de modificar otras partes de

la aplicación para que traten los objetos “marcados como borrados” como si no existieran.

El lugar más obvio para empezar es la acción de búsqueda. Si borras una factura y después tratas de buscarla, no deberías encontrarla. La figura 10.4 muestra como funciona la búsqueda en OpenXava.

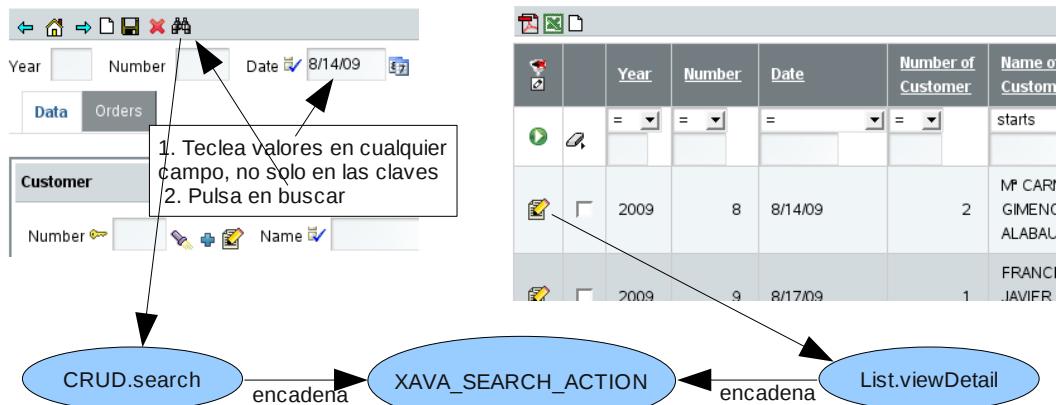


Figura 10.4 Buscar desde detalle y lista encadena con la misma acción

La primera cosa que puedes observar en la figura 10.4 es que buscar en modo detalle es más flexible de lo que parece. El usuario puede introducir cualquier valor en cualquier campo, o combinación de campos, y pulsar en el botón de búsqueda. Entonces el primer objeto cuyos valores coinciden es cargado en la vista.

Puedes pensar. Bueno, puedo refinar la acción `CRUD.search` de la misma forma que he refinado `CRUD.delete`. Por supuesto, puedes hacerlo así. Y funcionaría; cuando el usuario pulsara en la acción del modo detalle tu código se ejecutaría. Aunque, aquí hay un detalle un tanto sutil. La lógica de buscar no se llama solo desde el modo detalle, sino también desde otros puntos del módulo OpenXava. Por ejemplo, cuando el usuario escoge un detalle, la acción `List.viewDetail` coge la clave de la fila, la pone en la vista de detalle, y después ejecuta la acción de buscar.

Para hacerlo bien, hemos de poner la lógica para buscar en un módulo, en la misma acción, y todas las acciones que necesiten buscar encadenarán con esta acción. Tal como muestra la anterior figura 10.4.

Esto queda más claro si ves el código de la acción estándar `CRUD.search`, que es `org.openxava.actions.SearchAction` cuyo código está en el listado 10.19.

Listado 10.19 Acción estándar para buscar en modo detalle (CRUD.search)

```
public class SearchAction extends BaseAction
    implements IChainAction { // Encadena con otra acción

    public void execute() throws Exception { // No hace nada
    }

    public String getNextAction() throws Exception {
        return getEnvironment() // Para acceder a las variables de entorno
            .getValue("XAVA_SEARCH_ACTION");
    }

}
```

Como ves, la acción estándar para buscar en modo lista no hace nada, simplemente redirige a otra acción. Esta otra acción se define en una variable de entorno llamada XAVA_SEARCH_ACTION, que lee usando getEnvironment(). Por lo tanto, si quieras refinar la lógica de búsqueda de OpenXava la mejor manera es definiendo tu acción como valor para XAVA_SEARCH_ACTION. Hagámoslo pues de esta manera.

Para dar valor a la variable de entorno edita el archivo *controllers.xml* en la carpeta *xava* de tu proyecto, y añade al principio la línea <env-var /> como tienes en el listado 10.20.

Listado 10.20 Definición de variables de entorno en controllers.xml

```
...
<controllers>

    <!-- Para definir un valor global para una variable de entorno --&gt;
    &lt;env-var
        name="XAVA_SEARCH_ACTION"
        value="Invoicing.searchExcludingDeleted"/&gt;

    &lt;controller name="Invoicing"&gt;
    ...
</code>
```

De esta forma el valor para la variable de entorno XAVA_SEARCH_ACTION en cualquier módulo será “Invoicing.searchExcludingDeleted”, por lo tanto la lógica de búsqueda para todos los módulos estará en esta acción.

El siguiente paso lógico es definir esta acción en *controllers.xml*, tal como muestra el listado 10.21.

Listado 10.21 Definición de la acción de buscar para Invoicing

```
<controller name="Invoicing">

    ...

    <action name="searchExcludingDeleted"
        hidden="true"
        class="org.openxava.invoicing.actions.SearchExcludingDeletedAction" />
```

```
<!-- hidden="true" : Así la acción no se mostrará en la barra de botones -->
</controller>
```

Y ahora es el momento para escribir la clase de implementación. En este caso solo queremos refinar la lógica de búsqueda, es decir, la búsqueda se ha de hacer de la forma convencional, con la excepción de las entidades con una propiedad `deleted` cuyo valor sea `true`. Para hacer este refinamiento vamos a usar herencia. El listado 10.22 muestra el código de la acción.

Listado 10.22 Búsqueda que excluye los objetos marcados como borrados

```
public class SearchExcludingDeletedAction
    extends SearchByViewKeyAction { // La acción estándar de OpenXava para buscar

    private boolean isDeletable() { // Pregunta si la entidad tiene una propiedad deleted
        return getView().getMetaModel()
            .containsMetaProperty("deleted");
    }

    protected Map getValuesFromView() // Coge los valores visualizados en la vista
        throws Exception // Estos valores se usan como clave al buscar
    {
        if (!isDeletable()) { // Si no es 'deletable' usamos la lógica estándar
            return super.getValuesFromView();
        }
        Map values = super.getValuesFromView();
        values.put("deleted", false); // Llenamos la propiedad deleted con false
        return values;
    }

    protected Map getMemberNames() // Los miembros a leer de la entidad
        throws Exception
    {
        if (!isDeletable()) { // Si no es 'deletable' ejecutamos la lógica estándar
            return super.getMemberNames();
        }
        Map members = super.getMemberNames();
        members.put("deleted", null); // Queremos obtener la propiedad deleted,
        return members; // aunque no esté en la vista
    }

    protected void setValuesToView(Map values) // Asigna los valores desde
        throws Exception // la entidad a la vista
    {
        if (isDeletable() && // Si tiene una propiedad deleted y
            (Boolean) values.get("deleted")) { // vale true
            throw new ObjectNotFoundException(); // lanzamos la misma excepción que
        } // OpenXava lanza cuando el objeto no se encuentra
        else {
            super.setValuesToView(values); // En caso contrario usamos la lógica estándar
        }
    }
}
```

La lógica estándar para buscar está en la clase `SearchByViewKeyAction`.

179 Capítulo 10: Refinar el comportamiento predefinido

Básicamente, la lógica de esta clase consiste en coger los valores de la vista, si la propiedad id está presente buscar por id, en caso contrario coge todos los valores en la vista para usar en la condición de búsqueda, devolviendo el primer objeto que coincida con la condición. Queremos usar este mismo algoritmo cambiando solo algunos detalles sobre la propiedad deleted. Por tanto, en vez de sobrescribir el método `execute()`, que contiene la lógica de búsqueda, sobrescribimos tres métodos protegidos, que son llamados desde `execute()` y contienen algunos puntos susceptibles de ser refinados.

Después de estos cambios prueba tu aplicación, y verás como cuando tratas de buscar una factura o un pedido, si están borrados no se muestran. Incluso si escoges una factura o pedido borrado desde el modo lista se producirá un error y no verás los datos en modo detalle.

Has visto como al definir una variable de entorno `XAVA_SEARCH_ACTION` en `controllers.xml` estableces la lógica de búsqueda de una manera global, es decir, para todos los módulos a la vez. Si lo que quieras es definir una acción de búsqueda para un módulo en particular, simplemente define la variable de entorno en la definición del módulo en `application.xml`, tal como muestra el listado 10.23.

Listado 10.23 Variable de entorno a nivel de módulo en `application.xml`

```
<module name="Product">
    <!--Para dar un valor local a la variable de entorno para este módulo -->
    <env-var
        name="XAVA_SEARCH_ACTION"
        value="Product.searchByNumber"/>
    <model name="Product"/>
    <controller name="Product"/>
    <controller name="Invoicing"/>
</module>
```

De esta forma para el módulo Product la variable de entorno `XAVA_SEARCH_ACTION` valdrá “`Product.searchByNumber`”. Es decir, las variables de entorno son locales a los módulos. Aunque definas un valor por defecto en `controllers.xml`, siempre tienes la opción de sobrescribirlo para un módulo concreto. Las variables de entorno son una forma práctica de configurar tu aplicación declarativamente.

No queremos una forma especial de búsqueda para Product, por tanto no añadas esta definición de módulo a tu `application.xml`. Este código solo era para ilustrar el uso de `<env-var />` en los módulos.

10.3 Modo lista

Ya casi tenemos el trabajo hecho. Cuando el usuario borra una entidad con una

propiedad `deleted` la entidad se marca como borrada en vez de ser borrada físicamente de la base de datos. Y si el usuario trata de buscar una entidad “marcada como borrada” no puede verla en modo detalle. Aunque, el usuario todavía puede ver las entidades “marcadas como borradas” en modo lista, y lo que es peor si borra las entidades desde modo lista, éstas son efectivamente borradas de la base de datos. Atemos estos cabos sueltos.

10.3.1 Filtrar datos tabulares

Solo las entidades con su propiedad `deleted` igual a `false` tienen que ser mostradas en modo lista. Esto es muy fácil de conseguir usando la anotación `@Tab`. Esta anotación te permite definir la forma en que los datos tabulares (los datos mostrados en modo lista) son visualizados, y te permite además definir una condición. Por tanto, añadir esta anotación a las entidades que tengan una propiedad `deleted` es suficiente para conseguir nuestro objetivo, tal como muestra el listado 10.24.

Listado 10.24 Condición en `@Tab` excluye los objetos marcados como borrados

```
@Tab(baseCondition = "deleted = false")
public class Invoice extends CommercialDocument { ... }

@Tab(baseCondition = "deleted = false")
public class Order extends CommercialDocument { ... }
```

Y de esta forma tan sencilla el modo lista no mostrará las entidades “marcadas como borradas”.

10.3.2 Acciones de lista

El único detalle que nos queda es el borrar las entidades desde modo lista, éstas han de marcarse como borradas si procede. Vamos a refinar la acción estándar `CRUD.deleteSelected` de la misma manera que hemos hecho con `CRUD.delete`.

Primero, sobrescribimos la acciones `deleteSelected` y `deleteRow` para nuestra aplicación. Añade la definición de acción del listado 10.25 a tu controlador `Invoicing` definido en `controllers.xml`.

Listado 10.25 Definición de nuestra acción `deleteSelected` en `controllers.xml`

```
<controller name="Invoicing">
    <extends controller="Typical"/>
    ...
    <action name="deleteSelected" mode="list" confirm="true"
        class="org.openava.invoicing.actions.InvoicingDeleteSelectedAction"
        keystroke="Control D"/>

    <action name="deleteRow" mode="NONE" confirm="true"
```

181 Capítulo 10: Refinar el comportamiento predefinido

```
class="org.openxava.invoicing.actions.InvoicingDeleteSelectedAction"
image="images/delete.gif"
in-each-row="true"/>

</controller>
```

Las acciones estándar para borrar entidades desde modo lista son deleteSelected (para borrar las filas seleccionadas) y deleteRow (la acción que aparece en cada fila). Estas acciones están definidas en el controlador CRUD. Typical extiende de CRUD, e Invoicing extiende Typical; así que el controlador Invoicing incluye por defecto estas acciones. Dado que hemos definido unas acciones con los mismos nombres, nuestras acciones sobrescriben las estándares. Es decir, de ahora en adelante la lógica para borrar las filas seleccionadas en modo lista está en la clase InvoicingDeleteSelectedAction. Fíjate como la lógica para ambas acciones están en una única clase Java. El listado 10.26 muestra su código.

Listado 10.26 Acción con lógica propia para borrar entidades desde modo lista

```
public class InvoicingDeleteSelectedAction
    extends TabBaseAction // Para trabajar con datos tabulares (lista) por medio de getTab()
    implements IChainAction { // Encadena con otra acción, indicada con getNextAction()

    private String nextAction = null; // Para almacenar la siguiente acción a ejecutar

    public void execute() throws Exception {
        if (!getMetaModel().containsMetaProperty("deleted")) {
            nextAction="CRUD.deleteSelected"; // "CRUD.deleteSelected" se ejecutará
                                              // cuando esta acción finalice
            return;
        }
        markSelectedEntitiesAsDeleted(); // La lógica para marcar las filas
                                         // seleccionadas como objetos borrados
    }

    private MetaModel getMetaModel() {
        return MetaModel.get(getTab().getModelName());
    }

    public String getNextAction() // Obligatorio por causa de IChainAction
        throws Exception
    {
        return nextAction; // Si es nulo no se encadena con ninguna acción
    }

    private void markSelectedEntitiesAsDeleted() throws Exception {
        ...
    }
}
```

Puedes ver como esta acción es bastante parecida a InvoicingDeleteAction. Si las entidades no tienen la propiedad deleted encadena con la acción estándar, en caso contrario ejecuta su propia lógica para borrar las entidades. Generalmente

las acciones para modo lista extienden de TabBaseAction, así puedes usar getTab() para obtener los objetos Tab asociados a la lista. Un Tab (de org.openxava.tab) te permite manipular los datos tabulares. Por ejemplo en el método getMetaModel() preguntamos al Tab el nombre del modelo para obtener el MetaModel correspondiente.

Si la entidad tiene un propiedad deleted entonces se ejecuta nuestra propia lógica de borrado. Esta lógica está en markSelectedEntitiesAsDeleted() que puedes ver en el listado 10.27.

Listado 10.27 Lógica para marcar como borradas las entidades de modo lista

```
private void markSelectedEntitiesAsDeleted() throws Exception {
    Map values = new HashMap(); // Valores a asignar a cada entidad para marcarla
    values.put("deleted", true); // Pone deleted a true
    for (int row: getSelected()) { // Itera sobre todas las filas seleccionadas
        Map key = (Map) getTab().getTableModel().getObjectType(row);
        try { // seleccionadas. Obtenemos la clave de cada entidad
            MapFacade.setValues( // Modificamos cada entidad
                getTab().getModelName(),
                key,
                values);
        }
        catch (ValidationException ex) { // Si se produce una ValidationException...
            addError("no_delete_row", row + 1, key);
            addErrors(ex.getErrors()); // ...mostramos los mensajes
        }
        catch (Exception ex) { // Si se lanza cualquier otra excepción, se añade
            addError("no_delete_row", row + 1, key); // un mensaje genérico
        }
    }
    getTab().deselectAll(); // Despues de borrar deseleccionamos la filas
    resetDescriptionsCache(); // Y reiniciamos el caché de los combos para este usuario
}
```

Como ves la lógica es un simple bucle sobre las filas seleccionadas, y en cada iteración ponemos a *true* la propiedad deleted usando el método MapFacade.setValues(). Atrapamos las excepciones dentro de la iteración del bucle, así si hay algún problema borrando la entidad, esto no afecta al borrado de las demás entidades. Hemos hecho un pequeño refinamiento para el caso de ValidationException, añadiendo los errores de validación (ex.getErrors()) a los errores a mostrar al usuario.

Al final deseleccionamos todas las filas mediante getTab().deselectAll(), porque estamos borrando filas, por tanto si no eliminamos la selección, esta se habría recorrido después de la ejecución de la acción.

Hemos llamado a resetDescriptionsCache() para actualizar las entidades borradas en todos los combos de la actual sesión de usuario. Los combos, es decir las referencias marcadas con @DescriptionsList, usan el @Tab de la entidad referenciada para filtrar los datos. Es decir, si tuvieras un combo de facturas o

183 Capítulo 10: Refinar el comportamiento predefinido

pedidos con la condición “deleted = false” en el @Tab, en este caso el contenido del combo cambiaría.

Ahora ya tienes refinada del todo la forma en que tu aplicación borra las entidades. Aunque aún nos quedan cosas interesante por hacer.

10.4 Reutilizar el código de las acciones

Ahora tu aplicación marca como borradas las facturas y pedidos en vez de borrarlos. La ventaja de este enfoque es que el usuario puede restaurar en cualquier momento una factura o pedido borrado por error. Para que esta característica sea útil de verdad has de proporcionar al usuario una herramienta para restaurar las entidades borradas. Vamos a crear un módulo papelera para Invoice y otro para Order para traer los documentos borrados de vuelta a la vida.

10.4.1 Propiedades para crear acciones reutilizables

La papelera que queremos es como la que puedes ver en la figura 10.5. Es decir, una lista de facturas o pedidos donde el usuario pueda seleccionar varias y pulsar en el botón 'Restaurar', o simplemente pulsar en el vínculo 'Restaurar' en la fila del documento que quiera restaurar.

La lógica de esta acción de restaurar es simplemente poner la propiedad `deleted` de las entidades seleccionadas a `false`. Es decir, es exactamente la misma lógica que usamos para borrar, pero poniendo `false` en vez de `true`. Dado que nuestra conciencia no nos permite copiar y pegar, vamos a reutilizar nuestro código actual. La forma de reutilizar es añadiendo una propiedad `restore` a la acción `InvoicingDeleteSelectedAction`, para poder restaurar las entidades borradas.

El listado 10.28 muestra el código necesario para añadir una propiedad `restore` a la acción.

Invoicing - Invoice trash					
		Year	Number	Date	Amount
		=		Pulsa para restaurar ...	
Restore	<input checked="" type="checkbox"/>	2009	4	8/10/09	358.44
Restore	<input checked="" type="checkbox"/>	2009			32
1			... o selecciona varias filas y pulsa en el botón		
			Restore		

Figura 10.5 Papelera de facturas

Listado 10.28 Nueva propiedad restore en InvoiceDeleteSelectedAction

```
public class InvoicingDeleteSelectedAction ... {  
    ...  
    private boolean restore; // Una nueva propiedad restore...
```

```

public boolean isRestore() { // ...con su getter
    return restore;
}

public void setRestore(boolean restore) { // ...y su setter
    this.restore = restore;
}

private void markSelectedEntitiesAsDeleted()
throws Exception
{
    Map values = new HashMap();
    values.put("deleted", true); // En lugar de un true fijo, usamos
    values.put("deleted", !isRestore()); // el valor de la propiedad restore
    ...
}

...
}

```

Como puedes ver solo hemos añadido una propiedad `restore`, y el uso de su complemento como nuevo valor para la propiedad `deleted` en la entidad. Es decir, si `restore` es `false`, el caso por defecto, un `true` se grabará en `deleted`, así tu acción de borrar borrará. Pero si `restore` es `true` la acción guardará `false` en la propiedad `deleted` de la entidad, y por tanto la factura, pedido o cualquier otra entidad estará de nuevo disponible en la aplicación.

Para usar esta acción como una acción para restaurar has de definirla en `controllers.xml`, tal como muestra el listado 10.29.

Listado 10.29 Definición de la acción de restaurar en controllers.xml

```

<controller name="Trash">
    <action name="restore" mode="list"
        class="org.openxava.invoicing.actions.InvoicingDeleteSelectedAction">
        <set property="restore" value="true"/> 
    </action>
</controller>

```

A partir de ahora puedes referenciar a la acción `Trash.restore` cuando necesites una acción para restaurar. Estás reutilizando el mismo código para borrar y restaurar, gracias al elemento `<set />` de `<action />` que te permite configurar las propiedades de la acción.

Usemos esta nueva acción de restaurar en los nuevos módulos papelera.

10.4.2 Módulos personalizados

Como ya sabes, OpenXava genera un módulo por defecto para cada entidad de tu aplicación. Aunque, siempre tienes la opción de definir los módulos a mano, bien para refinar el comportamiento del módulo para cierta entidad, o bien para

185 Capítulo 10: Refinar el comportamiento predefinido

definir una funcionalidad completamente nueva sobre esa entidad. En este caso vamos a crear dos nuevos módulos, `InvoiceTrash` y `OrderTrash`, para restaurar los documentos borrados. Usaremos el controlador `Trash` en ellos. El listado 10.30 muestra la definición de módulos en el archivo `application.xml`.

Listado 10.30 Las definiciones de InvoiceTrash y OrderTrash en application.xml

```
<application name="Invoicing">

    <default-module>
        <controller name="Invoicing"/>
    </default-module>

    <module name="InvoiceTrash">
        <env-var name="XAVA_LIST_ACTION"
            value="Trash.restore"/> <!-- La acción a mostrar en cada fila -->
        <model name="Invoice"/>
        <tab name="Deleted"/> <!-- Para mostrar solo las entidades borradas -->
        <controller name="Trash"/> <!-- Con solo una acción: restore -->
        <mode-controller name="ListOnly"/> <!-- Modo lista solo -->
    </module>

    <module name="OrderTrash">
        <env-var name="XAVA_LIST_ACTION" value="Trash.restore"/>
        <model name="Order"/>
        <tab name="Deleted"/>
        <controller name="Trash"/>
        <mode-controller name="ListOnly"/>
    </module>

</application>
```

Estos módulos van contra Invoice y Order, pero son módulos de solo lista, gracias al controlador ListOnly usado como mode-controller. Además, definen una acción especial como acción de fila usando la variable de entorno XAVA_LIST_ACTION. La figura 10.6 muestra InvoiceTrash.

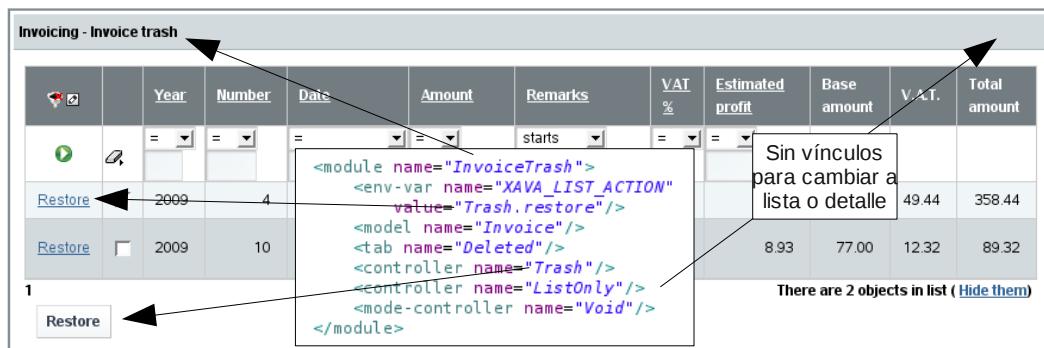


Figura 10.6 InvoiceTrash tiene solo modo lista y una acción de fila especial

10.4.3 Varias definiciones de datos tabulares por entidad

Otro detalle importante es que solo las entidades borradas se muestran en la

lista. Esto es posible porque definimos un @Tab específico indicando su nombre para el módulo. El listado 10.31 lo vuelve a mostrar.

Listado 10.31 Detalle sobre como escoger el @Tab para un módulo

```
<module ... >
  ...
  <tab name="Deleted"/>    <!-- 'Deleted' es un @Tab definido en la entidad -->
  ...
</module>
```

Por supuesto, has de tener un @Tab llamado “Deleted” en tus entidades Order e Invoice. Tal como muestra el listado 10.32.

Listado 10.32 La definición del tab 'Deleted' en Invoice y Order

```
@Tabs({ // @Tabs es para definir varios tabs para la misma entidad
  @Tab(baseCondition = "deleted = false"), // Tab sin nombre, es el de por defecto
  @Tab(name="Deleted", baseCondition = "deleted = true") // Tab con nombre
})
public class Invoice extends CommercialDocument { ... }

@Tabs({
  @Tab(baseCondition = "deleted = false"),
  @Tab(name="Deleted", baseCondition = "deleted = true")
})
public class Order extends CommercialDocument { ... }
```

Se ve como @Tabs permite poner varias definiciones de datos tabulares por entidad. Así, usamos el @Tab sin nombre como lista por defecto para Invoice y Order, pero tenemos un @Tab llamado 'Deleted' que puedes usar para generar una lista con solo las filas borradas. En este caso lo usamos para los módulos papelera.

10.4.4 Obsesión por reutilizar

¡Bien hecho! El código de InvoicingDeleteSelectedAction puede borrar y restaurar entidades, y hemos añadido la capacidad de restaurar con solo un poco más de código, sin copiar y pegar.

Y ahora un enjambre de perniciosos pensamientos bullen en tu cabeza. Seguramente estés pensando “Esta acción no es únicamente para borrar, sino también para borrar y restaurar”, y entonces, “Espera un momento, lo que es en realidad es una acción para actualizar la propiedad deleted de la entidad actual”, y tu siguiente pensamiento será “Con tan solo un poco más podemos actualizar cualquier propiedad de la entidad”.

Sí, estás en lo cierto. Con facilidad podemos crear una acción más genérica, una UpdatePropertyAction por ejemplo, y usarla para declarar tus acciones deleteSelected y restore, tal como muestra el listado 10.33.

Listado 10.33 Acción para actualizar cualquier propiedad de cualquier entidad

```

<action name="deleteSelected" mode="list" confirm="true"
  class="org.openxava.invoicing.actions.UpdatePropertyAction"
  keystroke="Control D">
  <set property="property" value="deleted"/>
  <set property="value" value="true"/>
</action>

<action name="restore" mode="list"
  class="org.openxava.invoicing.actions.UpdatePropertyAction">
  <set property="property" value="deleted"/>
  <set property="value" value="false"/>
</action>

```

Aunque parezca una buena idea, no vamos a crear esta flexible `UpdatePropertyAction`. Porque cuanto más flexible sea tu código, más sofisticado será. Y no queremos código sofisticado. Queremos código sencillo, y aunque el código sencillo es algo imposible de conseguir, hemos de esforzarnos por que nuestro código sea lo más sencillo posible. El consejo es: crea código reutilizable solo cuando éste simplifique tu aplicación en el presente.

10.5 Pruebas JUnit

Hemos refinado la manera en que tu aplicación borra entidades, además hemos añadido dos módulos personalizados, los módulos papelera. Antes de seguir adelante, tenemos que escribir las pruebas de estas nuevas funcionalidades.

10.5.1 Probar el comportamiento personalizado para borrar

No hemos de escribir una prueba para esto, porque el código actual de prueba ya comprueba esta funcionalidad de borrado. Generalmente, cuando cambias la implementación de cierta funcionalidad pero no su uso desde el punto de vista del usuario, como es nuestro caso, no necesitas añadir nuevas pruebas.

Ejecuta todas las prueba de tu aplicación, y ajusta los detalles necesarios para que funcionen bien. Realmente, solo necesitarás cambiar “`CRUD.delete`” por “`Invoicing.delete`” y “`CRUD.deleteSelected`” por “`Invoicing.deleteSelected`” en algunas pruebas. El listado 10.34 resume los cambios que necesitas aplicar a tu código de pruebas.

Listado 10.34 Cambiar “`CRUD.delete`” por “`Invoicing.delete`” en las pruebas

```

// En el archivo CustomerTest.java
public class CustomerTest extends ModuleTestBase {
    ...
    public void testCreateReadUpdateDelete() throws Exception {
        ...
        // Borrar
        execute("CRUD.delete");
    }
}

```

```

        execute("Invoicing.delete");
        assertMessage("Customer deleted successfully");
    }
    ...
}

// En el archivo CommercialDocumentTest.java
abstract public class CommercialDocumentTest extends ModuleTestBase {
    ...
    private void remove() throws Exception {
        execute("CRUD.delete");
        execute("Invoicing.delete");
        assertNoErrors();
    }
    ...
}

// En el archivo ProductTest.java
public class ProductTest extends ModuleTestBase {
    ...
    public void testRemoveFromList() throws Exception {
        ...
        execute("CRUD.deleteSelected");
        execute("Invoicing.deleteSelected");
        ...
    }
    ...
}

// En el archivo OrderTest.java
public class OrderTest extends CommercialDocumentTest {
    ...
    public void testSetInvoice() throws Exception {
        ...
        execute("CRUD.delete");
        execute("Invoicing.delete");
        ...
    }
}
}

```

Después de estos cambios todas tus prueba funcionarán bien, y esto confirma que tus acciones para borrar personalizadas conservan la semántica original. Solo ha cambiado la implementación.

10.5.2 Probar varios módulos en el mismo método de prueba

También has de probar los nuevos módulos personalizados, OrderTrash e InvoiceTrash. De paso, verificaremos que la lógica de borrado funciona bien, y que las entidades son solo marcadas como borradas y no son borradas de verdad.

Para probar el módulo InvoiceTrash seguiremos los siguientes pasos:

- Empezamos en el módulo Invoice.

189 Capítulo 10: Refinar el comportamiento predefinido

- Borramos una factura desde modo detalle y verificamos que ha sido borrada.
- Borramos una factura desde modo lista y verificamos que ha sido borrada.
- Vamos al módulo InvoiceTrash.
- Verificamos que contiene las dos facturas borradas.
- Las restauramos y verificamos que desaparecen de la lista del módulo papelera.
- Volvemos al módulo Invoice.
- Verificamos que las dos facturas restauradas están en la lista.

Puedes observar como empezamos en el módulo Invoice. Además, seguramente te hayas dado cuenta de que la prueba para Order es exactamente igual. Por tanto, en vez de crear dos nuevas clases de prueba, OrderTrashTest e InvoiceTrash, simplemente añadiremos un método de prueba en la ya existente CommercialDocumentTest. Así, reutilizaremos el mismo código para probar OrderTrash, InvoiceTrash y la lógica personalizada de borrado. Este código está en el método testTrash() mostrado en el listado 10.35.

Listado 10.35 El método testTrash() en CommercialDocumentTest

```
public void testTrash() throws Exception {
    assertListOnlyOnePage(); // Sólo una página en la lista, es decir menos de 10 filas
    // Borrar desde modo detalle
    int initialRowCount = getListRowCount();
    String year1 = getValueInList(0, 0);
    String number1 = getValueInList(0, 1);
    execute("Mode.detailAndFirst");
    execute("Invoicing.delete");
    execute("Mode.list");

    assertListRowCount(initialRowCount - 1); // Hay una fila menos
    assertDocumentNotInList(year1, number1); // La entidad borrada no está en lista

    // Borrar desde el modo lista
    String year2 = getValueInList(0, 0);
    String number2 = getValueInList(0, 1);
    checkRow(0);
    execute("Invoicing.deleteSelected");
    assertListRowCount(initialRowCount - 2); // Hay dos filas menos
    assertDocumentNotInList(year2, number2); // La otra entidad borrada
    // no está en la lista
    // Verificar la entidades borradas en el módulo papelera
    changeModule(model + "Trash"); // model puede ser 'Invoice' u 'Order'
    assertListOnlyOnePage();
    int initialTrashRowCount = getListRowCount();

    assertDocumentInList(year1, number1); // Verificamos que las entidades borradas
    assertDocumentInList(year2, number2); // están en la lista del módulo papelera
    // Restaurar usando una acción de fila
```

```

int row1 = getDocumentRowInList(year1, number1);
execute("Trash.restore", "row=" + row1);
assertListRowCount(initialTrashRowCount - 1); // 1 fila menos después de restaurar
assertDocumentNotInList(year1, number1); // La entidad restaurada ya
                                            // no se muestra en la lista del módulo papelera
// Restaurar seleccionando una fila y usando el botón de abajo
int row2 = getDocumentRowInList(year2, number2);
checkRow(row2);
execute("Trash.restore");
assertListRowCount(initialTrashRowCount - 2); // 2 filas menos
assertDocumentNotInList(year2, number2); // La entidad restaurada ya
                                            // no se muestra en la lista del módulo papelera

// Verificar las entidades restauradas
changeModule(model);
assertListRowCount(initialRowCount); // Despues de restaurar tenemos
assertDocumentInList(year1, number1); // las filas originales de nuevo
assertDocumentInList(year2, number2);
}

```

Como ves testTrash() sigue los susodichos pasos. Fíjate como usando el método changeModule() de ModuleTestBase tu prueba puede cambiar a otro módulo. Usamos esto para cambiar al módulo papelera, y volver atrás.

Aquí estamos utilizando algunos métodos auxiliares que has de añadir a CommercialDocumentTest. El primero es assertListOnlyOnePage() que confirma que el modo lista es apropiado para ejecutar esta prueba. El listado 10.36 muestra su código.

Listado 10.36 Método en CommercialDocumentTest verifica el estado de la lista

```

private void assertListOnlyOnePage() throws Exception {
    assertListNotEmpty(); // De ModuleTestBase
    assertTrue("Must be less than 10 rows to run this test",
               getListRowCount() < 10);
}

```

Necesitamos tener menos de 10 filas, porque el método getListRowCount() informa solo de las filas visualizadas, por tanto si tienes más de 10 filas (10 es el número de filas por página por defecto) no puedes aprovechar getListRowCount(), ya que siempre devolvería 10.

Los métodos restantes son para verificar que cierto pedido o factura está (o no está) en la lista. Míralos en el listado 10.37.

Listado 10.37 Verificar existencia de documentos en CommercialDocumentTest

```

private void assertDocumentNotInList(String year, String number)
    throws Exception
{
    assertTrue(
        "Document " + year + "/" + number + " must not be in list",
        getDocumentRowInList(year, number) < 0);
}

```

```
private void assertDocumentInList(String year, String number)
    throws Exception
{
    assertTrue(
        "Document " + year + "/" + number + " must be in list",
        getDocumentRowInList(year, number) >= 0);
}

private int getDocumentRowInList(String year, String number)
    throws Exception
{
    int c = getListRowCount();
    for (int i=0; i<c; i++) {
        if (year.equals(getValueInList(i, 0)) &&
            number.equals(getValueInList(i, 1)))
        {
            return i;
        }
    }
    return -1;
}
```

Puedes ver en `getDocumentRowInList()` como se hace un bucle para buscar valores concretos en una lista.

Ahora puedes ejecutar todas las pruebas de tu aplicación Invoicing. Todo tiene que salir en color verde.

10.6 Resumen

El comportamiento estándar de OpenXava solo es el punto de partida. Usando la acción de borrar como excusa, hemos explorado algunas formas de refinrar los detalles del comportamiento de la aplicación. Con las técnicas de este capítulo no solo puedes refinrar la lógica de borrado, sino también definir completamente la forma en que una aplicación OpenXava funciona. Así, tienes la posibilidad de adaptar el comportamiento de tu aplicación para cubrir las expectativas de tus usuarios.

El comportamiento por defecto de OpenXava es limitado: solo mantenimientos y listados. Si quieres una aplicación que de verdad aporte valor a tu usuario necesitas añadir funcionalidad específica que le ayude a resolver sus problemas. Haremos esto en el próximo capítulo.

*Comportamiento
y
lógica de
negocio*

capítulo 11

OpenXava no es simplemente un marco de trabajo para hacer mantenimientos (altas, bajas, modificaciones y consultas), más bien está concebido para desarrollar aplicaciones de gestión plenamente funcionales. Hasta ahora hemos aprendido como crear y refinar la aplicación para manejar los datos. Ahora vamos a posibilitar al usuario la ejecución de lógica de negocio específica.

En este capítulo vamos a ver como escribir lógica de negocio en el modelo y llamar a esta lógica desde acciones personalizadas. Así podrás transformar tu aplicación de gestión de datos en una herramienta útil para el trabajo cotidiano de tu usuario.

11.1 Lógica de negocio desde el modo detalle

Empezaremos con el caso más simple: un botón en modo detalle para ejecutar cierta lógica. En este caso para crear la factura desde un pedido (figura 11.1).

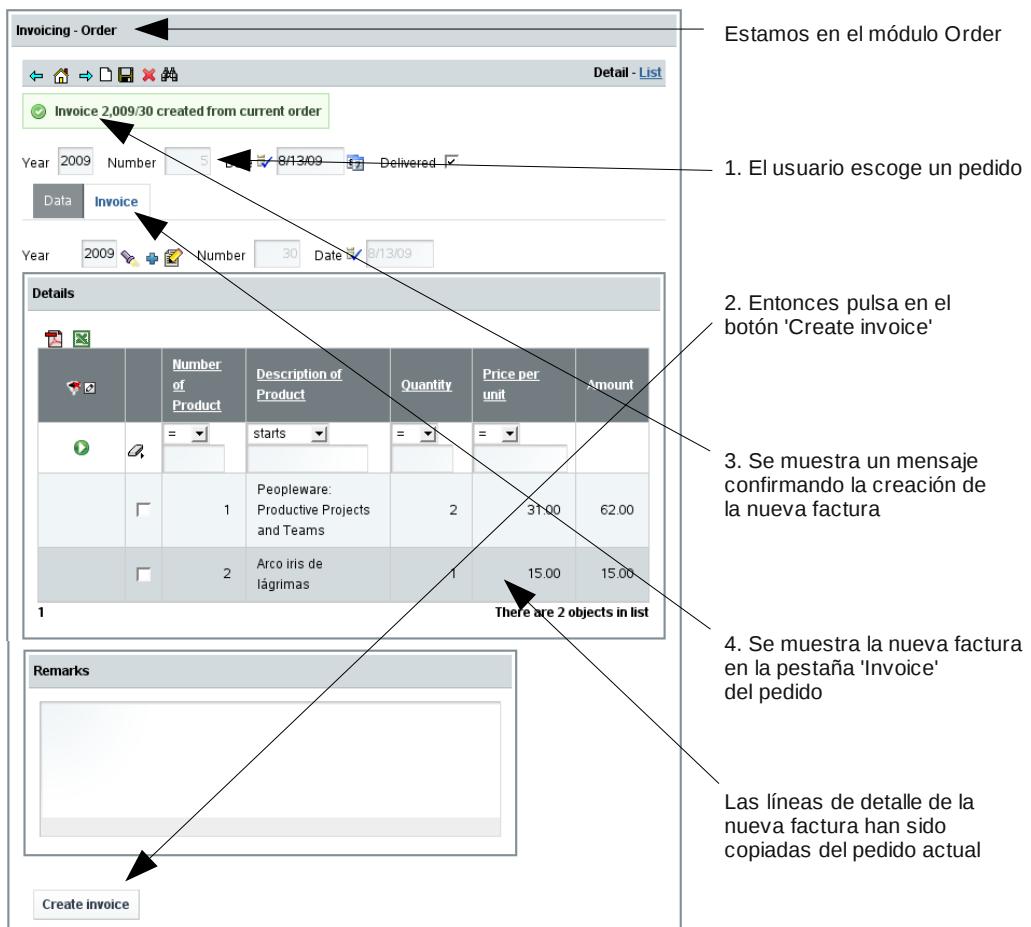


Figura 11.1 Crear una factura desde un pedido usando una acción

La figura 11.1 muestra como esta nueva acción coge el pedido actual y crea una factura a partir de él. Simplemente copia todos los datos del pedido a la nueva factura, incluyendo las líneas de detalle. Se muestra un mensaje y la pestaña 'Factura' del pedido visualizará la factura recién creada. Veamos como codificar este comportamiento.

11.1.1 Crear una acción para ejecutar lógica personalizada

Como ya sabes el primer paso para tener una acción personalizada en tu módulo es definir un controlador con esa acción. Por tanto, editemos *controllers.xml* y añadamos un nuevo controlador. El listado 11.1 muestra el controlador Order.

Listado 11.1 Controlador Order en controllers.xml, con la acción createInvoice

```
<controller name="Order">
    <extends controller="Invoicing"/> <!-- Para tener las acciones estándar -->

    <action name="createInvoice" mode="detail"
        class="org.openxava.invoicing.actions.CreateInvoiceFromOrderAction"/>
    <!-- mode="detail" : Sólo en modo detalle -->

</controller>
```

Dado que hemos seguido la convención de dar al controlador el mismo nombre que a la entidad y el módulo, ya tenemos automáticamente esta nueva acción disponible para Order. El controlador Order desciende del controlador Invoicing. Recuerda que creamos un controlador Invoicing en el capítulo 10. Es un refinamiento del controlador Typical.

Ahora hemos de escribir el código Java para la acción. Puedes verlo en el listado 11.2.

Listado 11.2 Código de la acción para crear una factura desde un pedido

```
package org.openxava.invoicing.actions; // En el paquete 'actions'

import org.openxava.actions.*;
import org.openxava.invoicing.model.*;
import org.openxava.jpa.*;

public class CreateInvoiceFromOrderAction
    extends ViewBaseAction { // Para usar getView()

    public void execute() throws Exception {
        Order order = XPersistence.getManager().find( // Usamos JPA para obtener la
            Order.class, // entidad Order visualizada en la vista
            getView().getValue("oid"));
        order.createInvoice(); // El trabajo de verdad lo delegamos en la entidad
        getView().refresh(); // Para ver la factura creada en la pestaña 'Invoice'
        addMessage("invoice_created_from_order", // Mensaje de confirmación
            order.getInvoice());
    }
}
```

```
}
```

Realmente simple. Buscamos la entidad Order, llamamos al método `createInvoice()`, refrescamos la vista y mostramos un mensaje. Nota como la acción es un mero intermediario entre la vista (la interfaz de usuario) y el modelo (la lógica de negocio).

Recuerda añadir el texto del mensaje en el archivo `Invoicing-messages_en.properties` de la carpeta `i18n`. El listado 11.3 muestra un posible texto.

Listado 11.3 Mensaje de confirmación en `Invoicing-messages_en.properties`

```
invoice_created_from_order=Invoice {0} created from current order
```

Sin embargo, el mensaje tal cual está no se muestra de forma agradable, porque enviamos como argumento un objeto Invoice. Necesitamos un `toString()` para Invoice y Order que sea útil para el usuario. Sobrescribiremos `toString()` de `CommercialDocument` (el padre de Invoice y Order) para conseguirlo. Puedes ver este método `toString()` en el listado 11.4.

Listado 11.4 Método `toString()` de `CommercialDocument`

```
abstract public class CommercialDocument extends Deletable {
    ...
    public String toString() {
        return year + "/" + number;
    }
}
```

El año y el número son perfectos para identificar una factura o pedido desde el punto de vista del usuario.

Esto es todo para la acción. Veamos la pieza restante. El método `createInvoice()` de la entidad Order.

11.1.2 Escribiendo la lógica de negocio real en la entidad

La lógica de negocio para crear una nueva Invoice está en la entidad Order, no en la acción. Esto es la forma natural de hacerlo. El principio esencial de la Orientación a Objetos es que los objetos no son solo datos, sino datos y lógica. El código más bello es aquel cuyos objetos contienen la lógica para manejar sus propios datos. Si tus entidades son meros contenedores de datos (simples envoltorios de las tablas de la base de datos) y tus acciones tienen toda la lógica para manipularlos, en ese caso tu código es una perversión del objetivo original

de la Orientación a Objetos¹⁵.

Aparte de las razones espirituales, poner la lógica para crear una Invoice dentro de Order es un enfoque pragmático, porque de esta forma podemos usar esta lógica desde otras acciones, proceso masivos, servicios web, etc.

Veamos el código. El listado 11.5 muestra el método `createInvoice()` de la clase Order.

Listado 11.5 Método `createInvoice()` en la entidad Order

```
public class Order extends CommercialDocument {
    ...
    public void createInvoice() throws Exception { // throws Exception para tener
                                                // un código más simple, de momento
        Invoice invoice = new Invoice(); // Instancia una factura
        BeanUtils.copyProperties(invoice, this); // y copia el estado del pedido actual
        invoice.setId(null); // Para que JPA sepa que esta entidad todavía no existe
        invoice.setDate(new Date());
        invoice.setDetails(new ArrayList()); // Borra la colección de detalles
        XPersistence.getManager().persist(invoice);
        copyDetailsToInvoice(invoice); // Rellena la colección de detalles
        this.invoice = invoice; // Siempre después de persist()
    }
}
```

La lógica consiste en crear un nuevo objeto Invoice, copiar los datos desde el Order actual a él y asignar la entidad resultante a la referencia invoice del Order actual.

Hay dos sutiles detalles aquí. Primero, has de escribir `invoice.setId(null)`, si no la nueva Invoice tendría la misma identidad que el Order original, además a JPA no le gusta persistir los objetos con el id autogenerado rellenado de antemano. Segundo, has de asignar la nueva Invoice a la actual Order (`this.invoice = invoice`) después de llamar a `persist(invoice)`, si no obtendrás un error de JPA (algo así como “object references an unsaved transient instance”).

11.1.3 Escribe menos código usando Apache Commons BeanUtils

Observa como hemos usado `BeanUtils.copyProperties()` para copiar todas las propiedades del actual Order a la nueva Invoice. Este método copia todas las propiedades con el mismo nombre de un objeto a otro, incluso si los objetos son de clases diferentes. Esta utilidad pertenece al proyecto de apache Commons BeanUtils. El jar para esta utilidad, `commons-beanutils.jar`, ya está incluido en tu proyecto.

¹⁵ Por desgracia muchos de los patrones y buenas prácticas J2EE son perversiones de la Orientación a Objetos

El listado 11.6 muestra como usando BeanUtils escribes menos código.

Listado 11.6 BeansUtil.copyProperties() frente a copiar las propiedades a mano

```
BeanUtils.copyProperties(invoice, this);
// Es lo mismo que
invoice.setOid(getOid());
invoice.setYear(getYear());
invoice.setNumber(getNumber());
invoice.setDate(getDate());
invoice.setDeleted(isDeleted());
invoice.setCustomer(getCustomer());
invoice.setVatPercentage(getVatPercentage());
invoice.setAmount(getAmount());
invoice.setRemarks(getRemarks());
invoice.setDetails(getDetails());
```

Sin embargo, la principal ventaja de usar BeanUtils no es ahorrar tiempo de teclado, sino que obtienes un código más resistente a los cambios. Porque, si añades, quitas o renombras alguna propiedad de ComercialDocument (el padre de Invoice y Order), si estás copiando las propiedades a mano tienes que cambiar el código, mientras que si estás usando BeanUtils.copyProperties() el código funcionará siempre bien, sin tener que cambiarlo.

11.1.4 Copiar una colección de entidad a entidad

La nueva Invoice tiene que tener las mismas líneas de detalle que el Order. Realmente, no la misma colección sino una copia. No podemos asignar la colección tal como muestra el listado 11.7.

Listado 11.7 Manera incorrecta de copiar una colección de una entidad a otra

```
invoice.setDetails(getDetails()); // Esto no funciona
```

Esto no funciona porque una misma colección uno-a-muchos no se puede asignar a dos entidades al mismo tiempo, por tanto hemos de hacer una copia. Nota como en el método createInvoice() (listado 11.4) usamos invoice.setDetails(new ArrayList()) para reiniciar la colección. Esto es porque BeanUtils.copyProperties() ha copiado la colección details de Order. De hecho, copia todo aquello que tenga *getter* y *setter*.

El listado 11.8 muestra el método copyDetailsToInvoice() que copia la colección details de Order a Invoice.

Listado 11.8 Método copyDetailsToInvoice() en la entidad Order

```
private void copyDetailsToInvoice(Invoice invoice) throws Exception {
    for (Detail orderDetail: getDetails()) { // Itera por los detalles del pedido actual
        Detail invoiceDetail = (Detail) // Clona el detalle (1)
            BeanUtils.cloneBean(orderDetail);
        invoiceDetail.setOid(null); // Para ser grabada como una nueva entidad (2)
        invoiceDetail.setParent(invoice); // El punto clave: poner un nuevo parent (3)
```

```

        Xpersistence.getManager().persist(invoiceDetail); // (4)
    }
}

```

Esta es la forma más simple de clonar una colección, simplemente clonando cada elemento (1) y asignándole un nuevo parente (3). También has de quitarle su identidad (2) y marcarlo como persistente (4).

Para clonar el bean usamos BeanUtils otra vez, en este caso el método `cloneBean()`. Este método crea una nueva instancia del mismo tipo que el argumento, y después copia todas las propiedades del objeto fuente en el objeto recién creado.

11.1.5 Excepciones de aplicación

Recuerda la frase: “La excepción que confirma la regla”. Las reglas, la vida, y el software están llenos de excepciones. Y nuestro método `createInvoice()` no es una excepción. Hemos escrito código que funciona en los casos más comunes. Pero, ¿qué ocurre si el pedido no está listo para ser facturado, o si hay algún problema para acceder a la base de datos? Obviamente, en este caso necesitamos tomar caminos diferentes.

Es decir, el simple `throws Exception` que hemos escrito para el método `createInvoice()` no es suficiente para un comportamiento refinado. El listado 11.9 es una versión mejorada del método, usando excepciones.

Listado 11.9 El método `createInvoice()` manejando casos excepcionales

```

public void createInvoice()
    throws ValidationException // Una excepción de aplicación (1)
{
    if (this.invoice != null) { // Si ya tiene una factura no podemos crearla
        throw new ValidationException( // Admite un id de 18n como argumento
            "impossible_create_invoice_order_already_has_one");
    }
    if (!isDelivered()) { // Si el pedido no está entregado no podemos crear la factura
        throw new ValidationException(
            "impossible_create_invoice_order_is_not_delivered");
    }
    try {
        Invoice invoice = new Invoice();
        BeanUtils.copyProperties(invoice, this);
        invoice.setOid(null);
        invoice.setDate(new Date());
        invoice.setDetails(new ArrayList());
        XPersistence.getManager().persist(invoice);
        copyDetailsToInvoice(invoice);
        this.invoice = invoice;
    }
    catch (Exception ex) { // Cualquier excepción inesperada (2)
        throw new SystemException( // Se lanza una excepción runtime (3)
            "impossible_create_invoice", ex);
    }
}

```

```
    }
}
```

Ahora declaramos explícitamente las excepciones de aplicación que este método lanza (1). Una excepción de aplicación es una excepción chequeada que indica un comportamiento especial pero esperado del método. Una excepción de aplicación está relacionada con la lógica de negocio del método. Puedes crear una excepción de aplicación para cada posible caso. Por ejemplo, podrías crear una `OrderAlreadyHasInvoiceException` y una `InvoiceNotDeliveryException`. Esto te permitiría tratar cada caso de forma diferente desde el código que usa el método. Aunque, esto no es necesario en nuestro caso, por tanto nosotros simplemente usamos `ValidationException`, una excepción de aplicación genérica incluida con OpenXava.

También hemos de enfrentarnos a problemas inesperados (2). Los problemas inesperados incluyen errores del sistema (acceso a base de datos, la red o problemas de hardware) o errores de programación (`NullPointerException`, `IndexOutOfBoundsException`, etc). Cuando nos encontramos con cualquier problema inesperado lanzamos una `RuntimeException`. En este caso hemos lanzado una `SystemException`, una `RuntimeException` incluida en OpenXava por comodidad, pero puedes lanzar la `RuntimeException` que quieras.

No necesitas modificar el código de la acción. Si tu acción no atrapa las excepciones, OpenXava lo hace automáticamente. Muestra los mensajes de las `ValidationExceptions` al usuario; y para las excepciones *runtime*, muestra un mensaje de error genérico y aborta la transacción.

Para rematar, añadimos el mensaje para la excepción en los archivos i18n. Edita el archivo `Invoicing-messages_en.properties` de la carpeta `Invoicing/i18n` añadiendo las entradas del listado 11.10.

Listado 11.10 Mensajes usados por las excepciones

```
impossible_create_invoice_order_already_has_one=Impossible to create invoice:  
the order already has an invoice  
impossible_create_invoice_order_is_not_delivered=Impossible to create invoice:  
the order is not delivered yet  
impossible_create_invoice=Impossible to create invoice
```

Hay cierto debate en la comunidad de desarrolladores sobre la manera correcta de usar las excepciones en Java. El enfoque usado en esta sección es la forma clásica de trabajar con excepciones en el mundo J2EE.

11.1.6 Validar desde la acción

Usualmente el mejor lugar para las validaciones es el modelo, es decir, las

201 Capítulo 11: Comportamiento y lógica de negocio

entidades. Sin embargo, a veces es necesario poner lógica de validación en las acciones. Por ejemplo, si quieras preguntar por el estado actual de la interfaz gráfica has de hacer la validación en la acción.

En nuestro caso si el usuario pulsa en “Crear factura” cuando está creando un nuevo pedido que todavía no ha grabado, fallará. Falla porque es imposible crear una factura desde un pedido inexistente. El usuario ha de grabar el pedido primero.

Modificamos el método `execute()` de `CreateInvoiceFromOrderAction` para validar que la factura visualizada actualmente esté grabada (listado 11.11).

Listado 11.11 Validación desde la acción para preguntar por el estado de la vista

```
public void execute() throws Exception {
    Object oid = getView().getValue("oid");
    if (oid == null) { // Si el oid es nulo el pedido actual no se ha grabado todavía
        addError(
            "impossible_create_invoice_order_not_exist");
        return;
    }
    MapFacade.setValues("Order", // Si el pedido existe lo grabamos (2)
        getView().getKeyValues(), getView().getValues());
    Order order = getManager().find(
        Order.class, oid);
    order.createInvoice();
    getView().refresh();
    addMessage("invoice_created_from_order",
        order.getInvoice());
}
```

La validación consiste en verificar que el `oid` es nulo (1), en cuyo caso el usuario está introduciendo un pedido nuevo, pero todavía no lo ha grabado. En este caso se muestra un mensaje y se aborta la creación de la factura. Si el pedido ya existe grabamos los datos desde la interfaz de usuario a la base de datos usando `MapFacade` (2). Es importante tener la base de datos sincronizada con la vista antes de llamar al método de la entidad para crear la factura. Imagina que el usuario marca el pedido como entregado (*delivered*) y después pulsa en “Create invoice”. En este caso obtendría un mensaje de error “Pedido no entregado”. Esto puede ser confuso, por tanto grabar la entidad automáticamente antes de llamar a un método de la entidad es buena idea. Fíjate como `MapFacade` es una herramienta muy útil para mover datos de la interfaz de usuario al modelo.

Aquí también tenemos un mensaje para añadir al archivo `i18n`. Edita el archivo `Invoicing-messages_en.properties` de la carpeta `Invoicing/i18n` añadiendo la entrada mostrada en el listado 11.12.

Listado 11.12 Mensaje usado por la validación de la acción

```
impossible_create_invoice_order_not_exist=Impossible to create invoice: the
order does not exist yet
```

Las validaciones le dicen al usuario que ha hecho algo mal. Esto es necesario, por supuesto, pero es mejor aún crear una aplicación que ayude al usuario a evitar hacer las cosas mal. Veamos una forma de hacerlo en la siguiente sección.

11.1.7 Evento OnChange para ocultar/mostrar una acción por código

Nuestro actual código es suficientemente robusto como para prevenir que equivocaciones del usuario estropeen los datos. Vamos a ir un paso más allá, impidiendo que el usuario se equivoque. Ocultaremos la acción para crear una nueva factura cuando el pedido no esté listo para ello.

OpenXava permite ocultar y mostrar acciones automáticamente. También permite ejecutar una acción cuando cierta propiedad sea cambiada por el usuario en la interfaz de usuario. Con estos dos ingredientes podemos mostrar el botón sólo cuando la acción esté lista para ser usada.

Recuerda que una factura puede ser generada desde un pedido si el pedido ha sido entregado y no tiene factura todavía. Por tanto, tenemos que vigilar los cambios en la referencia invoice y la propiedad delivered de la entidad Order. Haremos esto usando la anotación @OnChange como se muestra en el listado 11.13.

Listado 11.13 @OnChange añadido a invoice y delivered en Order

```
public class Order extends CommercialDocument {
    @ManyToOne
    @ReferenceView("NoCustomerNoOrders")
    @OnChange(ShowHideCreateInvoiceAction.class)
    private Invoice invoice;

    @OnChange(ShowHideCreateInvoiceAction.class)
    private boolean delivered;
    ...
}
```

Con el código de arriba cuando el usuario cambia el valor de delivered o invoice en la pantalla, la acción ShowHideCreateInvoiceAction se ejecutará. Observa el código de la acción en el listado 11.14.

Listado 11.14 Acción para mostrar/ocultar la acción createInvoice dinámicamente

```
package org.openxava.invoicing.actions; // En el paquete 'actions'

import org.openxava.actions.*; // Necesario para usar OnChangePropertyAction,
                           // IShowActionAction and IHideActionAction
public class ShowHideCreateInvoiceAction
    extends OnChangePropertyBaseAction // Necesario para acciones @OnChange (1)
    implements IShowActionAction, // Para mostrar una acción
              IHideActionAction { // Para ocultar una acción}
```

```

private boolean show; // Si true la acción 'Order.createInvoice' se mostrará

public void execute() throws Exception {
    show = isOrderCreated() // Establecemos el valor de 'show'. Este valor
        && isDelivered() // se usará en los métodos de abajo:
        && !hasInvoice(); // getActionToShow() y getActionToHide() (2)
}

private boolean isOrderCreated() {
    return getView().getValue("oid") != null; // Leemos el valor desde la vista
}

private boolean isDelivered() {
    Boolean delivered = (Boolean)
        getView().getValue("delivered"); // Leemos el valor desde la vista
    return delivered == null?false:delivered;
}

private boolean hasInvoice() {
    return getView().getValue("invoice.oid") != null; // Leemos el valor
} // desde la vista

public String getActionToShow() { // Obligatorio por causa de IShowActionAction
    return show?"Order.createInvoice":""; // La acción a mostrar (3)
}

public String getActionToHide() { // Obligatorio por causa de IHideActionAction
    return !show?"Order.createInvoice":""; // La acción a ocultar (3)
}
}

```

Ésta es una acción convencional con un método `execute()`, aunque extiende de `OnChangePropertyBaseAction` (1). Todas las acciones anotadas con `@OnChange` tienen que implementar `IOnChangePropertyAction`, aunque es más fácil extender de `OnChangePropertyBaseAction` la cual lo implementa. Desde esta acción puedes usar `getNewValue()` y `getChangedProperty()`, aunque en este caso concreto no los necesitamos.

El método `execute()` pone a `true` el campo `show` si la orden visualizada está grabada, entregada y no tiene factura (2). Este campo `show` se usa en los métodos `getActionToShow()` y `getActionToHide()`. Estos métodos indican el nombre calificado de la acción a ocultar o mostrar (3). Así, ocultamos o mostramos la acción `Order.createInvoice`, mostrándola solo cuando proceda.

Ahora puedes probar el módulo `Order`. Verás como cuando marcas o desmarcas la casilla entregado (`delivered`) o escoges una factura, el botón para la acción se muestra u oculta. También, cuando el usuario pulsa en 'Nuevo' para crear un nuevo pedido el botón para crear la factura se oculta. Sin embargo, al editar un pedido ya existente, el botón estará siempre presente, aunque el pedido no cumpla los requisitos. Esto es porque cuando un objeto se busca y visualiza las

acciones @OnChange no se ejecutan por defecto. Podemos cambiar esto con una pequeña modificación en SearchExcludingDeleteAction. Míralo en el listado 11.15.

Listado 11.15 La acción de búsqueda extiende SearchExecutingOnChangeAction

```
public class SearchExcludingDeletedAction
    extends SearchByViewKeyAction {
    extends SearchExecutingOnChangeAction { // Usa ésta como clase base}
```

La acción de búsqueda por defecto, es decir, SearchByViewKeyAction no ejecuta las acciones @OnChange por defecto, por tanto cambiamos nuestra acción de buscar para que extienda de SearchExecutingOnChangeAction. SearchExecutingOnChangeAction se comporta exactamente igual que SearchByViewKeyAction pero ejecutando los eventos *OnChange*. De esta forma cuando el usuario escoge un pedido la acción ShowHideCreateInvoiceAction se ejecuta.

Nos queda un pequeño detalle para que todo esto sea perfecto: cuando el usuario pulsa en 'Crear factura' después de que la factura se haya creado el botón se tiene que ocultar. El usuario no puede crear la factura otra vez. Podemos implementar esta funcionalidad con un ligero refinamiento de CreateInvoiceFromOrderAction. Veámoslo en el listado 11.16.

Listado 11.16 CreateInvoiceFromOrderAction se oculta a sí misma

```
public class CreateInvoiceFromOrderAction extends ViewBaseAction
    implements IHideActionAction // Para ocultar la acción
{
    private boolean hideAction = false; // Para indicar si la acción se ocultará

    public void execute() throws Exception {
        ...
        addMessage("invoice_created_from_order",
            order.getInvoice());
        hideAction = true; // Todo ha funciona a la perfección, así que ocultamos la acción
    }

    public String getActionToHide() { // La acción a ocultar, en este caso ella misma
        return hideAction?"Order.createInvoice":null;
    }
}
```

Como puedes ver la acción implementa IHideActionAction para ocultarse a sí misma.

Mostrar y ocultar acciones no es un sustituto para la validación en el modelo. Las validaciones siguen siendo necesarias porque las entidades pueden ser usadas desde cualquier otra parte de la aplicación, no solo de los módulos de mantenimiento. Sin embargo, el truco de ocultar y mostrar acciones mejora la

experiencia del usuario.

11.2 Lógica de negocio desde el modo lista

En el capítulo 10 (sección 10.3) aprendiste como crear acciones de lista. Las acciones de lista son una herramienta utilísima para dar al usuario la posibilidad de aplicar lógica a varios objetos a la vez. En nuestro caso, podemos añadir una acción en el modo lista para crear una nueva factura automáticamente a partir de varios pedidos seleccionados en la lista. La figura 11.2 muestra la forma en que queremos que esta acción funcione.

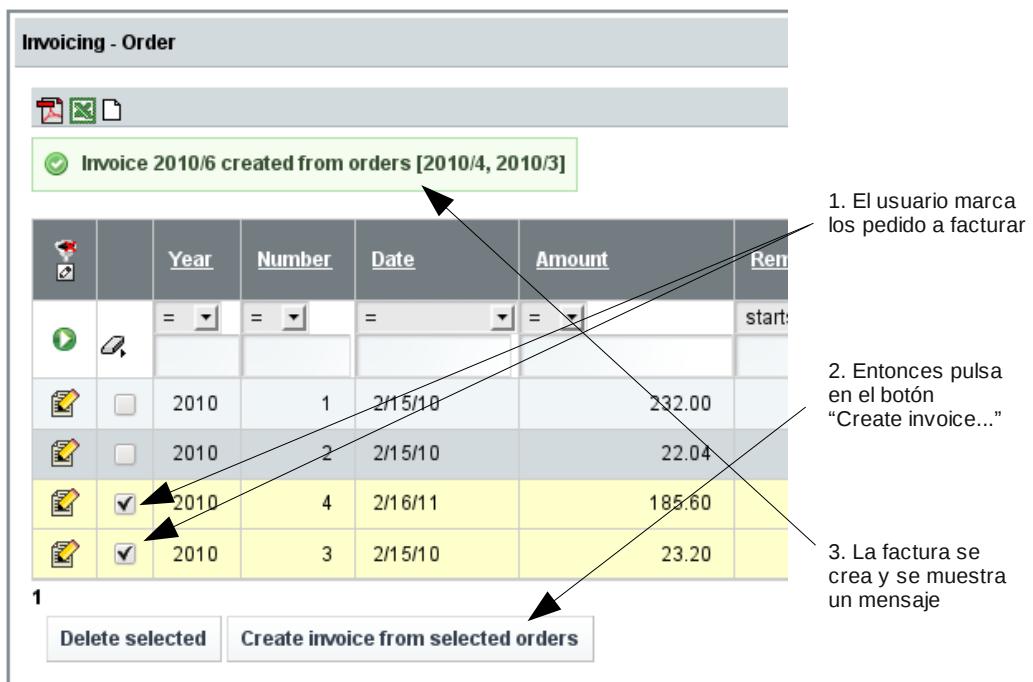


Figura 11.2 Crear una factura desde varios pedidos usando una acción de lista

La figura 11.2 muestra como esta acción de lista coge los pedidos seleccionados y crea una factura a partir de ellos. Simplemente copia los datos del pedido en la nueva factura, añadiendo las líneas de detalle de todos los pedidos en una única factura. También se muestra un mensaje. Veamos como codificar este comportamiento.

11.2.1 Acción de lista con lógica propia

Como ya sabes, el primer paso para tener una acción propia en tu módulo es añadirla a un controlador. Por tanto, editemos `controllers.xml` añadiendo una nueva acción al controlador Order. El listado 11.17 muestra el controlador Order

modificado.

Listado 11.17 Controlador Order con la acción createInvoiceFromSelectedOrders

```
<controller name="Order">
    <extends controller="Invoicing"/>

    <action name="createInvoice" mode="detail"
        class=
        "org.openxava.invoicing.actions.CreateInvoiceFromOrderAction">
        <use-object name="xava_view"/>
    </action>

    <!-- La nueva acción -->
    <action name="createInvoiceFromSelectedOrders"
        mode="list"
        class=
        "org.openxava.invoicing.actions.CreateInvoiceFromSelectedOrdersAction"
    />
    <!-- mode="list" Solo mostrada en modo lista -->

</controller>
```

Solo con esto ya tienes una nueva acción disponible para Order en modo lista.

Ahora hemos de escribir el código Java para la acción. Míralo en el listado 11.18.

Listado 11.18 Acción de lista que crea una factura a partir de varios pedidos

```
public class CreateInvoiceFromSelectedOrdersAction
    extends TabBaseAction // Típico para acciones de lista. Permite usar getTab() (1)
{
    public void execute() throws Exception {
        Collection<Order> orders = getSelectedOrders(); // (2)
        Invoice invoice = Invoice.createFromOrders(orders); // (3)
        addMessage( // (4)
            "invoice_created_from_orders", invoice, orders);
    }

    private Collection<Order> getSelectedOrders() // (5)
        throws FinderException
    {
        Collection<Order> result = new ArrayList<Order>();
        for (Map key: getTab().getSelectedKeys()) { // (6)
            Order order = (Order)
                MapFacade.findEntity("Order", key); // (7)
            result.add(order);
        }
        return result;
    }
}
```

Realmente sencillo. Obtenemos la lista de los pedido marcados en la lista (2), llamamos al método estático `createFromOrders()` (3) de `Invoice` y mostramos un mensaje (4). En este caso también ponemos la lógica real en la clase del

modelo, no en la acción. Dado que la lógica aplica a varios pedidos y crea una nueva factura, el lugar natural para ponerlo es en un método estático de la clase Invoice.

El método `getSelectedOrders()` (5) devuelve una colección con las entidades Order marcadas por el usuario en la lista. Para hacerlo, el método usa `getTab()` (6), disponible en `TabBaseAction` (1), que devuelve un objeto `org.openxava.tab.Tab`. El objeto Tab te permite manejar los datos tabulares de la lista. En este caso usamos `getSelectedKeys()` (6) que devuelve una colección con las claves de las filas seleccionadas. Dado que esas claves están en formato Map usamos `MapFacade.findEntity()` (7) para convertirlas en entidades Order.

Acuérdate de añadir el texto del mensaje al fichero *Invoicing-messages_en.properties* en la carpeta *i18n*. El listado 11.19 muestra un posible texto.

Listado 11.19 Mensaje de confirmación en *Invoicing-messages_en.properties*

```
invoice_created_from_orders=Invoice {0} created from orders: {1}
```

Eso es todo para la acción. Veamos la pieza que falta, el método `createFromOrders()` de la entidad Invoice.

11.2.2 Lógica de negocio en el modelo sobre varias entidades

La lógica de negocio para crear una nueva Invoice a partir de varias entidades Order está en la capa del modelo, es decir, en las entidades, no en la acción. No podemos poner el método en la clase Order, porque el proceso se hace a partir de varios Orders, no de uno. No podemos usar un método de instancia en Invoice porque todavía no existe el objeto Invoice, de hecho lo que queremos es crearlo. Por lo tanto, vamos a crear un método de factoría estático en la clase Invoice para crear una nueva Invoice a partir de varios Orders. Puedes ver este método en el listado 11.20.

Listado 11.20 Método `createFromOrders()` en entidad Invoice

```
public class Invoice extends CommercialDocument {
    ...
    public static Invoice createFromOrders(Collection<Order> orders)
        throws ValidationException
    {
        Invoice invoice = null;
        for (Order order: orders) {
            if (invoice == null) { // La primera vez, el primer pedido
                order.createInvoice(); // Reutilizamos la lógica para
                // crear una factura a partir de un pedido
            }
        }
    }
}
```

```

        invoice = order.getInvoice(); // y cogemos la factura recién creada
    }
    else { // Para el resto de los pedidos la factura ya está creada
        order.setInvoice(invoice); // Asigna la factura
        order.copyDetailsToInvoice(invoice); // Copia la línea. El método
        } // copyDetailsToInvoice es privado en Order.
        // por tanto tenemos que cambiarlo a público
    }
    if (invoice == null) { // Si no hay pedidos
        throw new ValidationException(
            "impossible_create_invoice_orders_not_specified");
    }
    return invoice;
}
}

```

Usamos el primer Order para crear una nueva Invoice usando el método ya existente `createInvoice()` de Order. Entonces copiamos las líneas de los Orders restantes a la nueva Invoice. Además, asignamos la nueva Invoice como la Invoice de los Orders de la colección.

Si invoice es nulo al final del proceso, es porque la colección orders está vacía. En este caso lanzamos una `ValidationException`, ya que la acción no atrapa las excepciones, OpenXava muestra el mensaje de la `ValidationException` al usuario. Esto está bien. Si el usuario no marca los pedido y pulsa en el botón para crear la factura, le aparecerá este mensaje de error.

Usamos el método `copyDetailsToInvoice()` de Order. Este método era privado, por tanto necesitamos cambiarlo a público para poder usarlo desde Invoice. Observa el cambio en el listado 11.21.

Listado 11.21 Refinamientos en `copyDetailsToInvoice()` de Order

```

public class Order extends CommercialDocument {
    ...
    public private // public en vez de private
        void copyDetailsToInvoice(Invoice invoice)
        throws Exception // throws Exception se quita. Ahora se lanza una excepción runtime
    {
        try { // Envolvemos todo el código del método con un try/catch
            for (Detail orderDetail: getDetails()) {
                Detail invoiceDetail = (Detail)
                    BeanUtils.cloneBean(orderDetail);
                invoiceDetail.setOid(null);
                invoiceDetail.setParent(invoice);
                XPersistence.getManager()
                    .persist(invoiceDetail);
            }
        }
        catch (Exception ex) { // Así convertimos cualquier excepción
            throw new SystemException( // en una excepción runtime

```

```
        "impossible_copy_details_to_invoice", ex);
    }
}
```

Además de cambiar 'private' por 'public' envolvemos cualquier excepción en una excepción *runtime*, de esta manera observamos la ya mencionada convención de usar excepciones *runtime* para los problemas inesperados.

Acuérdate de añadir los textos para los mensajes en el archivo *Invoicing-messages_en.properties* de la carpeta *i18n*. El listado 11.22 muestra unos textos posibles.

Listado 11.22 Error de validación en Invoicing-messages_en.properties

```
impossible_create_invoice_orders_not_specified=Impossible to create invoice:  
orders not specified  
impossible_copy_details_to_invoice=Impossible to copy details from order to  
invoice
```

Este no es el único error con el que el usuario puede encontrarse. Todas las validaciones que hemos escrito para Invoice y Order hasta ahora se aplican automáticamente, por lo tanto el usuario ha de escoger pedidos ya entregados y sin factura. La validación del modelo impide que el usuario cree una factura desde pedidos no apropiados.

11.3 Cambiar de módulo

Sería útil para el usuario que después de crear la factura a partir de varios pedidos, pudiera ver y editar la factura recién creada. Una forma de conseguir este comportamiento es creando un módulo sólo para editar una factura, es decir sin modo lista y sin las típicas acciones CRUD. De esta forma podemos cambiar a este módulo después de crear la factura para editarla. La figura 11.3 muestra el comportamiento deseado.

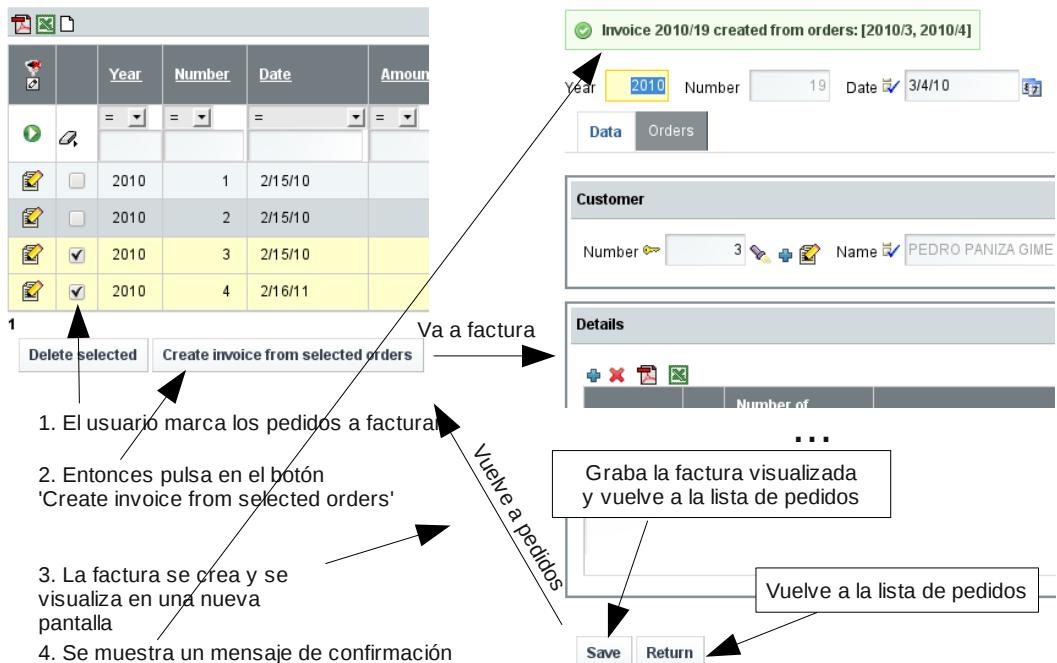


Figura 11.3 Editar la factura después de crearla a partir de varios pedidos

Veamos como implementar este comportamiento.

11.3.1 Uso de IChangeModuleAction

El primer paso es modificar `CreateInvoiceFromSelectedOrdersAction` para cambiar a otro módulo después de su ejecución. El listado 11.23 muestra la modificación.

Listado 11.23 Modificación en acción para cambiar a un nuevo módulo

```
public class CreateInvoiceFromSelectedOrdersAction
    extends TabBaseAction
    implements IChangeModuleAction // Para cambiar a otro módulo después de la ejecución
{
    public String getNextModule() {
        return "CurrentInvoiceEdition"; // Nombre de módulo como está definido en
                                         // application.xml
    }

    public boolean hasReinitNextModule() {
        return true; // Así el módulo se inicializa cada vez que cambiamos a él
    }

    ...
}
```

Como puedes ver, solo has de implementar `IChangeModuleAction`. Esto te

211 Capítulo 11: Comportamiento y lógica de negocio

obliga a añadir los métodos `getNextModule()` que devuelve el nombre del módulo tal como está definido en `application.xml`, y `hasReinitNextModule()`. Devolvemos `true` de `hasReinitNextModule()` porque escribiremos una acción `on-init` (acción ejecutada cuando el módulo se inicializa) en el módulo `CurrentInvoiceEdition` para cargar la factura correcta en la vista, por tanto necesitamos iniciar el módulo cada vez que cambiamos a él.

Obviamente, esto no funcionará hasta que tengamos el módulo `CurrentInvoiceEdition` definido. Haremos esto en la siguiente sección.

11.3.2 Módulo de solo detalle

El objetivo del módulo `CurrentInvoiceEdition` es visualizar una única factura y dar la opción de editarla.

Para definirlo edita el archivo `application.xml` y añade la definición de módulo del listado 11.24

Listado 11.24 Módulo de solo detalle para editar una factura en `application.xml`

```
<module name="CurrentInvoiceEdition">
    <model name="Invoice"/>
    <controller name="CurrentInvoiceEdition"/>
    <mode-controller name="Void"/> <!-- Así el módulo tiene sólo modo de detalle -->
</module>
```

Dado que este módulo es para editar una `Invoice` particular, no tiene modo lista, sino sólo modo detalle. Usamos `Void` como `mode-controller` para conseguirlo.

Este módulo sólo permite al usuario cambiar la `Invoice`, grabar los cambios o volver al módulo original. Para hacerlo define un controlador con estas acciones llamado `CurrentInvoiceEdition`. Has de añadirlo a `controllers.xml`, tal como se muestra en el listado 11.25.

Listado 11.25 Controlador para editar una factura en `controllers.xml`

```
<controller name="CurrentInvoiceEdition">
    <action name="save"
        class="org.openxava.invoicing.actions.SaveInvoiceAction"
        keystroke="Control S"/>

    <action name="return"
        class="org.openxava.actions.ReturnPreviousModuleAction"/>
</controller>
```

Las dos acciones de este controlador representan los dos botones, 'Save' y 'Return' que viste en la anterior figura 11.3.

11.3.3 Volviendo al módulo que llamó

SaveInvoiceAction es un pequeño refinamiento de la estándar SaveAction de OpenXava. El listado 11.26 muestra su código.

Listado 11.26 Acción que graba la factura y vuelve al módulo que llamó

```
public class SaveInvoiceAction
    extends SaveAction // Acción estándar de OpenXava para grabar el contenido de la vista
    implements IChangeModuleAction // Para navegación entre módulos
{
    public String getNextModule() {
        return PREVIOUS_MODULE; // Vuelve al módulo que llamó, Order en este caso
    }

    public boolean hasReinitNextModule() {
        return false; // No queremos inicializar el módulo Order
    }
}
```

La acción extiende de SaveAction sin sobreescribir el método execute(). Por lo tanto su comportamiento es exactamente el mismo que el de la acción genérica de OpenXava para grabar los datos visualizados en la base de datos. Adicionalmente, indicamos que la acción tiene que volver al módulo que la llamó, el módulo Order en nuestro ejemplo, cuando termine.

De esta forma cuando el usuario pulsa en 'Save' los datos de la factura se graban y vuelve a la lista de pedidos, listo para continuar creando facturas desde pedidos.

Para volver al módulo que llama tenemos que usar siempre PREVIOUS_MODULE. No uses el nombre del módulo, como muestra el listado 11.27.

Listado 11.27 Nunca usar el nombre de módulo para volver al módulo que llamó

<pre>public String getNextModule() { return PREVIOUS_MODULE; } // Bien</pre>
<pre>public String getNextModule() { return "Order"; } // Muy MAL</pre>

Si usas PREVIOUS_MODULE tienes la ventaja de que puedes llamar a este módulo desde varios módulos de la aplicación, y éste sabrá a qué módulo volver en cada caso. Pero más importante todavía es el hecho de que OpenXava usa una pila de llamadas a módulos para poder volver, por tanto si llamas a un módulo que te ha llamado se produce un problema de reentrada.

Para el botón 'Return' usamos ReturnPreviousModuleAction, una acción incluida en OpenXava que simplemente vuelve al módulo que llamó.

11.3.4 Objeto de sesión global y acción on-init

El código actual está todavía incompleto. Cuando el usuario genera la factura el módulo `CurrentInvoiceEdition` se activa, pero está vacío, no muestra la factura. Hemos de llenar la vista del nuevo módulo con la factura recién creada. Aprendamos como compartir datos entre módulos.

Una forma de compartir datos entre módulos es declarando un objeto de sesión de ámbito global. Esto se consigue añadiendo una entrada en `controllers.xml` como se muestra en el listado 11.28.

Listado 11.28 Objeto de sesión con ámbito global definido en controllers.xml

```
<controllers>
    ...
    <object name="invoicing_currentInvoiceKey"
        class="java.util.Map"
        scope="global"/>
    <!--
        name="invoicing_currentInvoiceKey": El nombre tiene que ser único
        class="java.util.Map": El tipo del objeto
        scope="global": Compartido por todos los módulos. Por defecto es "module"
    -->
    ...
</controllers>
```

Un objeto de sesión es un objeto asociado a la sesión del usuario, por lo tanto vivirá mientras que la sesión del usuario esté viva, y cada usuario tiene su propia copia del objeto. Si usas `scope="global"` el mismo objeto se compartirá por todos los módulos, en caso contrario cada módulo tiene su propia copia del objeto.

Declaramos el ámbito del objeto como global porque queremos usarlo para pasar datos desde el módulo `Order` al módulo `CurrentInvoiceEdition`. La forma de hace esto es inyectándolo en la acción mediante la anotación `@Inject`¹⁶. Antes de llamar al método `execute()` de la acción, el objeto `invoicing_currentInvoiceKey` se inyecta en el campo `currentInvoiceKey` de la acción. Nota como el nombre del campo es el nombre del objeto de sesión sin el prefijo (sin `invoicing_` en este caso), aunque puedes inyectar el objeto en una propiedad con otro nombre si usas la anotación `@Named`. El listado 11.29 muestra el campo `currentInvoiceKey` con `@Inject` añadido a la acción.

Listado 11.29 Campo currentInvoiceKey a ser inyectado del objeto de sesión

```
...
import javax.inject.*;
```

16 La anotación `@javax.inject.Inject` está definida por el estándar de Java JSR-330

```
public class CreateInvoiceFromSelectedOrdersAction ... {

    ...

    @Inject
    private Map currentInvoiceKey; // Un campo privado sin getter ni setter

    ...

}
```

Lo interesante de `@Inject` es que, además de inyectar el objeto en el campo antes de llamar a `execute()`, extrae el valor del campo y lo vuelve a poner en el contexto de la sesión después de ejecutar el método `execute()`. En otras palabras, si modificaras el valor del campo `currentInvoiceKey` de `CreateInvoiceFromSelectedOrdersAction` entonces el objeto de sesión `invoicing_currentInvoiceKey` se modificaría también. Por lo tanto, podemos usar esta acción para dar valor a este objeto de sesión. El listado 11.30 muestra la modificación en el código de la acción.

Listado 11.30 Llenar el objeto de sesión `currentInvoiceKey` desde la acción

```
public class CreateInvoiceFromSelectedOrdersAction ... {

    ...

    public void execute() throws Exception {
        Collection<Order> orders = getSelectedOrders();
        Invoice invoice = Invoice.createFromOrders(orders);
        addMessage("invoice_created_from_orders",
                   invoice, orders);
        currentInvoiceKey = toKey(invoice); // Pone la clave de la recién creada
    } // factura en el campo currentInvoiceKey, por lo tanto también
      // en el objeto de sesión invoicing_currentInvoiceKey

    private Map toKey(Invoice invoice) { // Extrae la clave de la factura en formato mapa
        Map key = new HashMap();
        key.put("oid", invoice.getOid());
        return key;
    }

    ...
}
```

Después de la creación de la factura, ponemos la clave de la factura en el objeto de sesión. Dar valor a un objeto de sesión es pan comido, solo has de asignar un valor al campo declarado con `@Inject`. En este caso asignar valor a `setCurrentInvoiceKey()` es suficiente para llenar el objeto correspondiente `invoicing_currentInvoiceKey`. Después puedes usar este objeto desde otras acciones, ya que su ámbito es global, también desde las acciones de otros módulos.

215 Capítulo 11: Comportamiento y lógica de negocio

Vamos a crear una nueva acción en el módulo CurrentInvoiceEdition para cargar el valor de la factura creada en el módulo Order con CreateInvoiceFromSelectedOrdersAction. El listado 11.31 muestra la declaración de esta acción load en el archivo *controllers.xml*.

Listado 11.31 La declaración de la acción load en controllers.xml con on-init=true

```
<controller name="CurrentInvoiceEdition">

    <action name="load"
        class="org.openxava.invoicing.actions.LoadCurrentInvoiceAction"
        hidden="true"
        on-init="true"/>
    <!--
        hidden="true" : No hay un vínculo o botón en la pantalla para esta acción
        on-init="true": Se ejecuta automáticamente cuando el módulo se inicializa
    -->
    ...
</controller>
```

Declaramos la acción como `hidden=true`, así no será visible, y por tanto el usuario no tendrá la posibilidad de ejecutarla. Además, la declaramos como `on-init=true`, por tanto se ejecutará automáticamente cuando el módulo se inicialice.

Recuerda que llamamos a este módulo devolviendo `true` para `hasReinitNextModule()`, así CurrentInvoiceEdition se inicializa cada vez que se llama desde el módulo Order, por ende la acción `load` se llama siempre. Esta acción `load` es el lugar ideal para llenar la vista con la factura recién creada. Veamos su código en el listado 11.32.

Listado 11.32 Acción para cargar la última factura cargada en la vista

```
public class LoadCurrentInvoiceAction
    extends SearchByViewKeyAction { // Para llenar la vista a partir de la clave

    @Inject
    private Map currentInvoiceKey; // Para coger el valor del objeto de sesión
                                    // invoicing_currentInvoiceKey, llenado en el módulo Order
    public void execute() throws Exception {
        getView().setValues(currentInvoiceKey); // Pone la clave en la vista
        super.execute(); // Llena toda la vista a partir de los campos clave
    }
}
```

Extiende de `SearchByViewKeyAction` la cual es la acción estándar de OpenXava para buscar. `SearchByViewKeyAction` coge los campos clave de la vista, busca la entidad correspondiente, y rellena el resto de la vista a partir de la entidad. Por lo tanto, nosotros sólo hemos de llenar la vista con los valores de la clave antes de llamar a `super.execute()`.

Puedes ver como usando `currentInvoiceKey` accedemos a los valores de la clave almacenados ahí por `CreateInvoiceFromSelectedOrdersAction`. Has visto como usar un objeto de sesión para compartir datos entre acciones, incluso si éstas son de módulos diferentes.

El trabajo está casi terminado. Puedes probar el módulo `Order`: escoge varios pedidos y pulsa en el botón 'Create invoice from selected orders'. Entonces verás la factura creada en un módulo de solo detalle. Tal como viste en la anterior figura 11.3.

11.4 Pruebas JUnit

El código que hemos escrito en este capítulo no estará completo hasta que no escribamos las pruebas. Recuerda, todo código nuevo tiene que tener su correspondiente código de prueba. Escribamos pues las pruebas para estas dos nuevas acciones.

11.4.1 Probar la acción de modo detalle

Primero probaremos la acción `Order.createInvoice`, la acción para crear una factura a partir del modo detalle del pedido visualizado. Reimprimimos aquí la figura 11.1 que muestra como funciona este proceso.

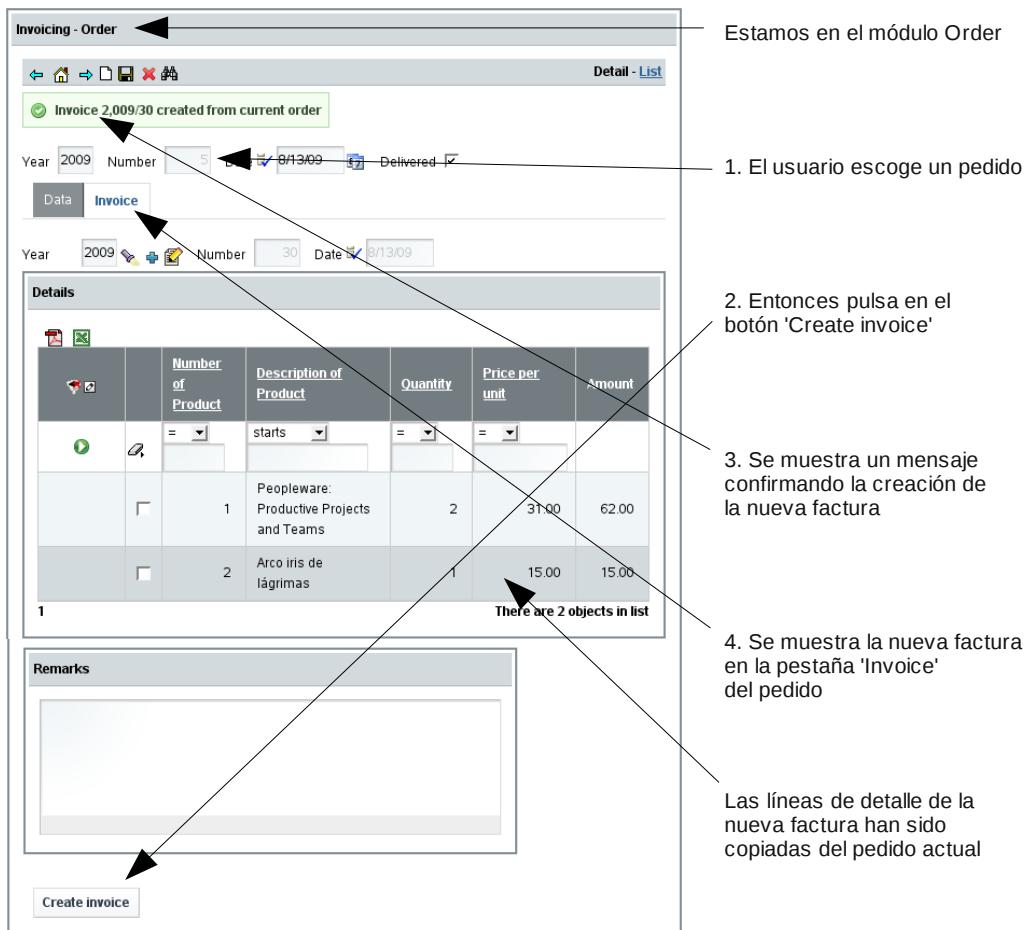


Figura 11.1 (reimpresión) Crear una factura desde un pedido usando una acción

Ahora vamos a escribir un test para verificar que funciona justo de esta forma. Añade el método `testCreateInvoiceFromOrder()` del listado 11.33 a la clase `OrderTest`.

Listado 11.33 El método `testCreateInvoiceFromOrder()` en `OrderTest`

```
public void testCreateInvoiceFromOrder() throws Exception {
    // Buscar el pedido
    searchOrderSusceptibleToBeInvoiced(); // Busca un pedido
    assertEquals("delivered", "true"); // El pedido está entregado
    int orderDetailsCount = getCollectionRowCount("details"); // Toma nota de
                                                               // la cantidad de detalles en el pedido
    execute("Sections.change", "activeSection=1"); // La sección de la factura
    assertEquals("invoice.year", ""); // Todavía no hay factura
    assertEquals("invoice.number", ""); // en este pedido

    // Crear la factura
    execute("Order.createInvoice"); // Ejecuta la acción que estamos probando (1)
    String invoiceYear = getValue("invoice.year"); // Verifica que ahora
```

```

    assertTrue("Invoice year must have value",           // hay una factura
               !Is.emptyString(invoiceYear));                // en la pestaña de factura (2)
    String invoiceNumber = getValue("invoice.number");
    assertTrue("Invoice number must have value",
               !Is.emptyString(invoiceNumber)); // Is.emptyString() es de org.openxava.util
    assertMessage("Invoice " + invoiceYear + "/" + invoiceNumber +
                  " created from current order"); // El mensaje de confirmación (3)
    assertCollectionRowCount("invoice.details", // La factura recién creada
                            orderDetailsCount); // tiene el mismo número de detalles que el pedido (4)

    // Restaurar el pedido para poder ejecutar la prueba la siguiente vez
    setValue("invoice.year", "");
    assertEquals("invoice.number", "");
    assertCollectionRowCount("invoice.details", 0);
    execute("CRUD.save");
    assertNoErrors();
}

```

Esta prueba pulsa el botón para ejecutar la acción `Order.createInvoice` (1), entonces verifica que una factura ha sido creada, está siendo visualizada en la pestaña de factura (2) y tiene la misma cantidad de líneas de detalle que el pedido actual (4). También verifica que se ha generado el mensaje de confirmación correcto (3).

Para ejecutarla es necesario escoger un pedido susceptible de ser facturado. Esto se hace en el método `searchOrderSusceptibleToBeInvoiced()` que vamos a examinar en la siguiente sección.

11.4.2 Buscar una entidad para la prueba usando el modo lista y JPA

Para seleccionar un pedido adecuado para nuestra prueba usaremos JPA para determinar el año y número de ese pedido, y entonces usaremos el modo lista para seleccionar este pedido y editarlo en modo detalle. El listado 11.34 muestra los métodos para implementar esto.

Listado 11.34 Métodos de `OrderTest` para buscar un pedido usando JPA en lista

```

private void searchOrderSusceptibleToBeInvoiced() throws Exception {
    searchOrderUsingList("o.delivered = true and o.invoice = null"); // Envía
} // la condición, en este caso buscamos por un pedido entregado y sin factura

private void searchOrderUsingList(String condition) throws Exception {
    Order order = findOrder(condition); // Busca el pedido con la condición usando JPA
    String year = String.valueOf(order.getYear());
    String number = String.valueOf(order.getNumber());
    setConditionValues(new String [] { year, number }); // Llena el año y el número
    execute("List.filter"); // y pulsa en el botón filtrar en la lista
    assertEquals(1); // Sólo una fila, correspondiente al pedido buscado
    execute("Mode.detailAndFirst"); // Para ver el pedido en modo detalle
    assertEquals("year", year); // Verifica que el pedido editado
    assertEquals("number", number); // es el deseado
}

```

```
private Order findOrder(String condition) {
    Query query = XPersistence.getManager().createQuery( // Crea una consulta JPA
        "from Order o where o.deleted = false and " // a partir de la condición. Fíjate en
        + condition);                                // deleted = false para excluir los pedidos borrados
    List orders = query.getResultList();
    if (orders.isEmpty()) { // Es necesario al menos un pedido con la condición
        fail("To run this test you must have some order with " + condition);
    }
    return (Order) orders.get(0);
}
```

El método `searchOrderSusceptibleToBeInvoiced()` simplemente llama a un método más genérico, `searchOrderUsingList()`, para buscar una entidad por una condición. El método `searchOrderUsingList()` obtiene la entidad `Order` mediante `findOrder()`, entonces usa la lista para filtrar por el año y el número a partir de este `Order`, yendo a modo detalle al finalizar. El método `findOrder()` usa JPA simple y llano para buscar.

Como puedes ver, combinar el modo lista con JPA es una herramienta muy útil en ciertas circunstancias. Usaremos los métodos `searchOrderUsingList()` y `findOrder()` en las siguientes pruebas.

11.4.3 Probar que la acción se oculta cuando no aplica

Recuerda que refinamos el módulo `Order` en la sección 11.1.7 para que mostrara la acción para crear la factura solo cuando el pedido visualizado fuese susceptible de ser facturado. El listado 11.35 muestra el método de prueba para este caso.

Listado 11.35 Probar que la acción se oculta correctamente en OrderTest

```
public void testHidesCreateInvoiceFromOrderWhenNotApplicable()
    throws Exception
{
    searchOrderUsingList(
        "delivered = true and invoice <> null"); // Si el pedido ya tiene factura
    assertNoAction("Order.createInvoice"); // no se puede facturar otra vez

    execute("Mode.list");

    searchOrderUsingList(
        "delivered = false and invoice = null"); // Si el pedido no está entregado
    assertNoAction("Order.createInvoice"); // no se puede facturar

    execute("CRUD.new"); // Si el pedido todavía no está grabado
    assertNoAction("Order.createInvoice"); // no puede ser facturado
}
```

Probamos tres casos en los que el botón para crear la factura no tiene que estar presente. Fíjate en el uso de `assertNoAction()` para preguntar si el vínculo o botón para una acción está presente en la interfaz de usuario. Aquí estamos

reutilizando el método `searchOrderUsingList()` desarrollado en la sección anterior.

Ya hemos probado que el botón está presente cuando el pedido es adecuado en la prueba `testCreateInvoiceFromOrder()`, porque `execute()` falla si la acción no está en la interfaz de usuario.

11.4.4 Probar la acción de modo lista

Ahora probaremos `Order.createInvoiceFromSelectedOrders`, la acción que crea una factura desde varios pedidos en modo lista (figura 11.3).

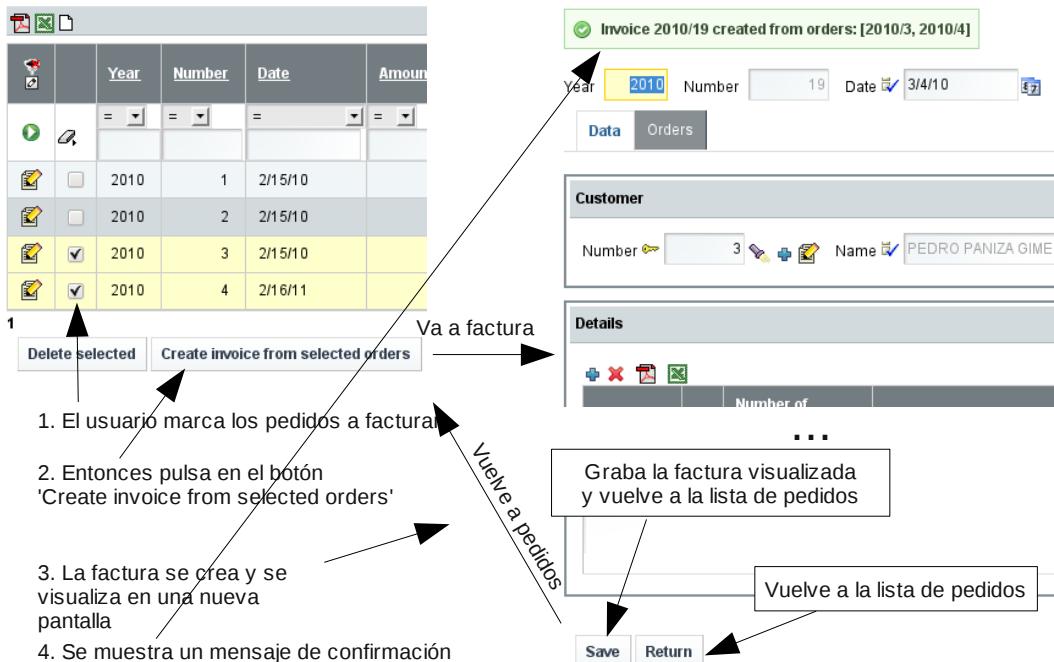


Figura 11.3 (reimp.) Editar la factura después de crearla a partir de varios pedidos

Escribamos una prueba para verificar que esto funciona justo de esta forma. Añade el método `testCreateInvoiceFromSelectedOrders()` del listado 11.36 a la clase `OrderTest`.

Listado 11.36 El método `testCreateInvoiceFromSelectedOrders()` en `OrderTest`

```
public void testCreateInvoiceFromSelectedOrders() throws Exception {
    assertOrder(2010, 9, 2, 362); // El pedido 2010/9 tiene 2 líneas y 362 de importe base
    assertOrder(2010, 10, 1, 126); // El pedido 2010/10 tiene 1 línea y 126 de importe base

    execute("List.orderBy", "property=number"); // Ordena la lista por número
    checkRow( // Marca la fila a partir del número de fila
        getDocumentRowInList("2010", "9") // Obtiene la fila del año y número del pedido
    ); // por tanto, esta línea marca la línea del pedido 2010/9 en la lista (1)
    checkRow(
```

221 Capítulo 11: Comportamiento y lógica de negocio

```
getDocumentRowInList("2010", "10")
); // Marca el pedido 2010/10 en la lista (1)

execute("Order.createInvoiceFromSelectedOrders"); // Ejecuta la acción que
// estamos probando (2)

String invoiceYear = getValue("year"); // Ahora estamos viendo el detalle de
String invoiceNumber = getValue("number"); // la factura recién creada
assertMessage("Invoice " + invoiceYear + "/" + invoiceNumber +
    " created from orders: [2010/9, 2010/10]"); // El mensaje de confirmación
assertCollectionRowCount("details", 3); // Confirma que el número de líneas de la
// factura recién creada es la suma de la de los pedidos fuente (3)
assertValue("baseAmount", "488.00"); // Confirma que el importe base de la factura
// recién creada es la suma de la de los pedidos fuente (4)
execute("Sections.change", "activeSection=1"); // Cambia a la pestaña de
// pedidos de la factura
assertCollectionRowCount("orders", 2); // La nueva factura tiene 2 pedidos (5)
assertValueInCollection("orders", 0, 0, "2010"); // y son los correctos
assertValueInCollection("orders", 0, 1, "9");
assertValueInCollection("orders", 1, 0, "2010");
assertValueInCollection("orders", 1, 1, "10");

assertAction("CurrentInvoiceEdition.save"); // Los botones 'Save' (6)
assertAction("CurrentInvoiceEdition.return"); // y 'Return' (6)

checkRowCollection("orders", 0); // Seleccionamos los 2 pedidos
checkRowCollection("orders", 1);
execute("Collection.removeSelected", // y los borramos, para ejecutar esta prueba
    "viewObject=xava_view_section1_orders"); // otra vez usando los mismo pedidos
assertNoErrors();

execute("CurrentInvoiceEdition.return"); // Vuelve a la lista de pedidos (7)
assertDocumentInList("2010", "9"); // Confirma que estamos realmente
assertDocumentInList("2010", "10"); // en la lista de pedidos
}
```

Esta prueba marca dos pedidos (1) y pulsa en el botón 'Create invoice from selected orders' (2). Entonces verifica que se ha creado una nueva factura con el número correcto de líneas (3), importe base (4) y lista de pedidos (5). También verifica que las acciones 'Save' y 'Return' están disponibles (6) y usa el botón 'Return' para volver a la lista de pedidos (7).

Usamos `getDocumentRowInList()` y `assertDocumentInList()`, métodos de la clase base `CommercialDocumentTest`, que fueron definidos originalmente como privados, por lo tanto tenemos que redefinirlos como protegidos para poder utilizarlos desde `OrderTest`. Edita `CommercialDocumentTest` y haz los cambios del listado 11.37.

Listado 11.37 Cambia de private a protected en CommercialDocumentTest

```
protected private void assertDocumentInList(String year, String number) ...
protected private int getDocumentRowInList(String year, String number) ...
```

El único detalle pendiente es el método `assertOrder()` que veremos en la

siguiente sección.

11.4.5 Verificar datos de prueba

En la sección 6.5 (capítulo 6) aprendiste como confiar en datos existentes en la base de datos para tus pruebas. Obviamente, si tu base de datos se altera accidentalmente tus pruebas, aunque correctas, no pasaran. Por tanto, verificar los valores de la base de datos antes de ejecutar la prueba que confía en ellos es una buena práctica. En nuestro ejemplo lo hacemos llamando a `assertOrder()` al principio. Veamos el contenido de `assertOrder()` en el listado 11.38.

Listado 11.38 Método para verificar el estado de un pedido ya existente

```
private void assertOrder(
    int year, int number, int detailsCount, int baseAmount)
{
    Order order = findOrder("year = " + year + " and number=" + number);
    assertEquals("To run this test the order " +
        order + " must have " + detailsCount + " details",
        detailsCount, order.getDetails().size());
    assertTrue("To run this test the order " +
        order + " must have " + baseAmount + " as base amount",
        order.getBaseAmount().compareTo(new BigDecimal(baseAmount)) == 0);
}
```

Este método busca un pedido y verifica la cantidad de líneas y el importe base. Usar este método tiene la ventaja de que si los pedidos necesarios para la prueba no están en la base de datos con los valores correctos obtienes un mensaje preciso. Así, no derrocharás tu tiempo intentando adivinar que es lo que está mal. Esto es especialmente útil si la prueba no la está ejecutando el programador original.

11.4.6 Probar casos excepcionales

Dado que la acción para crear la factura se oculta si el pedido no está listo para ser facturado, no podemos probar el código para los casos excepcionales que escribimos en la sección 11.1.5 desde modo detalle. Sin embargo, en modo lista el usuario todavía tiene la opción de escoger cualquier pedido para facturar. Por tanto, intentaremos crear la factura desde la lista de pedidos para probar que los casos excepcionales se comportan correctamente. El listado 11.39 muestra el código de prueba en `OrderTest`.

Listado 11.39 Probar casos excepcionales creando una factura desde un pedido

```
public void testCreateInvoiceFromOrderExceptions() throws Exception {
    assertCreateInvoiceFromOrderException( // Verifica que cuando el pedido ya tiene (1)
        "delivered = true and invoice <> null", // factura se produce el error correcto
        "Impossible to create invoice: the order already has an invoice"
);
```

```

    assertCreateInvoiceFromOrderException( // Verifica que cuando el pedido no está (2)
        "delivered = false and invoice = null", // entregado se produce el error correcto
        "Impossible to create invoice: the order is not delivered yet"
    );
}

private void assertCreateInvoiceFromOrderException(
    String condition, String message) throws Exception
{
    Order order = findOrder(condition); // Busca el pedido por la condición (3)
    int row = getDocumentRowInList( // y obtiene el número de fila para ese pedido (4)
        String.valueOf(order.getYear()),
        String.valueOf(order.getNumber())
    );
    checkRow(row); // Marca la fila (5)
    execute("Order.createInvoiceFromSelectedOrders"); // Trata de crear la factura (6)
    assertError(message); // ¿Se ha mostrado el mensaje esperado? (7)
    uncheckRow(row); // Desmarca la fila, así podemos llamar a este método otra vez
}

```

La prueba verifica que el mensaje es el correcto cuando tratamos de crear una factura a partir de un pedido que ya tiene factura (1), y también desde un pedido no entregado todavía (2). Para hacer estas verificaciones llama al método `assertCreateInvoiceFromOrderException()`. Este método busca la entidad `Order` usando la condición (3), localiza la fila donde la entidad se está visualizando (4) y la marca (5). Después, la prueba ejecuta la acción (6) y verifica que el mensaje esperado se muestra (7).

11.5 Resumen

La sal de tu aplicación son las acciones y los métodos. Gracias a ellos puedes convertir una simple aplicación de gestión de datos en una herramienta útil. En este caso, por ejemplo, hemos provisto al usuario con una forma de crear automáticamente facturas desde pedidos.

Has aprendido como crear métodos de lógica de negocio tanto estáticos como de instancia, y como llamarlos desde acciones de modo detalle y modo lista. Por el camino has visto como ocultar y mostrar acciones, usar excepciones, validar en las acciones, cambiar a otro módulo y cómo hacer las pruebas automáticas de todo esto.

Todavía nos quedan muchas cosas interesante por aprender, por ejemplo en el siguiente capítulo vamos a refinarn el comportamiento de las referencias y colecciones.

*Referencias
y colecciones*

capítulo 12

En capítulos anteriores aprendiste como añadir tus propias acciones. Sin embargo, esto no es suficiente para personalizar del todo el comportamiento de tu aplicación, porque la interfaz de usuario generada, en concreto la interfaz de usuario para referencias y colecciones, tiene un comportamiento estándar que a veces no es el más conveniente.

Por fortuna, OpenXava proporciona muchas formas de personalizar el comportamiento de las referencias y colecciones. En este capítulo aprenderás como hacer algunas de estas personalizaciones, y como esto añade valor a tu aplicación.

12.1 Refinar el comportamiento de las referencias

Possiblemente te hayas dado cuenta de que el módulo Order tiene un pequeño defecto: el usuario puede añadir cualquier factura que quiera al pedido actual, aunque el cliente de la factura sea diferente. Esto no es admisible. Arreglémoslo.

12.1.1 Las validaciones están bien, pero no son suficientes

El usuario sólo puede asociar un pedido a una factura si ambos, factura y pedido, pertenecen al mismo cliente. Esto es lógica de negocio específica de tu aplicación, por tanto el comportamiento estándar de OpenXava no lo resuelve.

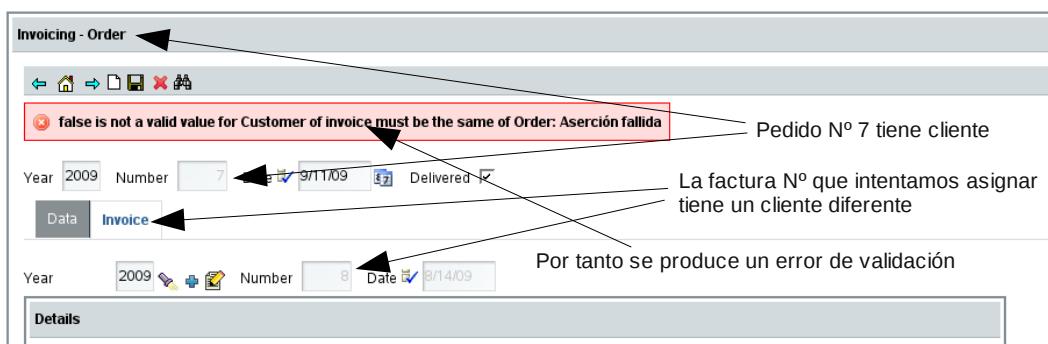


Figura 12.1 Error de validación cuando el cliente de la factura es incorrecto

Ya que esto es lógica de negocio la vamos a poner en la capa del modelo, es decir, en las entidades. Lo haremos añadiendo una validación. Así obtendrás el efecto de la figura 12.1.

Ya sabes como añadir esta validación a tu entidad Order. Se trata de añadir un método anotado con `@AssertTrue`. Puedes verlo en el listado 12.1.

Listado 12.1 Nuevo método de validación en la entidad Order

```
@AssertTrue // Este método tiene que devolver true para que este pedido sea válido
private boolean isCustomerOfInvoiceMustBeTheSame() {
    return invoice == null || // invoice es opcional
           invoice.getCustomer().getNumber()==getCustomer().getNumber();
}
```

Aquí comprobamos que el cliente de la factura es el mismo que el del pedido. Esto es suficiente para preservar la integridad de los datos, pero la validación sola es una opción bastante pobre desde el punto de vista del usuario.

12.1.2 Modificar los datos tabulares por defecto ayuda

Aunque la validación impide que el usuario pueda asignar una factura incorrecta a un pedido, lo tiene difícil a la hora de escoger una factura correcta. Porque cuando pulsa para buscar una factura, todas las facturas existentes se muestran, y lo que es todavía peor, la información del cliente no aparece en la lista. Fíjate en la figura 12.2.

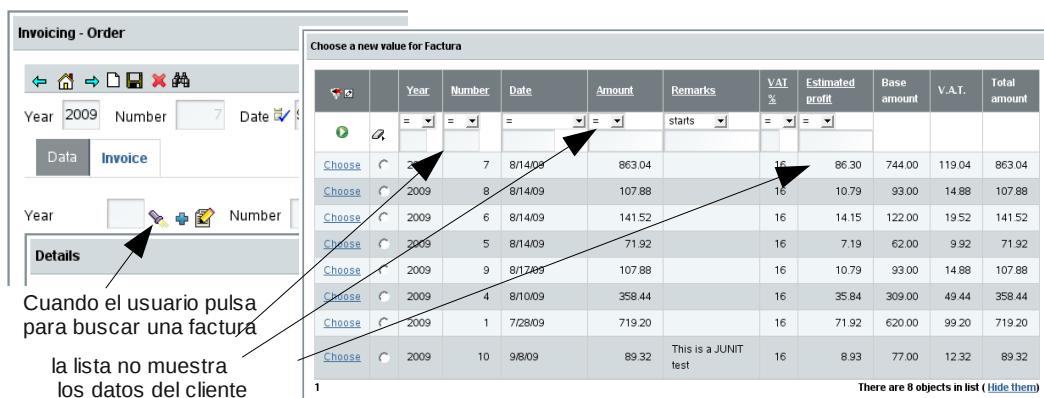


Figura 12.2 La lista para buscar facturas no muestra los datos del cliente

Obviamente, es difícil buscar una factura sin ver de qué cliente es. Añadimos pues el cliente a la lista usando el atributo `properties` de `@Tab` en la entidad `Invoice`, tal como muestra el listado 12.2.

Listado 12.2 Definición de datos tabulares para Invoice

```
@Tabs({
    @Tab(
        baseCondition = "deleted = false",
        properties="year, number, date, customer.number, customer.name," +
        "vatPercentage, estimatedProfit, baseAmount, " +
        "vat, totalAmount, amount, remarks"),
    @Tab(name="Deleted", baseCondition = "deleted = true")
})
public class Invoice extends CommercialDocument {
```

Los datos tabulares por defecto (es decir, el modo lista) para una entidad son todas sus propiedades planas, pero no incluyen las referencias. Para cambiar la forma en que los datos tabulares se muestran puedes usar properties en la anotación @Tab. Estas propiedades pueden ser calificadas, es decir puedes, usando la notación del punto, poner una propiedad de una referencia, como `customer.number` y `customer.name` en este caso.

The screenshot shows a table titled "Choose a new value for Factura". The columns are: Year, Number, Date, Number of Customer, Name of Customer, VAT %, Estimated profit, Base amount, V.A.T., and Total amount. The first row has filters: Year = 2009, Number = 7, Date = 8/14/09, and Number of Customer starts with FRANCISCO. The second row has filters: Year = 2009, Number = 8, Date = 8/14/09, and Number of Customer starts with M. The third row has filters: Year = 2009, Number = 2, Date = 8/14/09, and Number of Customer starts with J. A callout box points to the customer names in the table, with the text "Se pueden ver los datos del cliente". Below the table, the @Tab annotation is shown:

```
@Tab(properties="year, number, date, customer.number, customer.name, " +
    "vatPercentage, estimatedProfit, baseAmount, " +
    "vat, totalAmount, amount, remarks")
```

Figura 12.3 Gracias a @Tab los datos del cliente se ven en la lista de facturas

Ahora la lista para escoger una factura de un pedido es como la que se muestra en la figura 12.3.

Con esta lista de facturas es más fácil escoger la correcta, porque ahora el usuario puede ver el cliente de cada factura. Además, el usuario puede filtrar por cliente para mostrar las facturas del cliente que está buscando. Sin embargo, sería aun mejor si solo se mostraran las facturas cuyo cliente es el mismo que del pedido actual. De esta manera no habría opción para equivocarse. Lo haremos así en la siguiente sección.

12.1.3 Refinar la acción para buscar una referencia con una lista

Actualmente cuando el usuario busca una factura todas las facturas están disponibles para escoger. Vamos a mejorar esto para mostrar solo las facturas del cliente del pedido visualizado, tal como muestra la figura 12.4.



Figura 12.4 Buscar la factura desde el pedido tiene que filtrarse por cliente

Para definir nuestra propia acción de búsqueda para la referencia a factura usaremos la anotación `@SearchAction`. El listado 12.3 muestra la modificación necesaria en la clase `Order`.

Listado 12.3 @SearchAction define la acción personalizada para buscar facturas

```
public class Order extends CommercialDocument {
    @ManyToOne
    @ReferenceView("NoCustomerNoOrders")
    @OnChange(ShowHideCreateInvoiceAction.class)
    @SearchAction("Order.searchInvoice") // Define nuestra acción para buscar facturas
    private Invoice invoice;
    ...
}
```

De esta forma tan simple definimos la acción a ejecutar cuando el usuario pulsa en el botón de la linterna para buscar una factura. El argumento usado para `@SearchAction`, `Order.searchInvoice`, es el nombre calificado de la acción, es decir la acción `searchInvoice` del controlador `Order` definido en el archivo `controllers.xml`.

Ahora tenemos que editar `controllers.xml` y añadir la definición de nuestra nueva acción, tal como muestra el listado 12.4.

Listado 12.4 Declaración de la acción Order.searchInvoice en controllers.xml

```
<controller name="Order">
    ...
    <action name="searchInvoice"
        class="org.openxava.invoicing.actions.SearchInvoiceFromOrderAction"
        hidden="true" image="images/search.gif"/>
    <!--
```

```

hidden="true": Para que no se muestre en la barra de botones del módulo
image="images/search.gif": La misma imagen que la de la acción estándar
-->

</controller>

```

Nuestra acción hereda de ReferenceSearchAction y ésta los necesita. El listado 12.5 muestra el código de la acción.

Listado 12.5 Acción personalizada para buscar una factura desde un pedido

```

package org.openxava.invoicing.actions; // En el paquete 'actions'

import org.openxava.actions.*; // Para usar ReferenceSearchAction

public class SearchInvoiceFromOrderAction
    extends ReferenceSearchAction { // Lógica estándar para buscar una referencia

    public void execute() throws Exception {
        super.execute(); // Ejecuta la lógica estándar, la cual muestra un diálogo
        int customerNumber =
            getPreviousView().getValueInt("customer.number"); // Lee de la vista el número
            // de cliente del pedido actual
        if (customerNumber > 0) { // Si hay cliente los usamos para filtrar
            getTab().setBaseCondition("${customer.number} = " + customerNumber);
        }
    }
}

```

Observa como usamos getTab().setBaseCondition() para establecer una condición en la lista para escoger la referencia. Es decir, desde una ReferenceSearchAction puedes usar getTab() para manipular la forma en que se comporta la lista.

Si no hay cliente no añadimos ninguna condición por tanto se mostrarían todas las facturas, esto ocurre cuando el usuario escoge la factura antes que el cliente.

12.1.4 Buscar la referencia tecleando en los campos

La lista para escoger una referencia ya funciona bien. Sin embargo, queremos dar al usuario la opción de escoger una factura sin usar la lista, simplemente tecleando el año y el número. Muy útil si el usuario conoce de antemano que factura quiere.

OpenXava provee esa funcionalidad por defecto. Si los campos @Id son visualizados en la referencia serán usados para buscar, en caso contrario OpenXava usa el primer campo visualizado para buscar. Aunque en nuestro caso esto no es tan conveniente, porque el primer campo visualizado es el año, y buscar una factura sólo por el año no es muy preciso. La figura 12.5 muestra el comportamiento por defecto junto con una alternativa más conveniente.

231 Capítulo 12: Referencias y colecciones



Figura 12.5 Por defecto la factura se recupera solo por año

Afortunadamente es fácil indicar que campos queremos usar para buscar desde la perspectiva del usuario. Esto se hace por medio de la anotación @SearchKey. Edita la clase CommercialDocument (recuerda, el padre de Order e Invoice) y añade esta anotación a las propiedades year y number (listado 12.6).

Listado 12.6 Definir year y number como @SearchKey en CommercialDocument

```
abstract public class CommercialDocument extends Deletable {  
    ...  
    @SearchKey // Añade esta anotación aquí  
    @Column(length=4)  
    @DefaultValueCalculator(CurrentYearCalculator.class)  
    private int year;  
  
    @SearchKey // Añade esta anotación aquí  
    @Column(length=6)  
    @ReadOnly  
    private int number;  
    ...  
}
```

De esta forma cuando el usuario busque un pedido o una factura desde una referencia tiene que teclear el año y el número, y la entidad correspondiente será recuperada de la base de datos y llenará la interfaz de usuario.

Ahora es fácil para el usuario escoger una factura desde un pedido sin usar la lista de búsqueda, simplemente tecleando el año y el número.

12.1.5 Refinar la acción para buscar cuando se teclea la clave

Ahora que obtener una factura tecleando el año y el número funciona queremos refinarlo para ayudar al usuario a hacer su trabajo de forma más eficiente. Por ejemplo, sería útil que si el usuario todavía no ha escogido al

cliente para el pedido y escoge una factura, el cliente de esa factura sea asignado automáticamente al pedido actual. La figura 12.6 visualiza el comportamiento deseado.

Figura 12.6 Escoger una factura cuando todavía no hay un cliente seleccionado

Por otra parte, si el usuario ya ha seleccionado un cliente para el pedido, si no coincide con el de la factura, ésta será rechazada y se visualizará un mensaje de error, tal como muestra la figura 12.7.

Figura 12.7 Escoger una factura cuando el cliente ya está seleccionado

Para definir este comportamiento especial hemos de añadir una anotación @OnChangeSearch en la referencia invoice de Order. @OnChangeSearch permite definir nuestra propia acción para hacer la búsqueda de la referencia cuando su clave cambia en la interfaz de usuario. Puedes ver la referencia modificada en el listado 12.7.

Listado 12.7 Acción para obtener la factura desde el pedido al cambiar la clave

```
public class Order extends CommercialDocument {
```

```

@ManyToOne
@ReferenceView("NoCustomerNoOrders")
@OnChange(ShowHideCreateInvoiceAction.class)
@OnChangeSearch(OnChangeSearchInvoiceAction.class) // Añade esta anotación
@SearchAction("Order.searchInvoice")
private Invoice invoice;

...
}

```

A partir de ahora cuando un usuario teclee un nuevo año y número para la factura, OnChangeSearchInvoiceAction se ejecutará. En esta acción se han de leer los datos de la factura de la base de datos y actualizar la interfaz de usuario. El listado 12.8 muestra el código de la acción.

Listado 12.8 Acción para buscar la factura al teclear año y número

```

package org.openxava.invoicing.actions; // En el paquete 'actions'

import java.util.*;
import org.openxava.actions.*; // Para usar OnChangeSearchAction
import org.openxava.invoicing.model.*;
import org.openxava.model.*;
import org.openxava.view.*;

public class OnChangeSearchInvoiceAction
    extends OnChangeSearchAction { // Lógica estándar para buscar una referencia cuando
        // los valores clave cambian en la interfaz de usuario (1)
    public void execute() throws Exception {
        super.execute(); // Ejecuta la lógica estándar (2)
        Map keyValues = getView() // getView() aquí es la de la referencia, no la principal(3)
            .getKeyValuesWithValue();
        if (keyValues.isEmpty()) return; // Si la clave está vacía no se ejecuta más lógica
        Invoice invoice = (Invoice) // Buscamos la factura usando la clave tecleada (4)
            MapFacade.findEntity(getView().getModelName(), keyValues);
        View customerView = getView().getRoot().getSubview("customer"); // (5)
        int customerNumber = customerView.getValueInt("number");
        if (customerNumber == 0) { // Si no hay cliente lo llenamos (6)
            customerView.setValue("number", invoice.getCustomer().getNumber());
            customerView.refresh();
        }
        else { // Si ya hay un cliente verificamos que coincida con el cliente de la factura (7)
            if (customerNumber != invoice.getCustomer().getNumber()) {
                addError("invoice_customer_not_match",
                    invoice.getCustomer().getNumber(), invoice, customerNumber);
                getView().clear();
            }
        }
    }
}

```

Dado que la acción desciende de OnChangeSearchAction (1) y usamos super.execute() (2) se comporta de la forma estándar, es decir, cuando el usuario teclea el año y el número los datos de la factura se recuperan y rellenan la

interfaz de usuario. Después, usamos `getView()` (3) para obtener la clave de la factura visualizada y así encontrar su correspondiente entidad usando `MapFacade` (4). Desde dentro de `ChangeSearchAction` `getView()` devuelve la subvista de la referencia, y no la vista global. Por lo tanto, en este caso `getView()` es la vista de la referencia a factura. Esto permite crear acciones `@OnChangeEvent` más reutilizables. Has de escribir `getView().getRoot().getSubview("customer")` (5) para acceder a la vista del cliente.

Para implementar el comportamiento visualizado en la anterior figura 12.6, la acción pregunta si no hay cliente (`customerNumber == 0`) (6). Si éste es el caso rellena los datos del cliente desde el cliente de la factura. En caso contrario implementa la lógica de la figura 12.7 verificando que el cliente del pedido actual coincide con el cliente de la factura recuperada.

Nos queda un pequeño detalle, el texto del mensaje. Añade la entrada mostrada en el listado 12.9 al archivo *Invoicing-messages_en.properties* de la carpeta *i18n*.

Listado 12.9 Error de búsqueda de factura en Invoicing-messages_en.properties

```
invoice_customer_not_match=Customer Nº {0} of invoice {1} does not match with
Customer Nº {2} of the current order
```

Una cosa interesante de `@OnChangeEvent` es que también se ejecuta si la factura se escoge desde la lista, porque en este caso el año y el número también cambian. Por ende, este es un lugar centralizado donde refinar la lógica para recuperar la referencia y llenar la vista.

12.2 Refinar el comportamiento de las colecciones

Podemos refinar las colecciones de la misma forma que hemos hecho con las referencias. Esto es muy útil, porque nos permite mejorar el comportamiento actual del módulo *Invoice*. El usuario sólo puede añadir un pedido a una factura si la factura y el pedido pertenecen al mismo cliente. Además, el pedido tiene que estar entregado (*delivered*) y no tener todavía factura.

12.2.1 Modificar los datos tabulares ayuda

Con el comportamiento por defecto, el usuario puede tener dificultades al tratar de encontrar pedidos adecuados para asignar a su factura. Porque cuando el usuario pulsa para añadir pedidos, todos los pedidos existentes son mostrados, y lo que es peor, la información del cliente no se muestra en la lista. Queremos que los datos del cliente se muestren en la lista de pedidos, tal como muestra la figura 12.8.

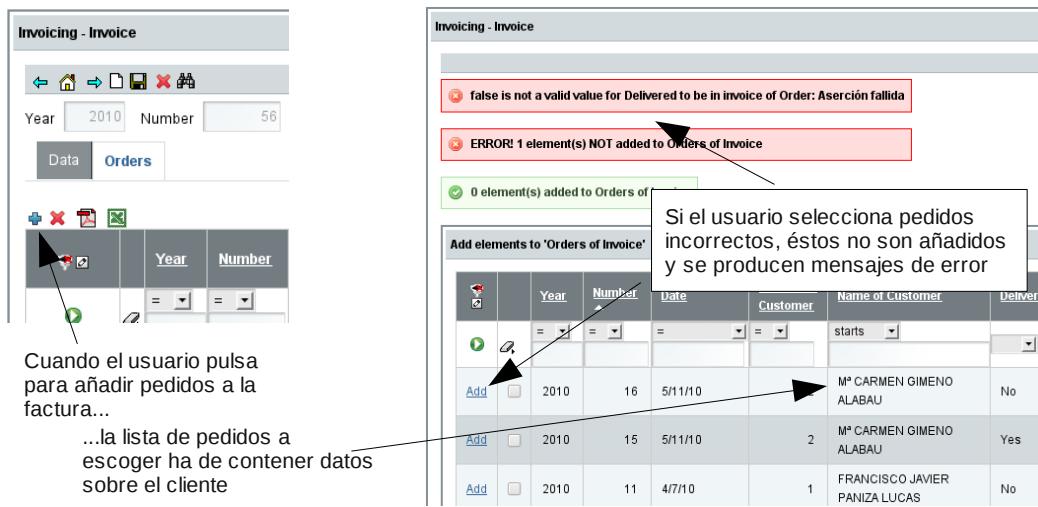


Figura 12.8 Añadir pedidos a factura mostrando los datos del cliente en la lista

El listado 12.10 muestra como añadir el cliente a la lista usando el atributo properties de @Tab en la entidad Order.

Listado 12.10 Definición de datos tabulares para Order

```
@Tabs({
    @Tab(baseCondition = "deleted = false",
        properties="year, number, date, customer.number, customer.name," +
        "delivered, vatPercentage, estimatedProfit, baseAmount, " +
        "vat, totalAmount, amount, remarks"
    ),
    @Tab(name="Deleted", baseCondition = "deleted = true")
})
public class Order extends CommercialDocument {
```

Fíjate como hemos añadido customer.number y customer.name.

La figura 12.8 también muestra como la validación en las entidades impide que el usuario añada pedidos incorrectos.

Sin embargo, sería mejor si sólo los pedidos susceptibles de ser añadidos a la factura actual estuvieran presentes en la lista, de tal modo que el usuario no tuviese forma de equivocarse. Lo haremos así en la siguiente sección.

12.2.2 Refinar la lista para añadir elementos a la colección

Actualmente cuando el usuario trata de añadir pedidos a la factura todos los pedidos están disponibles. Vamos a mejorar esto para mostrar solo los pedidos del cliente de la factura, entregados y todavía sin factura, tal como muestra la figura 12.9.

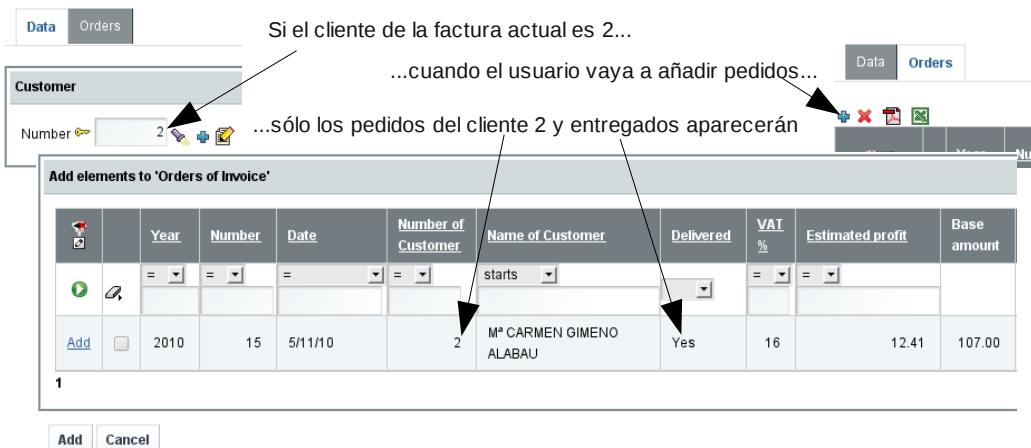


Figura 12.9 Añadir pedidos muestra solo los entregados y del cliente actual

Usaremos la anotación @NewAction para definir nuestra propia acción que muestre la lista para añadir pedidos. El listado 12.11 muestra la modificación necesaria en la clase Order.

Listado 12.11 @NewAction define la acción para ir a la lista de añadir pedidos

```
public class Invoice extends CommercialDocument {
    @OneToMany(mappedBy="invoice")
    @CollectionView("NoCustomerNoInvoice")
    @NewAction("Invoice.addOrders") // Define nuestra propia acción para añadir pedidos
    private Collection<Order> orders;
    ...
}
```

De esta forma tan sencilla definimos la acción a ejecutar cuando el usuario pulsa en el botón con el signo más (+) para añadir pedidos. El argumento usado para @NewAction, Invoice.addOrders, es el nombre calificado de la acción, es decir la acción addOrders del controlador Invoice tal como se ha definido en el archivo *controllers.xml*.

Ahora hemos de editar *controllers.xml* para añadir el controlador Invoice (todavía no existe) con nuestra acción. El listado 12.12 muestra la definición del controlador.

Listado 12.12 Declaración de la acción Invoice.addOrders en controllers.xml

```
<controller name="Invoice">
    <extends controller="Invoicing"/>

    <action name="addOrders"
        class="org.openxava.invoicing.actions.GoAddOrdersToInvoiceAction"
        hidden="true" image="images/create_new.gif"/>
```

```

<!--
hidden="true": No se mostrará en la barra de botones del módulo
image="images/create_new.gif": La misma imagen que la acción estándar
-->

</controller>

```

El listado 12.13 muestra el código de la acción.

Listado 12.13 Acción personalizada para ir a “añadir pedidos” desde una factura

```

package org.openxava.invoicing.actions; // En el paquete 'actions'

import org.openxava.actions.*; // Para usar GoAddElementsToCollectionAction

public class GoAddOrdersToInvoiceAction
    extends GoAddElementsToCollectionAction { // Lógica estándar para ir a la lista que
                                                // permite añadir elementos a la colección
    public void execute() throws Exception {
        super.execute(); // Ejecuta la lógica estándar, la cual muestra un diálogo
        int customerNumber =
            getPreviousView().getValueInt("customer.number"); // Lee el número de cliente de la
                                                       // factura actual de la vista
        getTab().setBaseCondition( // La condición de la lista de pedidos a añadir
            "${customer.number} = " + customerNumber +
            " and ${delivered} = true and ${invoice.oid} is null"
        );
    }
}

```

Fíjate como usamos `getTab().setBaseCondition()` para establecer la condición de la lista para escoger la entidades a añadir. Es decir, desde una `GoAddElementsToCollectionAction` puedes usar `getTab()` para manipular la forma en que la lista se comporta.

12.2.3 Refinar la acción que añade elementos a la colección

Una mejora interesante para la colección de pedidos sería que cuando el usuario añada pedidos a la factura actual, las líneas de detalle de estos pedidos se copien automáticamente a la factura.

No podemos usar `@NewAction` para esto, porque es la acción que muestra la lista de elementos a añadir a la colección. Pero no es la acción que añade los elementos. En esta sección aprenderemos como definir la acción que realmente añade los elementos (figura 12.10).

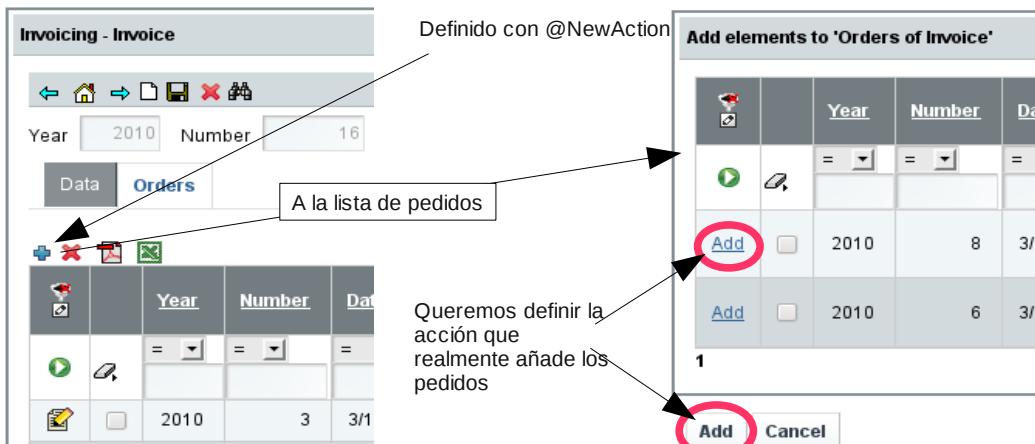


Figura 12.10 Queremos refinar la acción 'Add' en la lista de pedidos

Por desgracia, no hay una anotación para definir directamente esta acción de añadir. Sin embargo, no es una tarea demasiado difícil, solo hemos de refinar la acción @NewAction instruyéndola para mostrar nuestro propio controlador, y en este controlador podemos poner las acciones que queramos. Dado que ya hemos definido nuestra @NewAction en la sección anterior solo hemos de añadir un nuevo método a la ya existente GoAddOrdersToInvoiceAction. El listado 12.14 muestra este método.

Listado 12.14 getNextController() añadido a GoAddOrdersToInvoiceAction

```
public class GoAddOrdersToInvoiceAction ... {
    ...
    public String getNextController() { // Añadimos este método
        return "AddOrdersToInvoice"; // El controlador con las acciones disponibles en
        // la lista de pedidos a añadir
    }
}
```

Por defecto las acciones en la lista de entidades a añadir (los botones 'Add' y 'Cancel') son del controlador estándar de OpenXava AddToCollection. Sobrescribir getNextController() en nuestra acción nos permite definir nuestro propio controlador en su lugar. El listado 12.15 muestra la definición de nuestro controlador propio para añadir elementos en controllers.xml.

Listado 12.15 Controlador personalizado para añadir pedidos a la factura

```
<controller name="AddOrdersToInvoice">
    <extends controller="AddToCollection"/> <!-- Extiende del controlador estándar -->
    <!-- Sobrescribe la acción para añadir -->
    <action name="add"
```

```

    class="org.openxava.invoicing.actions.AddOrdersToInvoiceAction"/>
</controller>
```

De esta forma la acción para añadir pedidos a la factura será AddOrdersToInvoiceAction. Recuerda que el objetivo de nuestra acción es añadir los pedidos a la factura de la manera convencional, pero también copiar las líneas de estos pedidos a la factura. El listado 12.16 muestra el código de la acción.

Listado 12.16 Acción personalizada para añadir pedidos a una factura

```

package org.openxava.invoicing.actions; // En el paquete 'actions'

import java.rmi.*;
import java.util.*;
import javax.ejb.*;
import org.openxava.actions.*; // Para usar AddElementsToCollectionAction
import org.openxava.invoicing.model.*;
import org.openxava.model.*;
import org.openxava.util.*;
import org.openxava.validators.*;

public class AddOrdersToInvoiceAction
    extends AddElementsToCollectionAction { // Lógica estándar para añadir
                                            // elementos a la colección
    public void execute() throws Exception {
        super.execute(); // Usamos la lógica estándar "tal cual"
        getView().refresh(); // Para visualizar datos frescos, incluyendo los importes
                            // recalculados, que dependen de las líneas de detalle
    }

    protected void associateEntity(Map keyValues) // El método llamado para asociar
        throws ValidationException,           // cada entidad a la principal, en este caso para
            XavaException, NotFoundException, // asociar cada pedido a la factura
            FinderException, RemoteException
    {
        super.associateEntity(keyValues); // Ejecuta la lógica estándar (1)
        Order order = (Order) MapFacade.findEntity("Order", keyValues); // (2)
        order.copyDetailsToInvoice(); // Delega el trabajo principal en la entidad (3)
    }
}
```

Sobrescribimos el método `execute()` sólo para refrescar la vista después del proceso. Realmente, lo que nosotros queremos es refinar la lógica de asociar un pedido a la factura. La forma de hacer esto es sobrescribiendo el método `associateEntity()`. La lógica aquí es simple, después de ejecutar la lógica estándar (1) buscamos la entidad `Order` correspondiente y entonces llamamos al método `copyDetailsToInvoice()` de ese `Order`.

Obviamente, necesitamos tener un método `copyDetailsToInvoice()` en la entidad `Order`. El listado 12.17 muestra este método.

Listado 12.17 Método copyDetailsInvoice() en la clase Order

```
public class Order extends CommercialDocument {

    ...

    public void copyDetailsToInvoice() {
        copyDetailsToInvoice(getInvoice()); // Delegamos en un método ya existente
    }

}
```

Por suerte ya teníamos un método para copiar detalles desde una entidad Order a la Invoice especificada, simplemente llamamos a este método enviando la Invoice del Order.

Estas pequeñas modificaciones al comportamiento de la colección orders de Invoice son suficientes para convertir el módulo de Invoice en una herramienta efectiva para facturar clientes individualmente. Solo has de crear una factura nueva, escoger un cliente y añadir pedidos. Es incluso más fácil de usar que el modo lista del módulo Order (desarrollamos una acción para hacerlo en la sección 11.2) ya que el módulo Invoice solo se muestran los pedidos adecuados al cliente.

12.3 Pruebas JUnit

Todavía tenemos la sana costumbre de hacer un poco de código de aplicación, y después un poco de código de pruebas. Y ahora es el tiempo de escribir el código de pruebas para las nuevas características añadidas en este capítulo.

12.3.1 Adaptar OrderTest

Si ejecutaras OrderTest ahora, no pasaría. Esto es porque nuestro código confía en ciertos detalles que han cambiado. Por lo tanto, hemos de modificar nuestro código de pruebas actual. Edita el método testSetInvoice() de OrderTest y aplica los cambios mostrados en el listado 12.18.

Listado 12.18 Modificaciones en el método testSetInvoice() de OrderTest

```
public void testSetInvoice() throws Exception {
    ...

    assertEquals("invoice.number", "");
    assertEquals("invoice.year", "");
    execute("Reference.search", // Ya no usamos la acción estándar para
            "keyProperty=invoice.year"); // buscar la factura, en su lugar
    execute("Order.searchInvoice", // usamos nuestra acción personalizada (1)
```

```

    "keyProperty=invoice.number");
execute("List.orderBy", "property=number");

...
// Restaurar valores
setValue("delivered", "false");
setValue("invoice.year", ""); // Ahora es necesario teclear el año
setValue("invoice.number", ""); // y el número para buscar la factura (2)
execute("CRUD.save");
assertNoErrors();
}

```

Recuerda que anotamos la referencia invoice en Order con @SearchAction("Order.searchInvoice") (sección 12.1.3), por tanto hemos de modificar la prueba para llamar a Order.searchInvoice (1) en vez de a Reference.search. En la sección 12.1.4 añadimos @SearchKey a year y number de CommercialDocument, por lo tanto nuestra prueba ha de indicar tanto year como number para obtener (o en este caso borrar) una factura (2). Por causa de esto último también hemos de modificar testCreateInvoiceFromOrder() de OrderTest como muestra el listado 12.19.

Listado 12.19 Modificaciones en testCreateInvoiceFromOrder() de OrderTest

```

public void testCreateInvoiceFromOrder() throws Exception {
    ...
// Restaurar el pedido para ejecutar la prueba la siguiente vez
setValue("invoice.year", ""); // Ahora es necesario teclear el año
setValue("invoice.number", ""); // y el número para buscar la factura (2)
assertValue("invoice.number", "");
assertCollectionRowCount("invoice.details", 0);
execute("CRUD.save");
assertNoErrors();
}

```

Después de estos cambios OrderTest tiene que pasar. Sin embargo, todavía nos queda probar la nueva funcionalidad del módulo Order.

12.3.2 Probar @SearchAction

En la sección 12.1.3 usamos @SearchAction en la referencia invoice de Order para mostrar en la lista de búsqueda solo facturas del cliente del pedido actual. El listado 12.20 muestra la prueba de esta funcionalidad.

Listado 12.20 Probar buscar factura desde pedido en OrderTest

```

public void testSearchInvoiceFromOrder() throws Exception {
execute("CRUD.new");
setValue("customer.number", "1"); // Si el cliente es 1...
execute("Sections.change", "activeSection=1");
execute("Order.searchInvoice", // ...cuando el usuario pulsa para escoger una factura...
}

```

```

    "keyProperty=invoice.number");
assertCustomerInList("1"); // ...sólo se muestran las facturas del cliente 1
execute("ReferenceSearch.cancel");
execute("Sections.change", "activeSection=0");
setValue("customer.number", "2"); // Y si el cliente es 2...
execute("Sections.change", "activeSection=1");
execute("Order.searchInvoice", // ...cuando el usuario pulsa para escoger una factura...
    "keyProperty=invoice.number");
assertCustomerInList("2"); // ...sólo se muestran las facturas del cliente 2
}

```

La parte más peligrosa es verificar la lista de facturas, este es el trabajo de `assertCustomerInList()` cuyo código puede verse en el listado 12.21.

Listado 12.21 Confirmar que los valores en la columna cliente son los esperados

```

private void assertCustomerInList(String customerNumber) throws Exception {
    assertListNotEmpty();
    int c = getListRowCount();
    for (int i=0; i<c; i++) { // Un bucle por todas las filas
        if (!customerNumber.equals(getValueInList(i, "customer.number"))) {
            fail("Customer in row " + i + // Si el cliente no es el esperado falla
                " is not of customer " + customerNumber);
        }
    }
}

```

Consiste en un bucle por todas las filas verificando el número de cliente.

12.3.3 Probar @OnChangeListener

En la sección 12.1.5 usamos `@SearchAction` en la referencia `invoice` de `Order` para asignar automáticamente el cliente de la factura escogida al pedido actual cuando el usuario todavía no tiene cliente, o para verificar que el cliente de la factura y del pedido coinciden, si el pedido ya tiene cliente. El listado 12.22 muestra el método de prueba en `OrderTest`.

Listado 12.22 Probar los eventos al cambiar la factura de un pedido

```

public void testOnChangeInvoice() throws Exception {
    execute("CRUD.new"); // Estamos creando un nuevo pedido
    assertEquals("customer.number", ""); // por tanto no tiene cliente todavía
    execute("Sections.change", "activeSection=1");
    execute("Order.searchInvoice", // Busca la factura usando una lista
        "keyProperty=invoice.number");

    execute("List.orderBy", "property=customer.number"); // Ordena por cliente
    String customer1Number = getValueInList(0, "customer.number"); // Memoriza...
    String invoiceYear1 = getValueInList(0, "year"); // ...los datos de la...
    String invoiceNumber1 = getValueInList(0, "number"); // ...primera factura
    execute("List.orderBy", "property=customer.number"); // Ordena por cliente
    String customer2Number = getValueInList(0, "customer.number"); // Memoriza...
    String customer2Name = getValueInList(0, "customer.name"); // ...los datos de...
        ...la última factura

    assertNotEquals("Must be invoices of different customer",

```

```

customer1Number, customer2Number); // Las 2 facturas memorizadas no son la misma

execute("ReferenceSearch.choose", "row=0"); // La factura se escoge con la lista (1)
execute("Sections.change", "activeSection=0");
assertValue("customer.number", customer2Number); // Los datos del cliente
assertValue("customer.name", customer2Name); // se rellenan automáticamente (2)

execute("Sections.change", "activeSection=1");
setValue("invoice.year", invoiceYear1); // Tratamos de poner una factura de...
setValue("invoice.number", invoiceNumber1); // ...otro cliente (3)

assertError("Customer Nº " + customer1Number + " of invoice " + // Muestra...
    invoiceYear1 + "/" + invoiceNumber1 + // ...un mensaje de error... (4)
    " does not match with Customer Nº " +
    customer2Number + " of the current order");

assertValue("invoice.year", ""); // ...y reinicia los datos de la factura (5)
assertValue("invoice.number", "");
assertValue("invoice.date", "");
}

```

Aquí probamos que nuestra acción on-change rellene los datos del cliente (3) al escoger una factura (2), y que si el cliente ya está establecido se muestre un mensaje de error (4) y la factura se borre de la vista (5). Fíjate como la primera vez usamos la lista (1) para escoger la factura y la segunda lo hacemos tecleando el año y el número (3).

12.3.4 Adaptar InvoiceTest

Como en el caso de OrderTest, InvoiceTest también falla. Has de hacer unos pequeños ajustes para que funcione. Edita testAddOrders() de InvoiceTest y aplica los cambios del listado 12.23.

Listado 12.23 Modificaciones en el método testAddOrders() de InvoiceTest

```

public void testAddOrders() throws Exception {
    assertListNotEmpty();
    execute("List.orderBy", "property=number");
    execute("Mode.detailAndFirst");
    execute("Sections.change", "activeSection=1");
    assertCollectionRowCount("orders", 0);
    execute("Collection.add"); // La acción estándar para añadir pedidos ya no se usa
    execute("Invoice.addOrders", // En su lugar usamos nuestra propia acción
        "viewObject=xava_view_section1_orders");
    checkFirstOrderWithDeliveredEquals("Yes"); // Ahora todos los pedidos de la lista
    checkFirstOrderWithDeliveredEquals("No"); // están entregados; esto ya no hace falta

    execute("AddToCollection.add"); // En lugar de la acción estándar
    execute("AddOrdersToInvoice.add", "row=0"); // ...ahora tenemos la nuestra propia
    assertError("ERROR! 1 element(s) NOT added to Orders of Invoice"); // Es
        // imposible porque el usuario no puede escoger pedidos incorrectos
    assertMessage("1 element(s) added to Orders of Invoice");
    assertCollectionRowCount("orders", 1);
    checkRowCollection("orders", 0);
    execute("Collection.removeSelected",

```

```

    "viewObject=xava_view_section1_orders");
    assertCollectionRowCount("orders", 0);
}

```

Ya no necesitamos el método `checkFirstOrderWithDeliveredEquals()`, por tanto podemos quitarlo de `InvoiceTest` (listado 12.24).

Listado 12.24 Quitar `checkFirstOrderWithDeliveredEquals()` de `InvoiceTest`

```

private void checkFirstOrderWithDeliveredEquals(String value)
    throws Exception { ... }

```

Después de estos cambios `InvoiceTest` ha de funcionar. Sin embargo, todavía nos queda probar la nueva funcionalidad del módulo `Invoice`.

12.3.5 Probar `@NewAction`

En la sección 12.2.2 anotamos la colección `orders` de `Invoice` con `@NewAction` para refinar la lista de pedidos a ser añadidos a la colección. De esta forma solo los pedidos entregados del cliente de la factura actual y todavía sin facturar se mostraban. Vamos a probar esto, y al mismo tiempo, aprenderemos como refactorizar el código existente para poder reutilizarlo.

Primero queremos verificar que la lista para añadir pedidos solo contiene pedidos del cliente actual. El listado 12.25 muestra los cambios en `testAddOrders()` para conseguir esto.

Listado 12.25 Verificar que todos los pedidos en la lista son del cliente actual

```

public void testAddOrders() throws Exception {
    assertListNotEmpty();
    execute("List.orderBy", "property=number");
    execute("Mode.detailAndFirst");
    String customerNumber = getValue("customer.number"); // Tomamos nota del
    execute("Sections.change", "activeSection=1");           // cliente de la factura
    assertCollectionRowCount("orders", 0);
    execute("Invoice.addOrders",
        "viewObject=xava_view_section1_orders");
    assertCustomerInList(customerNumber); // Confirmamos que todos los cliente en
                                         // la lista coinciden con el cliente de la factura
    ...
}

```

Ahora hemos de escribir el método `assertCustomerInList()`. Pero, espera un momento, ya hemos escrito este método en `OrderTest`. Lo vimos en el listado 12.34 (sección 12.3.2). Estamos en `InvoiceTest` por tanto no podemos llamar a este método. Por fortuna tanto `InvoiceTest` como `OrderTest` heredan de `CommercialDocumentTest`, por lo tanto sólo tenemos que subir el método a la clase madre. Para hacer esto copia el método `assertCustomerInList()` desde

245 Capítulo 12: Referencias y colecciones

OrderTest a CommercialDocumentTest, cambiando private por protected, tal como muestra el listado 12.26.

Listado 12.26 assertCustomerInList() movido a CommercialDocumentTest

```
abstract public class CommercialDocumentTest extends ModuleTestBase {  
  
    private protected void // Cambiamos de private a protected  
        assertCustomerInList(String customerNumber) throws Exception {  
  
        ...  
    }  
  
    ...  
}
```

Ahora puedes quitar el método assertCustomerInList() de OrderTest (listado 12.27).

Listado 12.27 assertCustomerInList() quitado de OrderTest

```
public class OrderTest extends CommercialDocumentTest {  
  
    private void assertCustomerInList(String customerNumber)  
        throws Exception { ... }  
  
    ...  
}
```

Después de estos cambios el método testAddOrders() compila y funciona.

No solo queremos comprobar que la lista de pedidos son del cliente correcto, sino también que están entregados. Nuestro primer impulso es copiar y pegar assertCustomerInList() para crear un método assertDeliveredInList(). Sin embargo, resistimos la tentación, y en vez de eso vamos a crear un método reutilizable. Primero, copiamos y pegamos assertCustomerInList() como assertValueForAllRows(). El listado 12.28 muestra estos dos métodos en CommercialDocumentTest.

Listado 12.28 Crear assertValueForAllRows() a partir de assertCustomerInList()

```
protected void assertCustomerInList(String customerNumber) throws Exception {  
    assertListNotEmpty();  
    int c = getListRowCount();  
    for (int i=0; i<c; i++) {  
        if (!customerNumber.equals(  
            getValueInList(i, "customer.number")))  
            // Preguntamos por el cliente  
            // de forma fija  
        {  
            fail("Customer in row " + i + " is not of customer "  
                + customerNumber);  
        }  
    }  
}
```

```
protected void assertValueForAllRows(int column, String value)
    throws Exception
{
    assertListNotEmpty();
    int c = getListRowCount();
    for (int i=0; i<c; i++) {
        if (!value.equals(
            getValueInList(i, column))) // Preguntamos por la columna
        {
            fail("Column " + column + " in row " + i + " is not " + value);
        }
    }
}
```

Puedes ver como con unas ligeras modificaciones hemos convertido `assertCustomerInList()` en un método genérico para preguntar por el valor de cualquier columna, no solo por la del número de cliente. Ahora hemos de quitar el código redundante, puedes, bien quitar `assertCustomerInList()` o bien reimplementarlo usando el nuevo método. El listado 12.29 muestra la última opción.

Listado 12.29 Reimplementar `assertCustomerInList()` llamando al nuevo método

```
protected void assertCustomerInList(String customerNumber) throws Exception {
    assertValueForAllRows(3, customerNumber); // Número de cliente está en la columna 3
}
```

Usemos `assertValueForAllRows()` para verificar que la lista de pedidos contiene solo pedidos entregados. El listado 12.30 muestra la modificación necesaria en `testAddOrders()` de `InvoiceTest`.

Listado 12.30 Confirmar que todos los pedidos están entregados

```
public void testAddOrders() throws Exception {
    assertListNotEmpty();
    execute("List.orderBy", "property=number");
    execute("Mode.detailAndFirst");
    String customerNumber = getValue("customer.number");
    execute("Sections.change", "activeSection=1");
    assertCollectionRowCount("orders", 0);
    execute("Invoice.addOrders",
        "viewObject=xava_view_section1_orders");
    assertCustomerInList(customerNumber);
    assertValueForAllRows(5, "Yes"); // Todas las celdas de la columna 5 (delivered)
                                    // tienen 'Yes'
    ...
}
```

Además, queremos que solo los pedidos sin factura se muestren en la lista. Una forma sencilla de hacerlo es verificando que después de añadir un pedido a la factura actual, la lista de pedidos tenga una entrada menos. El listado 12.31 muestra los cambios necesarios en `testAddOrders()` para hacer esto.

Listado 12.31 Probar que los pedidos ya añadidos no pueden añadirse otra vez

```

public void testAddOrders() throws Exception {
    ...

    assertCustomerInList(customerNumber);
    assertEqualsForAllRows(5, "Yes");
    int ordersRowCount = getListRowCount(); // Tomamos nota de la cantidad de pedidos
    execute("AddOrdersToInvoice.add", "row=0"); // cuando se muestra la lista
    assertMessage("1 element(s) added to Orders of Invoice");
    assertCollectionRowCount("orders", 1); // Se añadió un pedido
    execute("Invoice.addOrders", // Mostramos la lista de pedidos otra vez
        "viewObject=xava_view_section1_orders");
    assertEqualsForRowCount(ordersRowCount - 1); // Tenemos un pedido menos en la lista
    execute("AddToCollection.cancel");

    ...
}
```

Con el código de esta sección hemos probado la @NewAction de la colección orders, y al mismo tiempo hemos visto como no es necesario crear código genérico desde el principio, porque no es difícil convertir el código concreto en genérico bajo demanda.

12.3.6 Probar la acción para añadir elementos a la colección

En la sección 12.2.3 aprendimos como refinar la acción para añadir pedidos a la factura, ahora es el momento de escribir su correspondiente código de prueba. Recuerda que esta acción copia las líneas de los pedidos seleccionados a la factura actual. El listado 12.32 muestra los cambios para probar nuestra acción personalizada para añadir pedidos.

Listado 12.32 Cuando un pedido se añade sus líneas se añaden a la factura

```

public void testAddOrders() throws Exception {
    ...

    String customerNumber = getValue("customer.number");
    deleteDetails(); // Borra la líneas de detalle si las hay (1)
    assertEqualsForRowCount("details", 0); // Ahora la factura no tiene detalles
    assertEquals("baseAmount", "0.00"); // Sin detalles el importe base es 0
    execute("Sections.change", "activeSection=1");
    assertEqualsForRowCount("orders", 0);
    execute("Invoice.addOrders", // Cuando mostramos la lista de pedidos (2) ...
        "viewObject=xava_view_section1_orders");
    assertCustomerInList(customerNumber);
    assertEqualsForAllRows(5, "Yes");
    String firstOrderBaseAmount = getValueInList(0, 8); // tomamos nota del importe
    int ordersRowCount = getListRowCount(); // base del primer pedido de la lista (3)
    ...
    assertEqualsForRowCount("orders", 1);
    execute("Sections.change", "activeSection=0");
```

```

assertCollectionNotEmpty("details"); // Hay detalles, han sido copiados (4)
assertEquals("baseAmount", firstOrderBaseAmount); // El importe base de la factura
execute("Sections.change", "activeSection=1"); // coincide con el del
                                                // pedido recién añadido (5)
...
}

}

```

Quitamos las líneas de detalle de la factura (1), después añadimos un pedido (2), tomando nota de su importe base (3), entonces verificamos que la factura actual tiene detalles (4) y que su importe base es el mismo que el del pedido añadido (5).

Nos queda el método `deleteDetails()`, mostrado en el listado 12.33.

Listado 12.33 Borra todos los detalles de la factura visualizada

```

private void deleteDetails() throws Exception {
    int c = getCollectionRowCount("details");
    for (int i=0; i<c; i++) { // Un bucle por todas las filas
        checkRowCollection("details", i); // Marca cada fila
    }
    execute("Collection.removeSelected", // Borra las filas marcadas
           "viewObject=xava_view_section0_details");
}

```

Selecciona todas las filas de la colección `details` y pulsa en el botón 'Remove selected'.

El método `testAddOrders()` está acabado. Puedes ver su código definitivo en el listado 12.34.

Listado 12.34 Código definitivo para `testAddOrders()` de `InvoiceTest`

```

public void testAddOrders() throws Exception {
    assertListNotEmpty();
    execute("List.orderBy", "property=number");
    execute("Mode.detailAndFirst");
    String customerNumber = getValue("customer.number");
    deleteDetails();
    assertCollectionRowCount("details", 0);
    assertEquals("baseAmount", "0.00");
    execute("Sections.change", "activeSection=1");
    assertCollectionRowCount("orders", 0);
    execute("Invoice.addOrders",
           "viewObject=xava_view_section1_orders");
    assertCustomerInList(customerNumber);
    assertEqualsForAllRows(5, "Yes");
    String firstOrderBaseAmount = getValueInList(0, 8);
    int ordersRowCount = getListRowCount();
    execute("AddOrdersToInvoice.add", "row=0");
    assertEquals("1 element(s) added to Orders of Invoice");
    assertCollectionRowCount("orders", 1);
    execute("Sections.change", "activeSection=0");
    assertCollectionNotEmpty("details");
    assertEquals("baseAmount", firstOrderBaseAmount);
}

```

```

execute("Sections.change", "activeSection=1");
execute("Invoice.addOrders",
       "viewObject=xava_view_section1_orders");
assertListRowCount(ordersRowCount - 1);
execute("AddToCollection.cancel");
checkRowCollection("orders", 0);
execute("Collection.removeSelected",
       "viewObject=xava_view_section1_orders");
assertCollectionRowCount("orders", 0);
}

```

Hemos finalizado el código de las pruebas automáticas. Ahora puedes ejecutar todas las pruebas de tu proyecto. Han de salir en color verde.

12.4 Resumen

Este capítulo te ha mostrado como refinar el comportamiento estándar de las referencias y colecciones para que tu aplicación se adapte a las necesidades del usuario. Aquí sólo has visto algunos ejemplos ilustrativos. OpenXava ofrece muchas más posibilidades para refinar el comportamiento de las colecciones y referencias, con anotaciones como @ReferenceView, @Readonly, @NoFrame, @NoCreate, @NoModify, @NoSearch, @AsEmbedded, @SearchAction, @DescriptionsList, @LabelFormat, @Action, @OnChange, @OnChangeSearch, @Editor, @CollectionView, @EditOnly, @ListProperties, @RowStyle, @EditAction, @ViewAction, @NewAction, @SaveAction, @HideDetailAction, @RemoveAction, @RemoveSelectedAction, @ListAction, @DetailAction o @OnSelectElementAction.

Y por si esto fuera poco, siempre tienes la opción definir tu propio editor¹⁷ para referencias o colecciones. Los editores te permiten crear una interfaz de usuario personalizada para visualizar y editar la referencia o colección.

Esta flexibilidad te permite usar la generación automática de la interfaz gráfica para prácticamente cualquier caso posible en las aplicaciones de gestión de la vida real.

¹⁷ Ver la sección 1.3.1 del capítulo 1

*Seguridad
y navegación
con Liferay*

capítulo 13

Algo importante falta en tu aplicación. Hasta el momento tienes módulos individuales, como Invoice, Delivery, Product o Author. Sin embargo, tu aplicación no está lista para los usuarios. Necesita navegación, es decir, una forma en la que el usuario pueda buscar el módulo que quiera, y aun más importante, necesita seguridad. El usuario tiene que identificarse antes de usar la aplicación, y las opciones disponibles dependerán de su perfil.

OpenXava no incluye seguridad y navegación, sin embargo, añadir seguridad y navegación a una aplicación OpenXava es facilísimo si usamos un portal Java. Vamos a desplegar tu aplicación Invoicing en un popular portal de código abierto, Liferay. De esta forma conseguirás seguridad, navegación, varios niveles de acceso y algunas cosas más que aprenderás en este capítulo.

13.1 Introducción a los portales Java

El objetivo de un Portal Empresarial es proveer un punto de entrada único y una gestión de usuarios unificada para todas las aplicaciones de una organización. Para poder ser desplegada en un portal, una aplicación tiene que proveer portlets. Los portlets se presentan visualmente en cajas independientes. La figura 13.1 muestra una página de un portal con tres portlets.

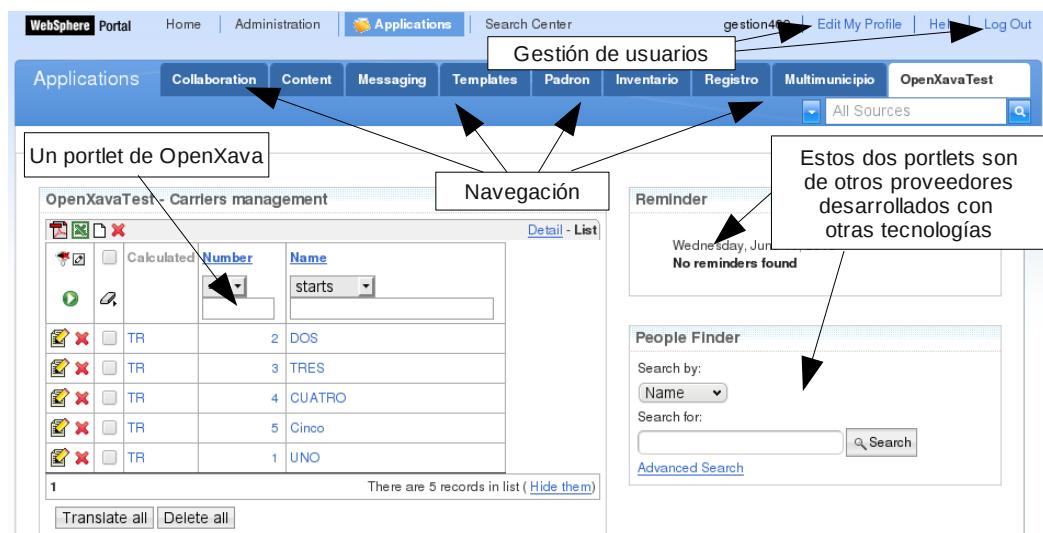


Figura 13.1 Página de un portal con varios portlets en IBM WebSphere Portal

Esta figura ilustra la magia tras los portales. Aquí tienes, en la misma página, tres portlets de diferentes aplicaciones, desarrollados por diferente proveedores y usando tecnologías diferentes. Todos con la misma apariencia y usando la misma gestión de usuarios y estructura de navegación. El portal no solo te provee navegación y seguridad sin coste de desarrollo, sino también te permite integrar

tu aplicación con otras con total naturalidad.

Para poder integrar una aplicación dentro de un portal, la aplicación ha de definir los portlets siguiendo ciertas normas. Estas normas están definidas en *La especificación de Portlet*, un estándar Java llamado JSR-168 (v1.0)¹⁸ y JSR-286 (v2.0)¹⁹. Prácticamente todos los portales Java soportan esta especificación (v1.0 o v2.0) por tanto puedes desplegar tu aplicación de portlets en una larga lista de portales tanto comerciales como de código abierto.

Las aplicaciones OpenXava son aplicaciones de portlets estándar por defecto²⁰.

13.2 Instalar Liferay

Liferay es un portal de código abierto muy popular, quizás por su alta calidad o porque incluye un nutrido conjunto de portlets bastante útiles y listos para usar. Con Liferay dispones de partida de portlets para gestión de contenido, foros, blogs, etc. Liferay no es un mero CMS sino un portal Java empresarial que soporta JSR-168/286, por lo tanto podemos usarlo como la infraestructura de seguridad y navegación para nuestra aplicación.

El primer paso es descargarlo. Ve a www.liferay.com, cambia la sección “Downloads” y descarga “Liferay Portal Community Edition”²¹. Obtendrás un archivo llamado *liferay-portal-tomcat-6.0.5.zip*. Descomprime el archivo, tendrás una carpeta *liferay-portal-6.0.5*. Esta carpeta contiene un Liferay listo para funcionar.

No obstante, todavía necesitamos ajustar algunos pequeños detalles para ejecutar las aplicaciones OpenXava dentro de él. Primero, copia el archivo *context.xml* desde tu carpeta *openxava-4.0/workspace/Servers/Tomcat v6.0 Server at localhost-config* a la carpeta *liferay-portal-6.0.5/tomcat-6.0.26/conf*. El archivo *context.xml* contiene la definición de las fuentes de datos.

Ahora copia el archivo *ejb.jar* desde *openxava-4.0/tomcat/lib* a la carpeta *liferay-portal-6.0.5/tomcat-6.0.26/lib*.

Para arrancar Liferay ve a *liferay-portal-6.0.5/tomcat-6.0.26/bin* y ejecuta *startup.bat* (Windows) o *startup.sh* (Unix). Espera 2 o 3 minutos, después tu navegador abrirá automáticamente la página de inicio del portal, si no ocurriera así, abre tu navegador y ve a <http://localhost:8080/>. Lo que ves debería parecerse a la figura 13.2.

18 <http://jcp.org/en/jsr/detail?id=168>

19 <http://jcp.org/en/jsr/detail?id=286>

20 OpenXava genera aplicaciones JSR-168 desplegables en portales JSR-168 y JSR-286

21 Aunque algunos pantallazos de este capítulo son de Liferay 5.2, todas las instrucciones funcionan con Liferay 6.0

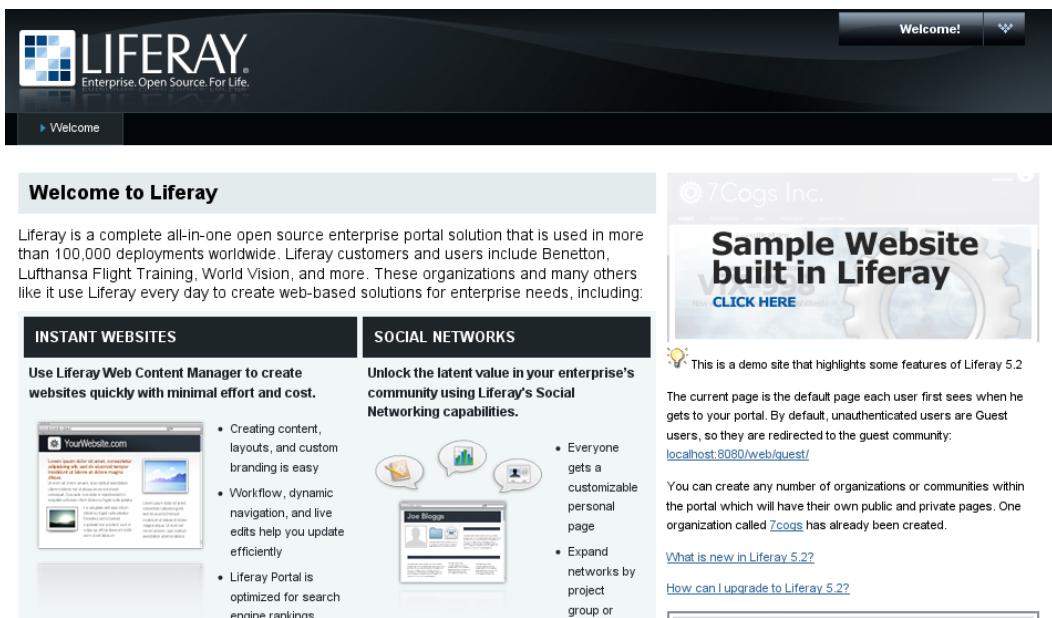


Figura 13.2 Página de bienvenida de Liferay

¡Enhorabuena! Tienes el Liferay instalado y ejecutándose en tu máquina. Despleguemos nuestra aplicación en él.

13.3 Desplegar nuestra aplicación en Liferay

El primer paso es generar una aplicación (un archivo war) preparada para ser desplegada en un portal. Para ello sigue las instrucciones de la figura 13.3.

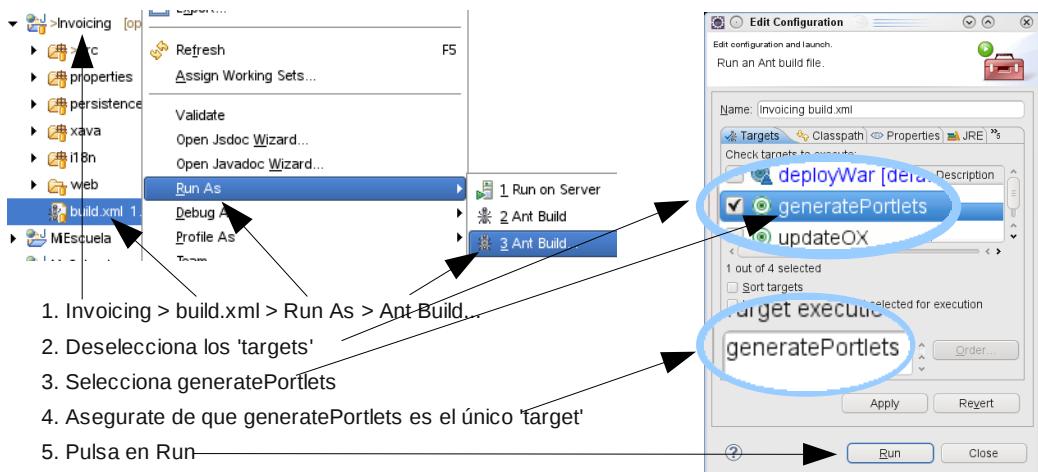


Figura 13.3 Ejecutar tarea ant generatePortlets

La tarea ant `generatePortlets` ha generado un archivo *Invoice.war* en la

carpeta `workspace.dist/Invoicing.dist`. Para desplegarlo en Liferay copia `Invoice.war` a `liferay-portal-6.0.5/deploy` y espera hasta que desaparezca.

Tu aplicación ya está desplegada, por lo tanto todos sus módulos son portlets listos para ser añadidos a cualquier página del Liferay. Intentemos añadir uno de ellos. Has de identificarte usando un usuario con permiso para añadir portlets a sus páginas. Puedes usar 'bruno', un usuario administrador incluido por defecto con Liferay. La figura 13.4 muestra el portlet 'Current Users'. Este portlet está en la página de bienvenida y te permite identificarte como bruno fácilmente. Pulta en el vínculo 'Login as bruno'.

Ahora añade un portlet de tu aplicación Invoicing siguiendo las instrucciones de la figura 13.5.



Figura 13.4 Identifícate como 'bruno'

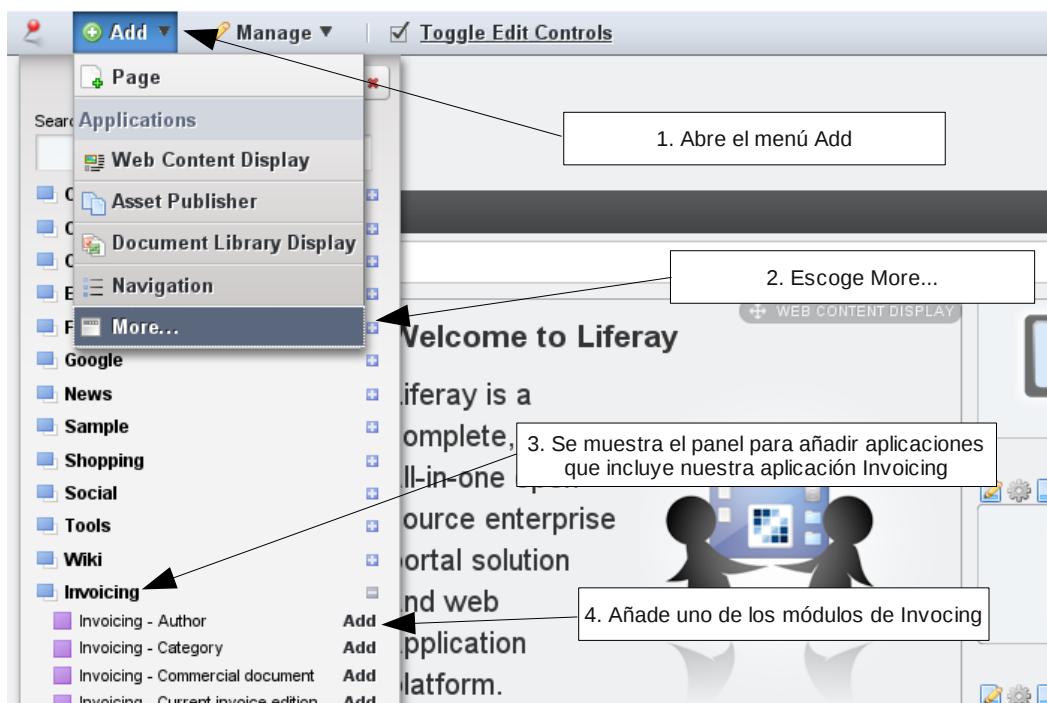


Figura 13.5 Añadir un portlet de la aplicación Invoicing a la página actual

Después de añadir el portlet deberías verlo en acción dentro de la página, si no es así, recarga la página. La figura 13.6 muestra la apariencia de la página con el nuevo portlet.



Figura 13.6 Página de Liferay con un portlet OpenXava

Has podido comprobar lo fácil que es generar una aplicación de portlets con OpenXava, y después desplegarla en Liferay. Una vez en Liferay, el usuario puede añadir los módulos OpenXava a sus páginas, creando así un espacio de trabajo que se ajuste a sus necesidades y preferencias personales.

13.4 Navegación

La opción de tener la aplicación dividida en piezas para que el usuario final pueda componer sus propias páginas es muy interesante, y válida para algunos escenarios relacionados con usuarios avanzados. Sin embargo, en una aplicación de gestión convencional, queremos que el usuario acceda a una página bien organizada con vínculos a los módulos que necesita para hacer su trabajo.

13.4.1 Añadir las páginas para la aplicación y sus módulos

Por defecto cuando entras en Liferay solo tienes una página 'Welcome'. Vamos a crear una nueva página para tu aplicación Invocing. Si estás identificado como

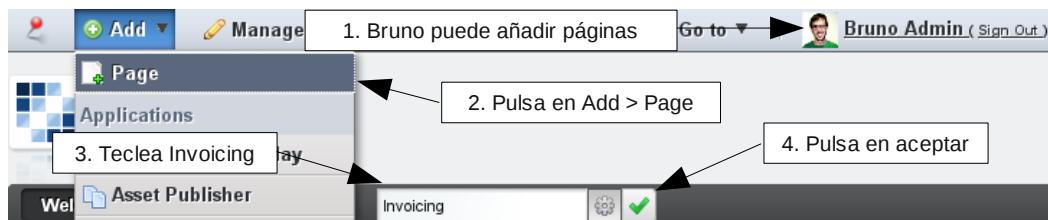


Figura 13.7 Añadir una nueva página

'bruno' tienes permisos para crear páginas nuevas. La figura 13.7 muestra los pasos necesarios para crear una nueva página. Después de eso tendrás una sección principal en el sitio llamada 'Invoicing'. Si pulsa en ella irás a una página vacía.

Sigue los pasos en la figura 13.8 para añadir páginas hijas.

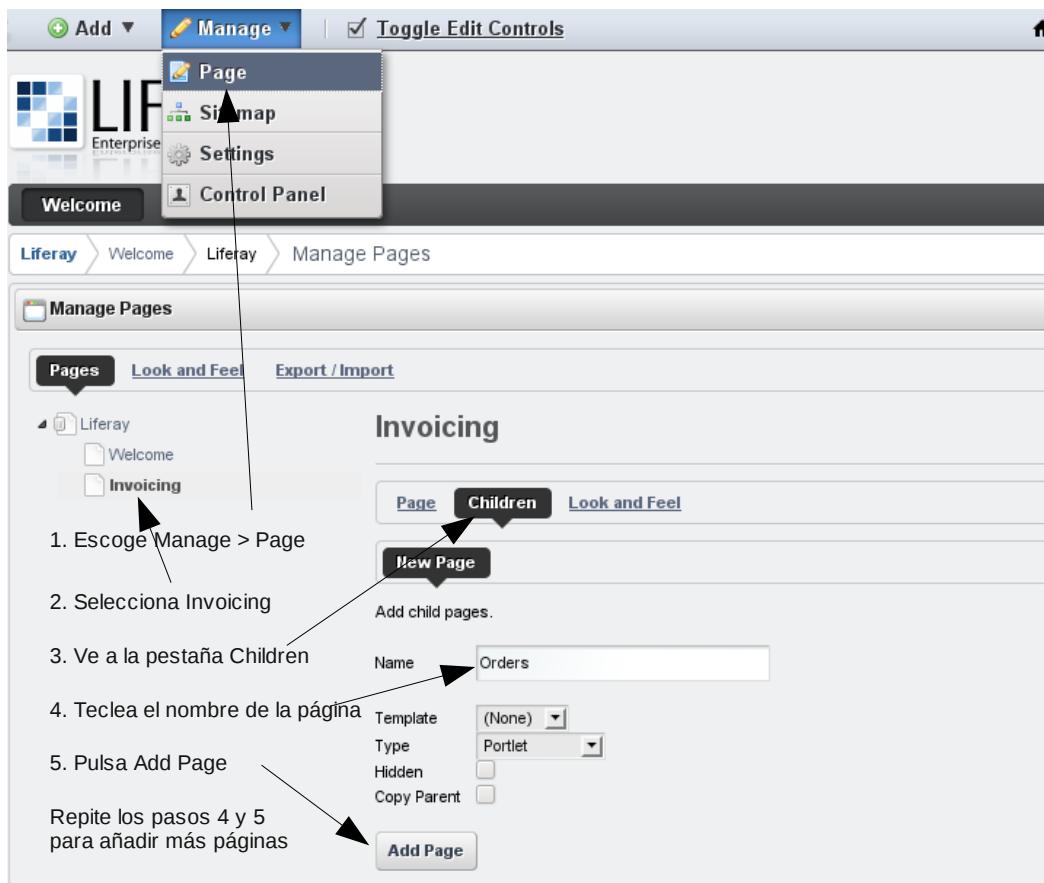


Figura 13.8 Añadir páginas hijas para los módulos

Siguiendo las instrucciones de la figura 13.8 añade las páginas: Orders, Invoices, Customers, Products, Categories, Authors y Trash. A partir de ahora, cuando se ponga el ratón sobre la sección Invoicing se mostrará un menú con las opciones que acabas de añadir. Puedes ver este menú en la figura 13.9. Aunque por ahora, todas estas opciones llevan a páginas vacías. En las siguientes secciones llenaremos estas páginas con los módulos correspondientes de la aplicación Invoicing.

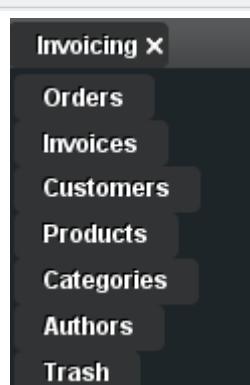


Figura 13.9 Menú

13.4.2 Llenar una página vacía con un Portlet

Nuestro trabajo ahora es llenar las páginas vacías con los portlets correspondientes. Empecemos con el portlet Order. La figura 13.10 muestra los pasos para añadir el portlet Order a la página Orders.



Figura 13.10 Añadir el portlet Order a la página Orders

A partir de ahora cuando el usuario pulse en el menú *Invoicing > Orders* verá el módulo para gestionar pedidos.

Usa esta técnica para añadir los portlets correspondientes a las páginas *Invoices* y *Customers*.

13.4.3 Añadir un menú de navegación

El camino de migas que Liferay añade por defecto a cada página es una herramienta de navegación bastante buena, sin embargo, algunos módulos ocupan poco, por tanto nos sobra espacio en la izquierda para usar una barra de navegación. Vamos a añadir un menú de navegación en la página *Customers*. Ve a la páginas *Customers* y usa la opción *Navigation* del menú *Add*, tal como muestra la figura 13.11.

De momento el resultado tiene el aspecto de un simple camino de migas. Hemos de cambiar su estilo de visualización para que se parezca más a un menú convencional. Sigue las instrucciones en la figura 13.12 para ello.

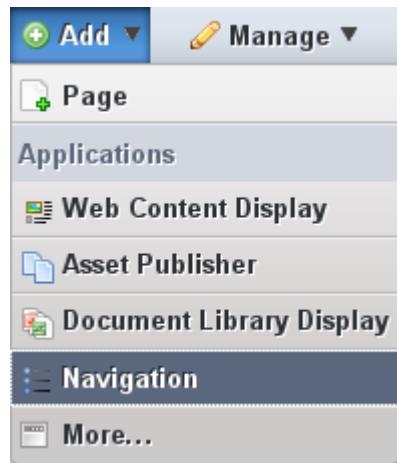


Figura 13.11 Opción para añadir navegación

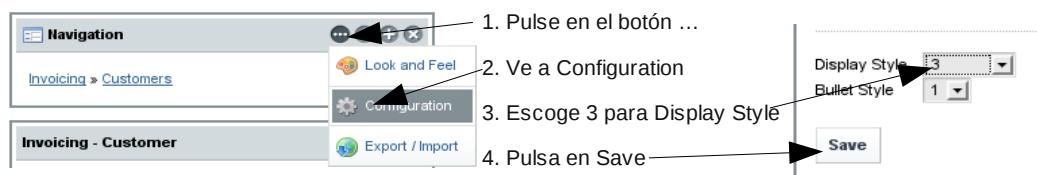


Figura 13.12 Cambiar el estilo de visualización del portlet Navigation

Escogemos 3 como estilo de visualización porque usa un formato de menú.

259 Capítulo 13: Seguridad y navegación con Liferay

Puedes experimentar con los otros formatos y escoger otro que te guste más. Después de escoger el formato de visualización solo has de dejar caer el portlet Customer en la parte derecha de la página para obtener el resultado de la figura 13.13.

The screenshot shows a Liferay page with a navigation bar at the top. The left sidebar contains a 'Navigation' portlet with a 'Invoicing' section and a list of links: Orders, Invoices, Customers, Products, Categories, Authors, and Trash. The main content area is titled 'Invoicing - Customer' and displays a table of customer data. The table has columns for Number and Name. There are three rows of data:

Number	Name
1	FRANCISCO JAVIER PANIZA LUCAS
2	Mª CARMEN GIMENO ALABAU
3	PEDRO PANIZA GIMENO

At the bottom of the table, it says 'There are 3 objects in list (Hide them)'. A 'Delete selected' button is also visible.

Figura 13.13 Página Customers con menú de navegación en la izquierda

La página Customers ya está lista. Hemos de repetir esta tarea para el resto de los portlets.

13.4.4 Copiar una página

Añadir el portlet Navigation, configurar su estilo de visualización y distribuir la disposición de los portlet “arrastrando y soltando” para todas las páginas que nos quedan es un trabajo largo y aburrido. Una forma de aliviar esta tarea tan repetitiva es copiando las páginas nuevas desde una ya existente. Vamos a configurar la página Products copiándola desde la ya configurada Customers. Para hacer esto, ve a la página Products, que de momento está vacía, y escoge la opción Manage Pages, tal como muestra la figura 13.14.

1. Ve a la página Products 2. Manage > Page

3. Escoge Customers para Copy Page
4. Pulsa en Save

Figura 13.14 Llenar la página Products copiando desde la página Customers

Ahora, tenemos la página configurada con un menú de navegación a la izquierda y un portlet Customer a la derecha. Sólo necesitamos cambiar Customer por Product, como se muestra en la figura 13.15.

Estamos en la Página Products

pero se muestra el módulo Customer 1. Quita el portlet Customer

2. Añade el portlet Product

Figura 13.15 Cambiar el portlet Customer por el portlet Product

Después de añadir el portlet Product puedes que necesites arrastrarlo y soltarlo en la parte derecha de la página. De esta forma has conseguido configurar tu página Products rápidamente.

Usa esta misma técnica para configurar las páginas Categories y Authors.

13.4.5 Dos módulos en la misma página

Solo nos quedan los módulos papelera. Recuerda que tenemos dos: InvoiceTrash y OrderTrash. Aunque, desde el punto de vista del usuario un único menú Trash sería mejor. Por lo tanto, vamos a añadir los dos módulos papelera en la misma página. El resultado sería como el de la figura 13.16.

Figura 13.16 Dos módulos OpenXava en la misma página

Si queremos conseguir el resultado de la figura 13.16 primero hemos de configurar la disposición de la página para que use tres columnas. Ve a la página Trash, actualmente vacía, y sigue las instrucciones de la figura 13.17 para establecer la disposición de la página.

Figura 13.17 Configurar la disposición para usar tres columnas

Ahora añade los portlets Navigation, InvoiceTrash y OrderTrash. Si es necesario muévelos para poner cada uno en su propia columna. En este punto seguro que ya eres un experto añadiendo portlets, por eso no ponemos aquí las capturas de pantalla, para no aburrirte demasiado.

Fíjate como mostramos solo tres columnas en cada módulo Trash, así podemos poner los portlets uno al lado del otro, y ver los tres portlets en la misma fila. Para configurar las columnas a mostrar puedes, bien usar los botones de personalización (figura 13.18) o bien definir



1. Pulsa customize
2. Pulsa remove

Figura 13.18 Quitar una columna de la lista

las propiedades usando la anotación @Tab en las definiciones de las entidades (listado 13.1).

Listado 13.1 @Tab modificado para mostrar tres propiedades en Invoice y Order

```
@Tab(name="Deleted", // El tab Deleted se usa para los módulos papelera
      baseCondition = "deleted = true",
      properties="year, number, date"
)
```

Has de modificar el @Tab Deleted para Invoice y Order. Si escoges esta opción, tienes que redesplegar la aplicación.

El punto importante aquí es que puedes arreglar la página a tu conveniencia, y esto incluye añadir varios portlets de OpenXava en la misma página.

13.4.6 Usar el CMS

Por fin tenemos todos los módulos en su sitio. No obstante, todavía tenemos una página vacía, la página principal de Invoicing. Una opción para llenar esta página es poner ahí una página de bienvenida. La figura 13.19²² muestra una posible opción para esta página.

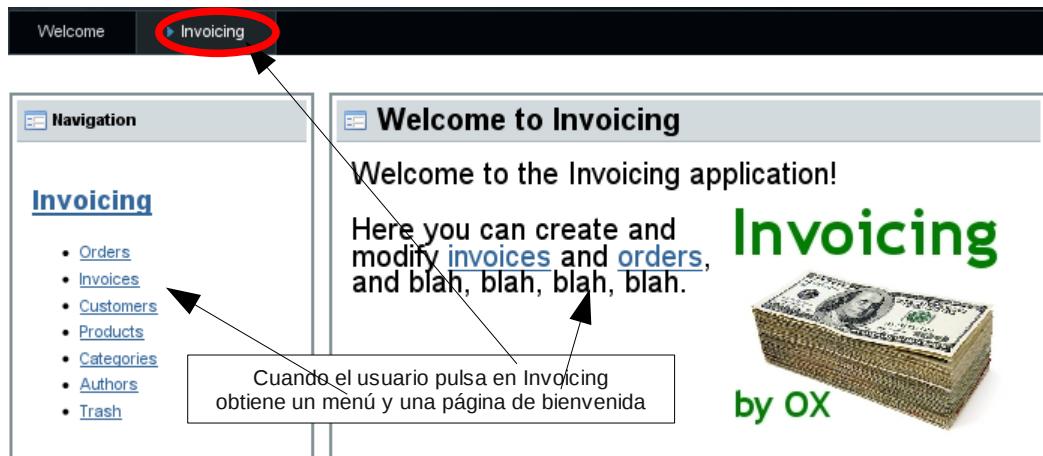


Figura 13.19 Página de bienvenida para Invoicing

Ya sabes como añadir el menú de navegación de la izquierda. Añadir el portlet Welcome con texto libre, vínculos e imágenes es pan comido gracias al CMS incluido con Liferay. La figura 13.20 muestra como añadir el portlet Web Content Display.

22 Imagen del dinero de <http://www.flickr.com/photos/amagill/3366720659/>

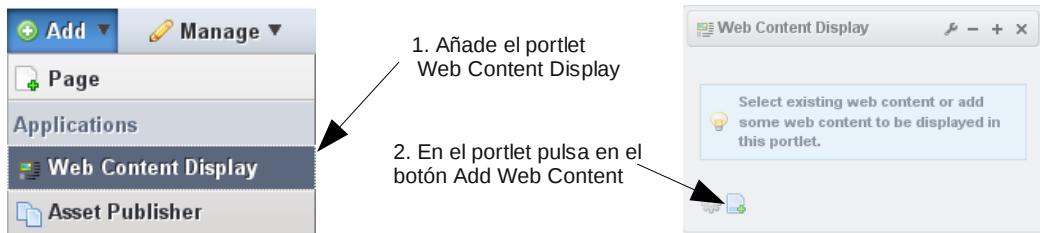


Figura 13.20 Añadir el portlet Web Content Display

Después de seguir las instrucciones de la figura 13.20 te encontrarás con un editor HTML que te permitirá añadir el contenido que deseas en tu página de bienvenida (figura 13.21).

La figura muestra el editor HTML para el portlet 'Invoicing Welcome'. Se observan los siguientes elementos:

- Encabezados:** Name: Invoicing Welcome, Language: English (United States), Default Language: English (United States).
- Editor de texto:** Ofrece herramientas para Estilo, Tamaño, Negrita, Cursiva, Subrayado, etc., así como iconos para imágenes y enlaces.
- Contenido:** El editor contiene el siguiente HTML:


```
<p>Wellcome to the Invoicing application!</p>
<p>Here you can create and modify <a href="#">invoices</a> and <a href="#">orders</a>, and blah, blah, blah, blah.</p>
<p>Blah, blah, blah.</p>
```
- Imagen:** Una imagen de un paquete de dólares estadounidenses.
- Botones de acción:** Save, Save and Continue, Save and Approve, Publish (destacado en azul).
- Etiquetas:** 1. Escribe un nombre para este contenido web, 2. Diseña el contenido usando texto, vínculos, imágenes o cualquier otra cosa que HTML permita, 3. Pulsa en Publish.

Figura 13.21 Crear el contenido del portlet de bienvenida

Si el botón Publish no está presente, ve al Control Panel, y escoge la opción Workflow Configuration; entonces pon el valor de Web Content value a No workflow.

De esta forma tan sencilla obtenemos un portlet Welcome con contenido. Ahora solo nos quedan dos pequeños detalles: modificar el tamaño de la fuente y poner el título del portlet (figura 13.22).

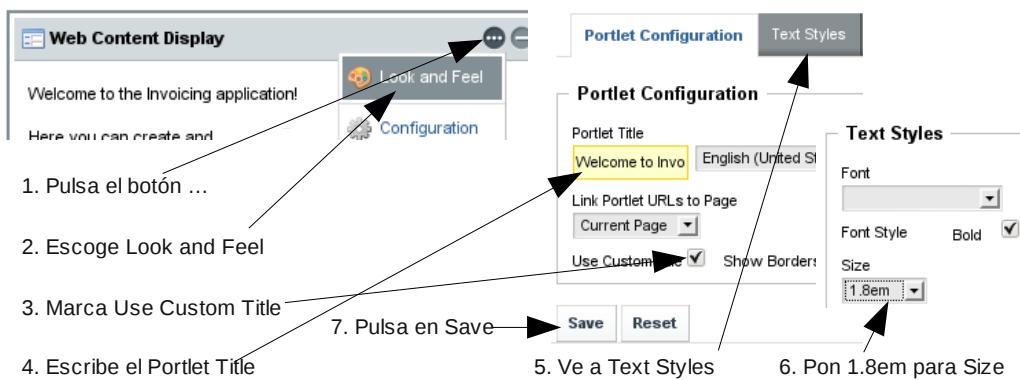


Figura 13.22 Cambiar el título del portlet y agrandar el tamaño de la fuente

¡Enhorabuena! Tu aplicación está desplegada en Liferay con una estructura de navegación e incluso con una página de bienvenida. Ha sido fácil gracias a las herramientas y portlets incluidos en Liferay. Sin embargo, queda todavía una pequeña deficiencia en nuestra aplicación: todo el mundo puede acceder a todos los módulos de Invoicing. Vamos a arreglar esto en la próxima sección.

13.5 Gestión de usuarios

Una cosa buena de trabajar con un portal es que de partida dispones de una gestión de usuarios completa y lista para usar. Por lo tanto, podemos empezar ya a añadir los roles y usuarios necesarios para configurar la seguridad de la aplicación.

13.5.1 Roles

Si bien Liferay permite asignar permisos directamente sobre usuarios individuales, suele ser más práctico configurar la seguridad usando los roles, y asociar estos roles a usuarios individuales. La tabla 13.1 muestra los roles que vamos a usar en la aplicación Invoicing.

Rol	Permisos
Invoicing admin	Acceso total a todos los datos y funcionalidad de la aplicación
Orderer	Acceso total a pedidos, pero no al resto de la aplicación
Seller	Acceso a facturas sólo para leer
Customer	Acceso sólo a sus propios pedidos

Tabla 13.1 Roles para la aplicación Invoicing

La figura 13.23 muestra los pasos necesarios para añadir un nuevo rol.

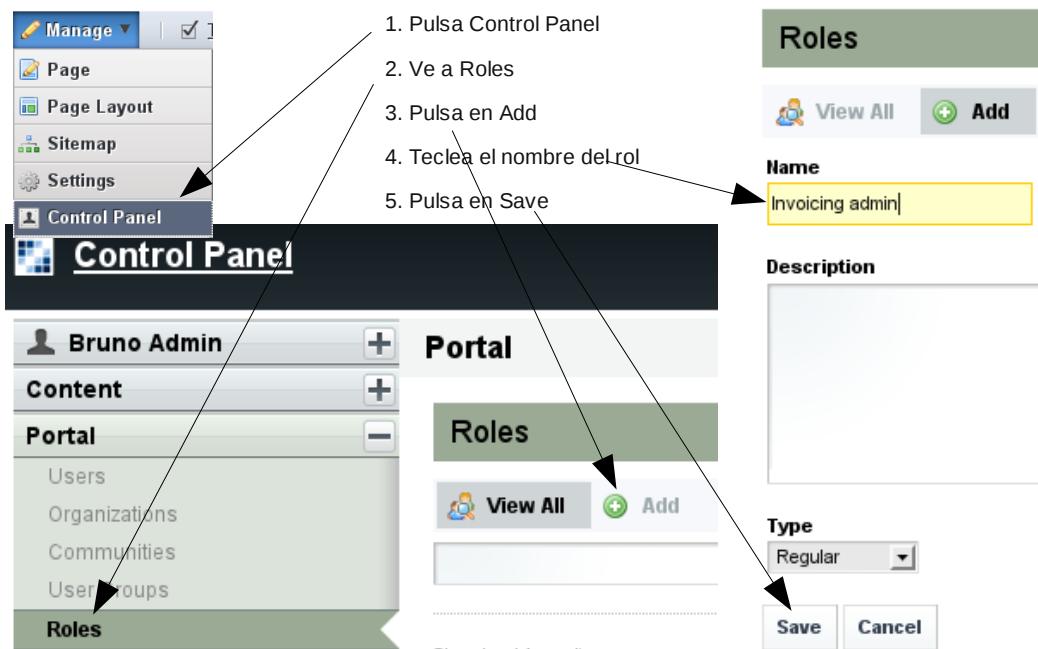


Figura 13.23 Añadir un nuevo rol a Liferay

Usa las instrucciones de la figura 13.23 para añadir los roles Invoicing admin, Orderer, Seller y Customer.

13.5.2 Usuarios

Hemos usado el panel de control para crear roles. Este panel de control aglutina todas las tareas administrativas de Liferay, así que podemos usarlo para crear usuarios también. La figura 13.24 muestra como crear un nuevo usuario desde el panel de control.

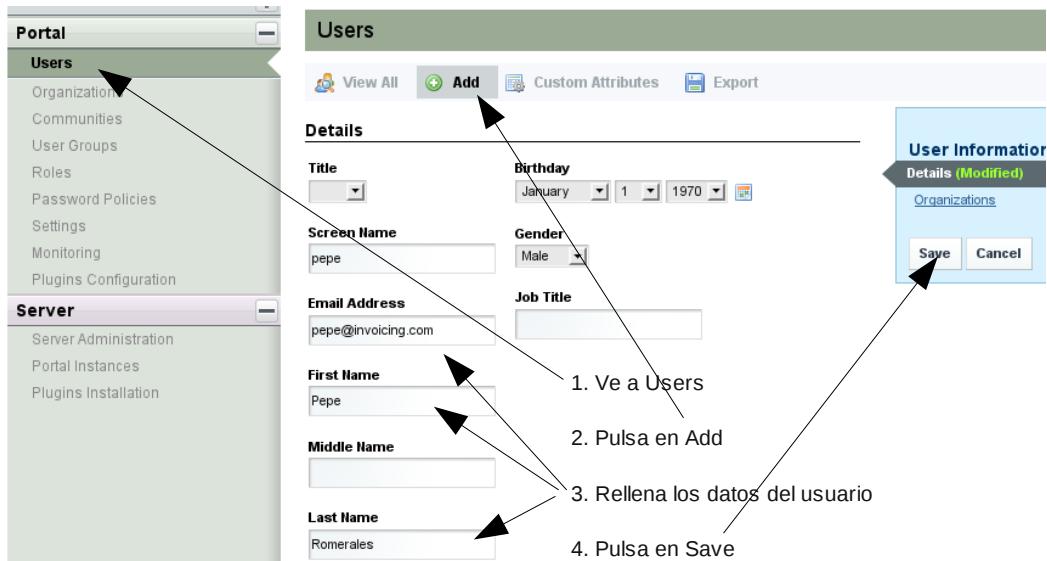


Figura 13.24 Añadir un nuevo usuario a Liferay

El siguiente paso es asignar una contraseña al usuario recién creado. Después de crear el usuario el menú de la derecha muestra más opciones disponibles para los usuarios, una de estas opciones es Password, úsala para asignar una contraseña al usuario, tal como muestra la figura 13.25.

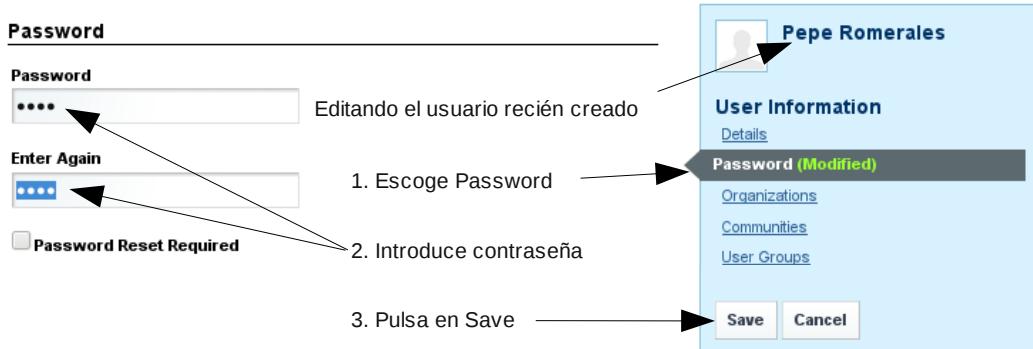


Figura 13.25 Asignar una contraseña al usuario

Lo último y más importante es asignar el rol correspondiente al usuario. La figura 13.26 muestra como hacerlo.

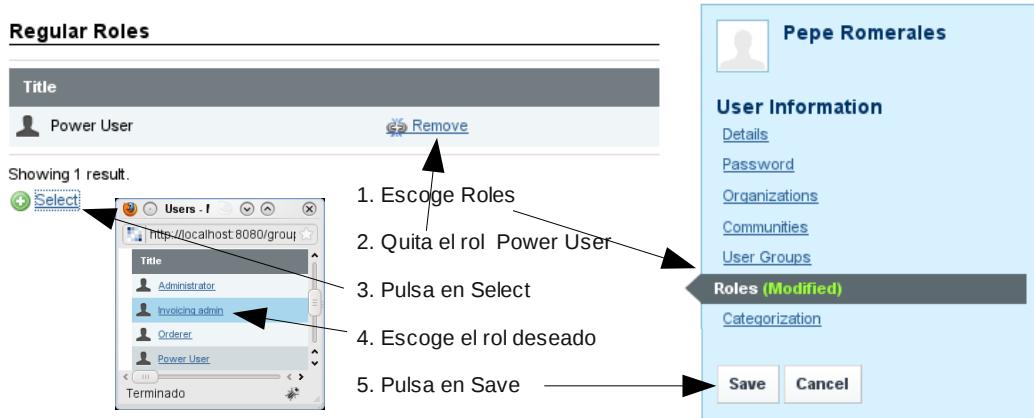


Figura 13.26 Configurar los roles para el usuario

Fíjate como el usuario ya tiene un rol asignado, Power User. Lo quitamos, porque el rol Power User permite al usuario configurar sus páginas añadiendo portlets, y queremos un entorno más restringido para los usuarios de Invoicing.

Usa las instrucciones de esta sección para crear cuatro usuarios, uno por cada rol que hemos creado en la sección anterior.

13.6 Niveles de acceso

De momento tenemos una aplicación disponible de forma pública y un puñado de usuarios y roles. Es el momento de juntar ambas cosas para restringir el acceso a nuestra aplicación.

13.6.1 Limitar el acceso a toda la aplicación

Nuestro primer objetivo es que nadie pueda acceder a Invoicing, excepto los usuarios del rol Invoicing admin. Hemos de cambiar los permisos de la página Invoicing en Liferay. La figura 13.27 muestra como acceder a estos permisos.



Figura 13.27 Ir a los permisos de la página Invoicing

Una vez en la página de permisos, quita el rol Guest y añade el rol Invoicing admin, tal como muestra la figura 13.28.

Role	Add Discussion	Update	View
Customer	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Guest	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Invoicing admin	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Orderer	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Submit

Figura 13.28 Configurar los roles autorizados a la página Invoicing

De esta manera, Invoicing ya no es accesible públicamente, a partir de ahora sólo los usuarios del rol Invoicing admin pueden acceder a él. Para comprobarlo, termina la sesión actual del usuario (sign off), y verás como el menú Invoicing no está presente, entonces vuelve a entrar usando un usuario con el rol Invoicing admin y verás el menú de Invoicing en su sitio.

13.6.2 Limitar el acceso a módulos individuales

El siguiente paso es que los usuarios del rol Orderer puedan acceder a la aplicación Invoicing, pero solo a la página Orders, como muestra la figura 13.29.

Guest no tiene menú Invoicing

Los usuarios Invoicing admin pueden acceder a todas las opciones de Invoicing

Los usuarios Orderer pueden acceder solo al módulo Order

Figura 13.29 Opciones disponibles para Guest, Invoicing admin y Orderer

Para conseguir este efecto tenemos que asignar permisos a cada página

individual (Orders, Invoices, Customer, Products, Categories, Authors y Trash), además de dar permisos a la página raíz (Invoicing). Esto es un poco más laborioso, pero nos permite tener diferentes niveles de acceso en la misma aplicación. Ya sabes como asignar roles a una página (lo aprendiste en las figuras 13.27 y 13.28 de la sección anterior), simplemente repite la tarea para cada página, como muestra la figura 13.30.

Role	Add Discussion	Update	View
<u>Customer</u>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<u>Guest</u>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<u>Invoicing admin</u>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<u>Orderer</u>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<u>Owner</u>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Role	Add Discussion	Update	View
<u>Customer</u>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<u>Guest</u>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<u>Invoicing admin</u>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<u>Orderer</u>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<u>Owner</u>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Figura 13.30 Asignar roles individualmente por cada página de módulo

Después de asignar los roles para cada página, asigna el rol Orderer a la página raíz Invoicing y tu aplicación estará lista para funcionar con estos dos roles. Ahora vamos a poner en juego un tercer rol, Seller.

13.6.3 Limitar funcionalidad

Los usuarios del rol Seller pueden consultar facturas, pero no pueden modificarlas o añadir nuevas. Por lo tanto, asignar la página Invoices a este rol no es una solución válida, porque el portlet asignado permite al usuario modificar y añadir facturas. Tenemos que añadir un nuevo módulo para consultar facturas. Edita el archivo *application.xml* de tu proyecto añadiendo el módulo del listado 13.2.

Listado 13.2 Módulo de solo lectura para Invoice definido en application.xml

```
<module name="ConsultInvoice">
    <env-var name="XAVA_SEARCH_ACTION"
        value="CRUD.searchReadonly"/> <!-- Usando esta acción de búsqueda
                                         los datos no son editables en el detalle -->
    <model name="Invoice"/>
    <controller name="Print"/> <!-- Sólo están disponibles las acciones de imprimir,
                                    CRUD no se incluye -->
</module>
```

De esta manera tan simple definimos un módulo de solo lectura para Invoice. Ahora, redespliega tu aplicación y configura la seguridad para este nuevo módulo. Durante el transcurso de este capítulo has aprendido como hacerlo. Tienes que:

- Generar portlets y redesplegar la aplicación en Liferay (sección 13.3).
- Añadir una nueva página llamada “Consult invoices” copiándola de la página “Invoice” y poniendo ahí el portlet ConsultInvoice (sección 13.4.4).
- Configurar los permisos de la página para ser accesible solo para los usuarios del rol Seller (13.6.2).

Aquí has visto como dar a cada tipo de usuario (rol) diferentes posibilidades. Simplemente crea módulos con diferentes capacidades (con distintos controladores) y asignalos a los roles deseados dentro de Liferay.

13.6.4 Limitar la visibilidad de los datos en modo lista

A veces queremos limitar no solo la funcionalidad disponible sino también los datos que el usuario puede ver. Por ejemplo, podemos permitir que el cliente se identifique en el portal, y vea la lista de sus propios pedidos, pero obviamente no los pedidos de otros clientes.

Podemos conseguir esto usando una condición en la lista con el id como argumento. Añade otra anotación @Tab a la entidad Order, como se muestra en el listado 13.3.

Listado 13.3 Nuevo @Tab en Order para filtrar por cliente

```
import org.openxava.filters.*; // Para usar UserFilter

...
@Tabs({
    ...
    @Tab(name="CurrentCustomer", // Este nombre se usa al definir el módulo
        baseCondition =
            "cast (${customer.number} as varchar) = ?", // Filtra por cliente
        filter=UserFilter.class, // Pone el usuario actual como valor para el parámetro(el ?)
        properties="year, number, date, customer.number, customer.name, "
            + "delivered, baseAmount, vat, totalAmount"
    ),
})
public class Order extends CommercialDocument {
```

Como ya sabes baseCondition establece una condición a aplicar a los datos de la lista. Aquí usamos un parámetro (el interrogante, ?), el valor del parámetro es rellenado por UserFilter con el usuario actualmente identificado. UserFilter se incluye en OpenXava. Fíjate que usamos un molde (cast (as varchar)) en la condición, esto es porque customer.number es numérico y UserFilter devuelve una cadena. Usar un molde sobre el campo de una

condición ofrece un rendimiento pobre en algunas bases de dato (ya que impide el uso de los índices). Si te encuentras con este problema de rendimiento, quita el molde de la condición y crea tu propio filtro implementando la interfaz `org.openxava.filters.IFilter` que devuelva el id de usuario como un valor numérico.

Lo siguiente es definir un módulo que use este @Tab. El listado 13.4 muestra la definición de módulo que has de añadir a `application.xml`.

Listado 13.4 Módulo para mostrar los pedidos del cliente actual

```
<module name="CustomerOrders">
<env-var name="XAVA_SEARCH_ACTION" value="CRUD.searchReadOnly"/>
<model name="Order"/>
<tab name="CurrentCustomer"/> <!-- Usa el tab CurrentCustomer para la lista -->
<controller name="Print"/>
</module>
```

Como puedes ver este módulo es un simple módulo de solo lectura que usa el tab `CurrentCustomer` para los datos de la lista. Ahora has de redesplegar la aplicación, añadir la entrada de menú para este nuevo módulo y configurar su seguridad:

- Genera los portlets y redesplica la aplicación en Liferay (sección 13.3).
- Añade una nueva página llamada “My orders” copiándola de la página “Orders” y pon el portlet `CustomerOrders` ahí (sección 13.4.4).
- Configura los permisos de la página para ser accesible sólo por usuarios con el rol `Customer` (13.6.2).

Nos apoyamos en el hecho de que el número de cliente coincide con el id de usuario; por tanto primero has de crear los usuarios en el portal, y después los clientes usando el id de usuario como número de cliente, como en la figura 13.31.

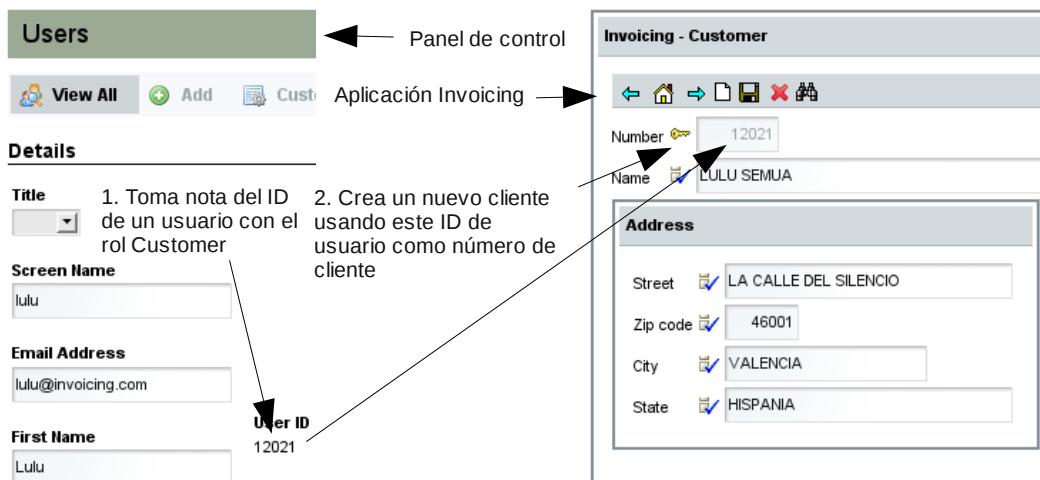


Figura 13.31 El número de cliente ha de coincidir con el id de usuario

Aquí se ve como tenemos un usuario llamado “lulu”, cuyo id de usuario es 12021, entonces creamos un cliente para “lulu” usando 12021 como número. Para comprobar que todo funciona correctamente puedes crear algunos pedidos para “lulu” usando el portlet Order. Después, identifícate en Liferay como “lulu”, verás solo la opción My orders del menú Invoicing, y dentro de él una lista de los pedidos de “lulu”.

Obviamente esta técnica no funciona si ya tienes los clientes creados. En este caso, añade una nueva propiedad a Customer, portalUser, y adapta la condición base del @Tab para filtrar por esta propiedad.

13.6.5 Limitar la visibilidad de los datos en modo detalle

Una manera interesante de mejorar la página My orders para los clientes es permitiendo crear nuevos pedidos desde ahí. En este caso, solo queremos añadir la posibilidad de crear nuevos pedidos, pero no editar o borrar los ya existentes. Además, los datos del cliente tienen que estar rellenados con los datos del usuario actual y no se pueden cambiar. La figura 13.32 muestra el comportamiento deseado.



Figura 13.32 Crear un nuevo pedido por un usuario con el rol Customer

Para añadir este comportamiento al módulo CustomerOrders has de definir un nuevo controlador para él. Modifica su definición de módulo en *application.xml*, tal como muestra el listado 13.5.

Listado 13.5 Módulo CustomerOrders con el controlador CustomerOrder

```
<module name="CustomerOrders">
<env-var name="XAVA_SEARCH_ACTION" value="CRUD.searchReadOnly"/>
<model name="Order"/>
<tab name="CurrentCustomer"/>
<controller name="CustomerOrders"/> <!-- Añade este nuevo controlador -->
<controller name="Print"/>
</controller>
```

Este nuevo controlador, CustomerOrders, define las acciones “new” y “save”. El listado 13.6 muestra su definición en *controllers.xml*.

Listado 13.6 Controlador CustomerOrders en controllers.xml

```
<controller name="CustomerOrders">

    <!-- Definida como CRUD.new pero usando nuestra acción -->
    <action name="new"
        class="org.openxava.invoicing.actions.NewOrderForCurrentUserAction"
        image="images/new.gif"
        keystroke="Control N"/>

    <!-- Definida como la CRUD.save estándar -->
    <action name="save" mode="detail"
        class="org.openxava.actions.SaveAction"
        image="images/save.gif"
        keystroke="Control S"/>

</controller>
```

Este controlador es una versión refinada del controlador estándar CRUD de OpenXava. Solo incluimos dos acciones, “new” y “save”. La acción save es la estándar de OpenXava, pero para la acción new usamos nuestra propia clase. El listado 13.7 muestra el código para esta clase.

Listado 13.7 Acción para crear un nuevo pedido a partir del usuario actual

```
package org.openxava.invoicing.actions; // En el paquete 'actions'

import org.openxava.actions.*;
import org.openxava.util.*; // Este paquete contiene la clase Users
import org.openxava.view.*;

public class NewOrderForCurrentUserAction extends NewAction {

    public void execute() throws Exception {
        super.execute(); // Ejecuta la lógica estándar para una acción new
        String user = Users.getCurrent(); // El id del usuario actualmente identificado (1)
        if (user == null) { // No hay usuario identificado
            getView().setEditable(false);
            addError("no_user_logged");
            return;
        }
        int customerNumber = Integer.parseInt(user); // Porque en Liferay el id de
        // usuario es un número, que hacemos coincidir con el número de cliente
        View customerView = getView().getSubview("customer");
        customerView.setValue("number", customerNumber); // Rellena el número de
```

```

    customerView.findObject(); // Carga los datos del cliente
    // usando el número como clave
    customerView.setKeyEditable(false); // El usuario no puede cambiar el cliente (2)
}

}

```

Como puedes ver esta acción rellena los datos del cliente usando el id de usuario como clave. Así, cuando el usuario pulsa en el botón para new, los datos del cliente se llenarán automáticamente. Un detalle importante es que la vista del cliente no es editable (2), de esta forma la lista para buscar clientes no está disponible, por lo tanto el usuario no puede acceder a los datos de otros clientes.

Es de notar como puedes usar `Users.getCurrent()` (una utilidad del paquete `org.openxava.util`) para obtener el usuario actualmente identificado. Ésta es una herramienta muy útil para limitar la visibilidad de los datos programáticamente.

13.7 Pruebas JUnit

La mayor parte del trabajo que hemos hecho en este capítulo ha sido sobre la configuración de nuestra aplicación dentro de Liferay. Aunque, también hemos añadido algunos módulos a la aplicación: `ConsultInvoice` y `CustomerOrders`. El primero es un mero módulo de solo lectura normal y corriente, sin embargo el último añade alguna funcionalidad interesante que vamos a probar en esta sección.

Hemos de probar el módulo `CustomerOrders` dentro de Liferay, porque este módulo no tiene sentido si no hay un usuario autenticado. Para probar un módulo OpenXava dentro de Liferay has de cumplir estas dos reglas: el módulo ha de ser públicamente accesible y ha de tener una URL amigable con la forma *aplicación/módulo*. Para hacer esto, ve a Manage Page dentro de Liferay y sigue las instrucciones en la figura 13.33.

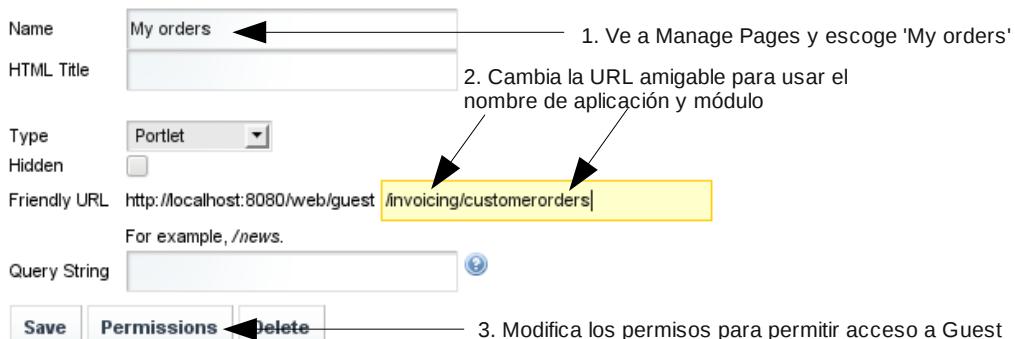


Figura 13.33 Estableciendo URL amigable y permisos para la página My orders

Aunque estés dando acceso público a esta página, no es un problema ya que los datos se filtran por usuario, y no hay facturas asociadas al usuario Guest. Además, estamos haciendo la prueba contra un Liferay de desarrollo, y, como desarrollador, nunca tocarás los permisos de un Liferay de producción.

Para instruir a tus pruebas de forma que vayan contra Liferay en vez de contra un Tomcat plano, edita el archivo *xava-junit.properties* en la carpeta *properties* del proyecto Invoicing, y añade la entrada del listado 13.8.

Listado 13.8 Entrada en *xava-junit.properties* para probar contra Liferay

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

Después, escribimos la prueba para el módulo *CustomerOrders*. Esta prueba se identifica en Liferay usando un usuario de prueba llamado 'lulu'. Este usuario tiene pedidos. El listado 13.9 muestra el código de la prueba.

Listado 13.9 Prueba para *CustomerOrders* que corre dentro de Liferay

```
package org.openxava.invoicing.tests;

import org.openxava.tests.*;

public class CustomerOrdersTest extends ModuleTestCase {

    public CustomerOrdersTest(String testName) {
        super(testName, "Invoicing", "CustomerOrders");
    }

    public void testLimitingDataVisibility() throws Exception {
        login("lulu@invoicing.com", "lulu"); // Para identificarse en Liferay

        assertListNotEmpty(); // El usuario tiene pedidos
        int rowCount = getListRowCount();
        for (int row=0; row<rowCount; row++) { // Todos los pedidos en la lista han de ser
            assertEquals(row, "customer.name", "LULU SEMUA"); // del usuario (1)
        }

        execute("CustomerOrders.new"); // Para crear un nuevo pedido
        assertEquals("customer.number"); // Los datos del cliente no se pueden cambiar
        assertEquals("customer.name", "LULU SEMUA"); // Los datos del cliente en el
        // pedido se rellenan dependiendo del usuario identificado (2)
        logout();
    }
}
```

Aquí verificamos primero que la lista muestra solo las facturas del usuario autenticado (1), así comprobamos que el @Tab que creamos en la sección 13.6.4 para limitar la visibilidad de los datos funciona. Entonces confirmamos que la acción 'new' que escribimos en la sección 13.6.5 rellena correctamente los datos del cliente (2).

13.8 Resumen

En este capítulo has visto como integrar tu aplicación con Liferay. Esto añade navegación y seguridad a tu aplicación sin ningún coste de programación. Liferay es una herramienta muy potente y aquí nos hemos limitado a rascar su superficie. Te invito a que aprendas más en la sección de documentación del sitio de Liferay²³.

La combinación entre OpenXava y Liferay te ofrece una herramienta de nivel empresarial para desarrollar y desplegar tus aplicaciones.

23 <http://www.liferay.com/documentation>

Código fuente

apéndice A

Aquí tienes el código fuente completo para la aplicación desarrollada en este libro. De hecho, si creas un proyecto OpenXava desde cero²⁴ y copias el código fuente de este apéndice en él, obtendrás la aplicación Invoicing.

Este código puede ser una referencia útil, especialmente si estás leyendo la versión impresa del libro sin un ordenador a mano. Además, es una evidencia palpable del poco código que se requiere al desarrollar con OpenXava. Tanto es así, que tienes una aplicación de facturación en menos de 2400 líneas de código.

A.1 Modelo

Las clases del modelo representan los conceptos de negocio. Contienen la estructura de los datos, la lógica de negocio y metadatos que permiten a OpenXava dibujar la interfaz de usuario, hacer las validaciones y grabar y cargar objetos desde la base de datos.

Estas clases se encuentran en *Invoicing/src/org.openxava.invoicing.model*.

Listado A.1 Identifiable.java

```
package org.openxava.invoicing.model;

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

@MappedSuperclass
public class Identifiable {

    @Id @GeneratedValue(generator="system-uuid") @Hidden
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid;

    public String getOid() {
        return oid;
    }

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

}
```

Listado A.2 Category.java

```
package org.openxava.invoicing.model;

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

@Entity
@NamedQueries({
```

²⁴ Usando OpenXavaTemplate como se explica en el capítulo 4

279 Apéndice A: Código fuente

```
public class Category extends Identifiable {

    @Column(length=50)
    private String description;

    public String getDescription() {
        return description;
    }

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

}
```

Listado A.3 Product.java

```
package org.openxava.invoicing.model;

import java.math.*;
import javax.persistence.*;
import org.openxava.annotations.*;
import org.openxava.invoicing.annotations.*;

@Entity
@View(name="Simple", members="number, description")
public class Product {

    @Id @Column(length=9)
    private int number;

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

    @ManyToOne(fetch=FetchType.LAZY)
    @DescriptionsList
    private Author author;

    @ManyToOne(
        fetch=FetchType.LAZY,
        optional=true)
    @DescriptionsList
    private Category category;

    @Column(length=10) @ISBN
    private String isbn;

    @Stereotype("MONEY")
    private BigDecimal price;

    @Stereotype("PHOTO")
    private byte [] photo;

    @Stereotype("IMAGES_GALLERY")
    @Column(length=32)
    private String morePhotos;
```

```
@Stereotype("MEMO")
private String remarks;

public Category getCategory() {
    return category;
}

public void setCategory(Category category) {
    this.category = category;
}

public int getNumber() {
    return number;
}

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

public String getDescription() {
    return description;
}

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

public BigDecimal getPrice() {
    return price;
}

public void setPrice(BigDecimal price) {
    this.price = price;
}

public byte[] getPhoto() {
    return photo;
}

public void setPhoto(byte[] photo) {
    this.photo = photo;
}

public String getMorePhotos() {
    return morePhotos;
}

public void setMorePhotos(String morePhotos) {
    this.morePhotos = morePhotos;
}

public String getRemarks() {
    return remarks;
}

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

public void setAuthor(Author author) {
```

281 Apéndice A: Código fuente

```
        this.author = author;
    }

    public Author getAuthor() {
        return author;
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

}
```

Listado A.4 Author.java

```
package org.openxava.invoicing.model;

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

@Entity
public class Author extends Identifiable {

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

    @OneToMany(mappedBy="author")
    @ListProperties("number, description, price")
    private Collection<Product> products;

    // Getters and setters
    public String getName() {
        return name;
    }

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

    public Collection<Product> getProducts() {
        return products;
    }

    public void setProducts(Collection<Product> products) {
        this.products = products;
    }
}
```

Listado A.5 Address.java

```
package org.openxava.invoicing.model;
```

```

import javax.persistence.*;

@Embeddable
public class Address {

    @Column(length=30)
    private String street;

    @Column(length=5)
    private int zipCode;

    @Column(length=20)
    private String city;

    @Column(length=30)
    private String state;

    public String getStreet() {
        return street;
    }

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

    public int getZipCode() {
        return zipCode;
    }

    public void setZipCode(int zipCode) {
        this.zipCode = zipCode;
    }

    public String getCity() {
        return city;
    }

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

    public String getState() {
        return state;
    }

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

```

Listado A.6 Customer.java

```

package org.openxava.invoicing.model;

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

```

283 Apéndice A: Código fuente

```
@Entity
@View(name="Simple",
      members="number, name"
)
public class Customer {

    @Id
    @Column(length=6)
    private int number;

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

    @Embedded
    private Address address;

    public Address getAddress() {
        if (address == null) address = new Address();
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    public int getNumber() {
        return number;
    }

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

    public String getName() {
        return name;
    }

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

Listado A.7 Deletable.java

```
package org.openxava.invoicing.model;

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

@MappedSuperclass
public class Deletable extends Identifiable {

    @Hidden
    private boolean deleted;

    public boolean isDeleted() {
```

```

        return deleted;
    }

    public void setDeleted(boolean deleted) {
        this.deleted = deleted;
    }

}

```

Listado A.8 CommercialDocument.java

```

package org.openxava.invoicing.model;

import java.math.*;
import java.util.*;
import javax.persistence.*;

import org.hibernate.validator.*;
import org.openxava.annotations.*;
import org.openxava.calculators.*;
import org.openxava.invoicing.calculators.*;
import org.openxava.jpa.*;

@Entity
@View(members=
    "year, number, date;" +
    "data {" +
        "customer;" +
        "details;" +
        "amounts [ " +
            "vatPercentage, baseAmount, vat, totalAmount" +
        "];" +
        "remarks" +
    "}" +
)
abstract public class CommercialDocument extends Deletable {

    @Column(length=4)
    @DefaultValueCalculator(CurrentYearCalculator.class)
    @SearchKey
    private int year;

    @Column(length=6)
    @ReadOnly(forViews="DEFAULT")
    @SearchKey
    private int number;

    @Required
    @DefaultValueCalculator(FromDateCalculator.class)
    private Date date;

    @ManyToOne(fetch=FetchType.LAZY, optional=false)
    @ReferenceView("Simple")
    private Customer customer;

    @OneToMany(
        mappedBy="parent",

```

285 Apéndice A: Código fuente

```
    cascade=CascadeType.ALL)
@ListProperties(
    "product.number, product.description, " +
    "quantity, pricePerUnit, amount")
private Collection<Detail> details = new ArrayList<Detail>();

@Digits(integerDigits=2, fractionalDigits=0)
@Required
@DefaultValueCalculator(VatPercentageCalculator.class)
private BigDecimal vatPercentage;

@stereotype("MONEY")
private BigDecimal amount;

@stereotype("MONEY")
public BigDecimal getBaseAmount() {
    BigDecimal result = new BigDecimal("0.00");
    for (Detail detail: getDetails()) {
        result = result.add(detail.getAmount());
    }
    return result;
}

@stereotype("MONEY")
@Depends("vatPercentage")
public BigDecimal getVat() {
    return getBaseAmount()
        .multiply(getVatPercentage())
        .divide(new BigDecimal("100"));
}

@stereotype("MONEY")
@Depends("baseAmount, vat")
public BigDecimal getTotalAmount() {
    return getBaseAmount().add(getVat());
}

@org.hibernate.annotations.Formula("AMOUNT * 0.10")
@stereotype("MONEY")
private BigDecimal estimatedProfit;

public BigDecimal getEstimatedProfit() {
    return estimatedProfit;
}

@stereotype("MEMO")
private String remarks;

@Transient
private boolean removing = false;

boolean isRemoving() {
    return removing;
}

@PreRemove
private void markRemoving() {
    this.removing = true;
}
```

```
@PostRemove
private void unmarkRemoving() {
    this.removing = false;
}

@PrePersist
public void calculateNumber() throws Exception {
    Query query = XPersistence.getManager()
        .createQuery("select max(i.number) from " +
            getClass().getSimpleName() + " i where i.year = :year");
    query.setParameter("year", year);
    Integer lastNumber = (Integer) query.getSingleResult();
    this.number = lastNumber == null?1:lastNumber + 1;
}

public String toString() {
    return year + "/" + number;
}

public void recalculateAmount() {
    setAmount(getTotalAmount());
}

// Getters and setters
public int getYear() {
    return year;
}

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

public int getNumber() {
    return number;
}

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

public Date getDate() {
    return date;
}

public void setDate(Date date) {
    this.date = date;
}

public String getRemarks() {
    return remarks;
}

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

public Customer getCustomer() {
    return customer;
}
```

287 Apéndice A: Código fuente

```
public void setCustomer(Customer customer) {
    this.customer = customer;
}

public Collection<Detail> getDetails() {
    return details;
}

public void setDetails(Collection<Detail> details) {
    this.details = details;
}

public BigDecimal getVatPercentage() {
    return vatPercentage==null?
        BigDecimal.ZERO:vatPercentage;
}

public void setVatPercentage(BigDecimal vatPercentage) {
    this.vatPercentage = vatPercentage;
}

public BigDecimal getAmount() {
    return amount;
}

public void setAmount(BigDecimal amount) {
    this.amount = amount;
}

}
```

Listado A.9 Detail.java

```
package org.openxava.invoicing.model;

import java.math.*;
import javax.persistence.*;
import org.openxava.annotations.*;
import org.openxava.invoicing.calculators.*;

@Entity
@View(members="product; quantity, pricePerUnit, amount")
public class Detail extends Identifiable {

    @ManyToOne
    private CommercialDocument parent;

    private int quantity;

    @ManyToOne(fetch=FetchType.LAZY, optional=true)
    @ReferenceView("Simple")
    @NoFrame
    private Product product;

    @DefaultValueCalculator(
        value=PricePerUnitCalculator.class,
        properties=@PropertyValue(
```

```
        name="productNumber",
        from="product.number")
)
@stereotype("MONEY")
private BigDecimal pricePerUnit;

@PrePersist
private void onPersist() {
    getParent().getDetails().add(this);
    getParent().recalculateAmount();
}

@PreUpdate
private void onUpdate() {
    getParent().recalculateAmount();
}

@PreRemove
private void onRemove() {
    if (getParent().isRemoving()) return;
    getParent().getDetails().remove(this);
    getParent().recalculateAmount();
}

@stereotype("MONEY")
@Depends("pricePerUnit, quantity")
public BigDecimal getAmount() {
    return new BigDecimal(quantity)
        .multiply(getPricePerUnit());
}

// Getters and setters
public CommercialDocument getParent() {
    return parent;
}

public void setParent(CommercialDocument parent) {
    this.parent = parent;
}

public int getQuantity() {
    return quantity;
}

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

public Product getProduct() {
    return product;
}

public void setProduct(Product product) {
    this.product = product;
}

public BigDecimal getPricePerUnit() {
    return pricePerUnit==null?
        BigDecimal.ZERO:pricePerUnit;
}
```

289 Apéndice A: Código fuente

```
public void setPricePerUnit(BigDecimal pricePerUnit) {
    this.pricePerUnit = pricePerUnit;
}

}
```

Listado A.10 Order.java

```
package org.openxava.invoicing.model;

import org.openxava.filters.*;
import java.util.*;

import javax.persistence.*;

import org.apache.commons.beanutils.*;
import org.hibernate.validator.*;
import org.openxava.annotations.*;
import org.openxava.invoicing.actions.*;
import org.openxava.jpa.*;
import org.openxava.util.*;
import org.openxava.validators.*;

@Entity
@Views({
    @View( extendsView="super.DEFAULT",
        members="delivered; invoice { invoice } "
    ),
    @View( name="NoCustomerNoInvoice",
        members=
            "year, number, date;" +
            "details;" +
            "remarks"
    )
})
@Tabs({
    @Tab(baseCondition = "deleted = false",
        properties="year, number, date, customer.number, customer.name," +
        "delivered, vatPercentage, estimatedProfit, baseAmount, " +
        "vat, totalAmount, amount, remarks"),
    @Tab(name="Deleted", baseCondition = "deleted = true"),
    @Tab(name="CurrentCustomer",
        baseCondition =
            "cast (${customer.number} as varchar) = ?",
        filter=UserFilter.class,
        properties="year, number, date, customer.number, customer.name," +
            "delivered, baseAmount, vat, totalAmount"
    )
})
public class Order extends CommercialDocument {

    @ManyToOne
    @ReferenceView("NoCustomerNoOrders")
    @OnChange(ShowHideCreateInvoiceAction.class)
    @OnChangeSearch(OnChangeSearchInvoiceAction.class)
    @SearchAction("Order.searchInvoice")
    private Invoice invoice;
```

```
@OnChange(ShowHideCreateInvoiceAction.class)
private boolean delivered;

public boolean isDelivered() {
    return delivered;
}

public void setDelivered(boolean delivered) {
    this.delivered = delivered;
}

public Invoice getInvoice() {
    return invoice;
}

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

public void setDeleted(boolean deleted) {
    if (deleted) validateOnRemove();
    super.setDeleted(deleted);
}

@AssertTrue
private boolean isDeliveredToBeInInvoice() {
    return invoice == null || isDelivered();
}

@PreRemove
private void validateOnRemove() {
    if (invoice != null) {
        throw new IllegalStateException(
            XavaResources.getString(
                "cannot_delete_order_with_invoice"));
    }
}

public void createInvoice()
    throws ValidationException
{
    if (this.invoice != null) {
        throw new ValidationException(
            "impossible_create_invoice_order_already_has_one");
    }
    if (!isDelivered()) {
        throw new ValidationException(
            "impossible_create_invoice_order_is_not_delivered");
    }
    try {
        Invoice invoice = new Invoice();
        BeanUtils.copyProperties(invoice, this);
        invoice.setId(null);
        invoice.setDate(new Date());
        invoice.setDetails(new ArrayList());
        XPersistence.getManager().persist(invoice);
        copyDetailsToInvoice(invoice);
        this.invoice = invoice;
    }
}
```

291 Apéndice A: Código fuente

```
        }
    } catch (Exception ex) {
        throw new SystemException(
            "impossible_create_invoice", ex);
    }
}

public void copyDetailsToInvoice(Invoice invoice) {
    try {
        for (Detail orderDetail: getDetails()) {
            Detail invoiceDetail = (Detail)
                BeanUtils.cloneBean(orderDetail);

            invoiceDetail.setOid(null);
            invoiceDetail.setParent(invoice);
            XPersistence.getManager().persist(invoiceDetail);
        }
    } catch (Exception ex) {
        throw new SystemException(
            "impossible_copy_details_to_invoice", ex);
    }
}

public void copyDetailsToInvoice() {
    copyDetailsToInvoice(getInvoice());
}

@AssertTrue
private boolean isCustomerOfInvoiceMustBeTheSame() {
    return invoice == null ||
        invoice.getCustomer().getNumber()==getCustomer().getNumber();
}

}
```

Listado A.11 Invoice.java

```
package org.openxava.invoicing.model;

import java.util.*;
import javax.persistence.*;
import org.openxava.annotations.*;
import org.openxava.validators.*;

@Entity
@Views({
    @View(  extendsView="super.DEFAULT",
        members="orders { orders } "
    ),
    @View(  name="NoCustomerNoOrders",
        members=
            "year, number, date;" +
            "details;" +
            "remarks"
    )
})
@Tabs({
```

```

@Tab(
    baseCondition = "deleted = false",
    properties="year, number, date, customer.number, customer.name," +
        "vatPercentage, estimatedProfit, baseAmount, " +
        "vat, totalAmount, amount, remarks"),
    @Tab(name="Deleted", baseCondition = "deleted = true")
})
public class Invoice extends CommercialDocument {

    @OneToMany(mappedBy="invoice")
    @NewAction("Invoice.addOrders")
    @CollectionView("NoCustomerNoInvoice")
    private Collection<Order> orders;

    public static Invoice createFromOrders(Collection<Order> orders)
        throws ValidationException
    {
        Invoice invoice = null;
        for (Order order: orders) {
            if (invoice == null) {
                order.createInvoice();

                invoice = order.getInvoice();
            }
            else {
                order.setInvoice(invoice);
                order.copyDetailsToInvoice(invoice);
            }
        }
        if (invoice == null) {
            throw new ValidationException(
                "impossible_create_invoice_orders_not_specified");
        }
        return invoice;
    }

    public Collection<Order> getOrders() {
        return orders;
    }

    public void setOrders(Collection<Order> orders) {
        this.orders = orders;
    }
}

```

A.2 Acciones

Las acciones reaccionan a las pulsaciones de vínculos y botones por parte del usuario. Son las responsables del comportamiento de la aplicación, la así llamada capa del controlador. Las acciones conectan la interfaz de usuario con los objetos del modelo. Usualmente no tienen lógica de negocio ellas mismas, sino que delegan en los objetos del modelo para ejecutar la lógica de negocio.

Estas clases se encuentran en *Invoicing/src/org.openxava.invoicing.actions*.

293 Apéndice A: Código fuente

Listado A.12 AddOrdersToInvoiceAction.java

```
package org.openxava.invoicing.actions;

import java.rmi.*;
import java.util.*;
import javax.ejb.*;
import org.openxava.actions.*;
import org.openxava.invoicing.model.*;
import org.openxava.model.*;
import org.openxava.util.*;
import org.openxava.validators.*;

public class AddOrdersToInvoiceAction
    extends AddElementsToCollectionAction {

    public void execute() throws Exception {
        super.execute();
        getView().refresh();
    }

    protected void associateEntity(Map keyValues)
        throws ValidationException,
               XavaException, ObjectNotFoundException,
               FinderException, RemoteException
    {
        super.associateEntity(keyValues);
        Order order = (Order) MapFacade.findEntity("Order", keyValues);
        order.copyDetailsToInvoice();
    }

}
```

Listado A.13 CreateInvoiceFromOrderAction.java

```
package org.openxava.invoicing.actions;

import org.openxava.actions.*;
import org.openxava.invoicing.model.*;
import org.openxava.jpa.*;
import org.openxava.model.*;

public class CreateInvoiceFromOrderAction
    extends ViewBaseAction
    implements IHideActionAction
{
    private boolean hideAction = false;

    public void execute() throws Exception {
        Object oid = getView().getValue("oid");
        if (oid == null) {
            addError(
                "impossible_create_invoice_order_not_exist");
            return;
        }
        MapFacade.setValues("Order",
            getView().getKeyValues(), getView().getValues());
    }
}
```

```

        Order order = XPersistence.getManager().find(
            Order.class,
            getView().getValue("oid"));
        order.createInvoice();
        getView().refresh();
        addMessage("invoice_created_from_order",
            order.getInvoice());
        hideAction = true;
    }

    public String getActionToHide() {
        return hideAction?"Order.createInvoice":null;
    }
}

```

Listado A.14 CreateInvoiceFromSelectedOrdersAction.java

```

package org.openxava.invoicing.actions;

import java.util.*;
import javax.ejb.*;
import javax.inject.*;
import org.openxava.actions.*;
import org.openxava.invoicing.model.*;
import org.openxava.model.*;

public class CreateInvoiceFromSelectedOrdersAction
    extends TabBaseAction
    implements IChangeModuleAction
{
    @Inject
    private Map currentInvoiceKey;

    public void execute() throws Exception {
        Collection<Order> orders = getSelectedOrders();
        Invoice invoice = Invoice.createFromOrders(orders);
        addMessage(
            "invoice_created_from_orders", invoice, orders);
        currentInvoiceKey = toKey(invoice);
    }

    private Map toKey(Invoice invoice) {
        Map key = new HashMap();
        key.put("oid", invoice.getId());
        return key;
    }

    private Collection<Order> getSelectedOrders()
        throws FinderException
    {
        Collection<Order> result = new ArrayList<Order>();
        for (Map key: getTab().getSelectedKeys()) {
            Order order = (Order)
                MapFacade.findEntity("Order", key);
            result.add(order);
        }
    }
}

```

295 Apéndice A: Código fuente

```
        return result;
    }

    public String getNextModule() {
        return "CurrentInvoiceEdition";
    }

    public boolean hasReinitNextModule() {
        return true;
    }

}
```

Listado A.15 GoAddOrdersToInvoiceAction.java

```
package org.openxava.invoicing.actions;

import org.openxava.actions.*;

public class GoAddOrdersToInvoiceAction
    extends GoAddElementsToCollectionAction {

    public void execute() throws Exception {
        super.execute();
        int customerNumber =
            getPreviousView()
                .getValueInt("customer.number");

        getTab().setBaseCondition(
            "${customer.number} = " + customerNumber +
            " and ${delivered} = true and ${invoice.oid} is null"
        );
    }

    public String getNextController() {
        return "AddOrdersToInvoice";
    }

}
```

Listado A.16 InvoicingDeleteAction.java

```
package org.openxava.invoicing.actions;

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

public class InvoicingDeleteAction
    extends ViewBaseAction
    implements IChainAction {

    private String nextAction = null;

    public void execute() throws Exception {
        if (getView())
            .getKeyValuesWithValue()
```

```

        .isEmpty())
    {
        addError("no_delete_not_exists");
        return;
    }
    if (!getView().getMetaModel()
        .containsMetaProperty("deleted"))
    {
        nextAction = "CRUD.delete";
        return;
    }
    Map values = new HashMap();
    values.put("deleted", true);
    MapFacade.setValues(
        getModelName(),
        getView().getKeyValues(),
        values);
    resetDescriptionsCache();
    addMessage("object_deleted", getModelName());
    getView().clear();
    getView().setEditable(false);
}

public String getNextAction()
    throws Exception
{
    return nextAction;
}

}

```

Listado A.17 InvoicingDeleteSelectedAction.java

```

package org.openxava.invoicing.actions;

import java.util.*;
import org.openxava.actions.*;
import org.openxava.model.*;
import org.openxava.model.meta.*;
import org.openxava.validators.*;

public class InvoicingDeleteSelectedAction
    extends TabBaseAction
    implements IChainAction {

    private String nextAction = null;
    private boolean restore;

    public void execute() throws Exception {
        if (!getMetaModel().containsMetaProperty("deleted")) {
            nextAction="CRUD.deleteSelected";

            return;
        }
        markSelectedEntitiesAsDeleted();
    }

    private MetaModel getMetaModel() {

```

297 Apéndice A: Código fuente

```
        return MetaModel.get(getTab().getModelName());
    }

    public String getNextAction()
        throws Exception
    {
        return nextAction;
    }

    private void markSelectedEntitiesAsDeleted() throws Exception {
        Map values = new HashMap();
        values.put("deleted", !isRestore());
        for (int row: getSelected()) {
            Map key = (Map) getTab().getTableModel().get0bjectAt(row);
            try {
                MapFacade.setValues(
                    getTab().getModelName(),
                    key,
                    values);
            }
            catch (ValidationException ex) {
                addError("no_delete_row", row + 1, key);
                addErrors(ex.getErrors());
            }
            catch (Exception ex) {
                addError("no_delete_row", row + 1, key);
            }
        }
        getTab().deselectAll();
        resetDescriptionsCache();
    }

    public boolean isRestore() {
        return restore;
    }

    public void setRestore(boolean restore) {
        this.restore = restore;
    }
}
```

Listado A.18 LoadCurrentInvoiceAction.java

```
package org.openxava.invoicing.actions;

import java.util.*;
import javax.inject.*;
import org.openxava.actions.*;

public class LoadCurrentInvoiceAction
    extends SearchByViewKeyAction {

    @Inject
    private Map currentInvoiceKey;

    public void execute() throws Exception {
        getView().setValues(currentInvoiceKey);
    }
}
```

```

        super.execute();
    }

}

```

Listado A.19 NewOrderForCurrentUserAction.java

```

package org.openxava.invoicing.actions;

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

public class NewOrderForCurrentUserAction extends NewAction {

    public void execute() throws Exception {
        super.execute();
        String user = Users.getCurrent();
        if (user == null) {
            getView().setEditable(false);
            addError("no_user_logged");
            return;
        }
        int customerNumber = Integer.parseInt(user);

        View customerView = getView().getSubview("customer");
        customerView.setValue("number", customerNumber);

        customerView.findObject();
        customerView.setKeyEditable(false);
    }
}

```

Listado A.20 OnChangeSearchInvoiceAction.java

```

package org.openxava.invoicing.actions;

import java.util.*;
import org.openxava.actions.*;
import org.openxava.invoicing.model.*;
import org.openxava.model.*;
import org.openxava.view.*;

public class OnChangeSearchInvoiceAction
    extends OnChangeSearchAction {

    public void execute() throws Exception {
        super.execute();
        Map keyValues = getView()
            .getKeyValuesWithValue();
        if (keyValues.isEmpty()) return;
        Invoice invoice = (Invoice)
            MapFacade.findEntity(getView().getModelName(), keyValues);
        View customerView = getView().getRoot().getSubview("customer");
        int customerNumber = customerView.getValueInt("number");
        if (customerNumber == 0) {

```

299 Apéndice A: Código fuente

```
        customerView.setValue("number", invoice.getCustomer().getNumber());
        customerView.refresh();
    }
    else {
        if (customerNumber != invoice.getCustomer().getNumber()) {
            addError("invoice_customer_not_match",
                     invoice.getCustomer().getNumber(), invoice, customerNumber);
            getView().clear();
        }
    }
}
```

Listado A.21 SaveInvoiceAction.java

```
package org.openxava.invoicing.actions;

import org.openxava.actions.*;

public class SaveInvoiceAction
    extends SaveAction
    implements IChangeModuleAction
{

    public String getNextModule() {
        return PREVIOUS_MODULE;
    }

    public boolean hasReinitNextModule() {
        return false;
    }

}
```

Listado A.22 SearchExcludingDeletedAction

```
package org.openxava.invoicing.actions;

import java.util.*;
import javax.ejb.*;
import org.openxava.actions.*;

public class SearchExcludingDeletedAction
    extends SearchExecutingOnChangeAction {

    private boolean isDeletable() {
        return getView().getMetaModel()
            .containsMetaProperty("deleted");
    }

    protected Map getValuesFromView()
        throws Exception
    {
        if (!isDeletable())
            return super.getValuesFromView();
    }
}
```

```

        Map values = super.getValuesFromView();
        values.put("deleted", false);
        return values;
    }

    protected Map getMemberNames()
        throws Exception
    {
        if (!isDeletable()) {
            return super.getMemberNames();
        }
        Map members = super.getMemberNames();
        members.put("deleted", null);
        return members;
    }

    protected void setValuesToView(Map values)
        throws Exception
    {
        if (isDeletable() &&
            (Boolean) values.get("deleted")) {
            throw new ObjectNotFoundException();
        }
        else {
            super.setValuesToView(values);
        }
    }
}

```

Listado A.23 SearchInvoiceFromOrderAction.java

```

package org.openxava.invoicing.actions;

import org.openxava.actions.*;

public class SearchInvoiceFromOrderAction
    extends ReferenceSearchAction {

    public void execute() throws Exception {
        super.execute();
        int customerNumber =
            getPreviousView()
                .getValueInt("customer.number");

        if (customerNumber > 0) {
            getTab().setBaseCondition("${customer.number} = " + customerNumber);
        }
    }
}

```

Listado A.24 ShowHideCreateInvoiceAction.java

```

package org.openxava.invoicing.actions;

import org.openxava.actions.*;

```

301 Apéndice A: Código fuente

```
public class ShowHideCreateInvoiceAction
    extends OnChangePropertyBaseAction
    implements IShowActionAction,
               IHideActionAction {

    private boolean show;

    public void execute() throws Exception {
        show = isOrderCreated()
            && isDelivered()
            && !hasInvoice();
    }

    private boolean isOrderCreated() {
        return getView().getValue("oid") != null;
    }

    private boolean isDelivered() {
        Boolean delivered = (Boolean)
            getView().getValue("delivered");
        return delivered == null?false:delivered;
    }

    private boolean hasInvoice() {
        return getView().getValue("invoice.oid") != null;
    }

    public String getActionToShow() {
        return show?"Order.createInvoice":"";
    }

    public String getActionToHide() {
        return !show?"Order.createInvoice":"";
    }
}
```

A.3 Calculadores

Los calculadores contienen pequeños trozos de lógica que pueden ser utilizados y reutilizados a través de toda tu aplicación, especialmente para calcular los valores por defecto de las propiedades de las entidades.

Estas clases están en *Invoicing/src/org.openxava.invoicing.calculators*.

Listado A.25 PricePerUnitCalculator.java

```
package org.openxava.invoicing.calculators;

import org.openxava.calculators.*;
import org.openxava.invoicing.model.*;
import static org.openxava.jpa.XPersistence.*;

public class PricePerUnitCalculator implements ICalculator {

    private int productNumber;
```

```

public Object calculate() throws Exception {
    Product product = getManager()
        .find(Product.class, productNumber);
    return product.getPrice();
}

public void setProductNumber(int productNumber) {
    this.productNumber = productNumber;
}

public int getProductNumber() {
    return productNumber;
}

}

```

Listado A.26 VatPercentageCalculator.java

```

package org.openxava.invoicing.annotations;

import java.lang.annotation.*;
import org.hibernate.validator.*;
import org.openxava.invoicing.validators.*;

@ValidatorClass(ISBNValidator.class)
@Target({ ElementType.FIELD, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
public @interface ISBN {

    boolean search() default true;
    String message() default "ISBN does not exist";
}

```

A.4 Anotaciones

El conjunto de anotaciones incluido en OpenXava es suficiente para la mayoría de aplicaciones. Sin embargo, podemos crear nuestras propias anotaciones si es necesario. En esta aplicación hemos creado una anotación para una validación personalizada: @ISBN.

Esta anotación está en *Invoicing/src/org.openxava.invoicing.annotations*.

Listado A.27 ISBN.java

```

package org.openxava.invoicing.annotations;

import java.lang.annotation.*;
import org.hibernate.validator.*;
import org.openxava.invoicing.validators.*;

@ValidatorClass(ISBNValidator.class)
@Target({ ElementType.FIELD, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)

```

303 Apéndice A: Código fuente

```
public @interface ISBN {  
    boolean search() default true;  
    String message() default "ISBN does not exist";  
}
```

A.5 Validadores

Los validadores contienen la lógica de validación. En la aplicación Invoicing hemos escrito un validador, usando Hibernate Validator, con la lógica de validación a usar con las propiedades anotadas con @ISBN.

Esta clase está en *Invoicing/src/org.openxava.invoicing.validators*.

Listado A.28 ISBNValidator.java

```
package org.openxava.invoicing.validators;  
  
import org.apache.commons.logging.*;  
import org.hibernate.validator.*;  
import org.openxava.util.*;  
import org.openxava.invoicing.annotations.*;  
import com.gargoylesoftware.htmlunit.*;  
import com.gargoylesoftware.htmlunit.html.*;  
  
public class ISBNValidator implements Validator<ISBN> {  
  
    private static Log log = LogFactory.getLog(ISBNValidator.class);  
    private static org.apache.commons.validator.ISBNValidator  
        validator =  
            new org.apache.commons.validator.ISBNValidator();  
  
    private boolean search;  
  
    public void initialize(ISBN isbn) {  
        this.search = isbn.search();  
    }  
  
    public boolean isValid(Object value) {  
        if (Is.empty(value)) return true;  
        if (!validator.isValid(value.toString())) return false;  
        return search?isbnExists(value):true;  
    }  
  
    private boolean isbnExists(Object isbn) {  
        try {  
            WebClient client = new WebClient();  
            HtmlPage page = (HtmlPage) client.getPage(  
                "http://www.bookfinder4u.com/" +  
                "IsbnSearch.aspx?isbn=" +  
                isbn + "&mode=direct");  
            return page.asText()  
                .indexOf("ISBN: " + isbn) >= 0;  
        }  
        catch (Exception ex) {  
            log.warn("Impossible to connect to bookfinder4u" +
```

```
        "to validate the ISBN. Validation fails", ex);
    return false;
}
}
```

A.6 Clases de utilidad

En este caso sólo tenemos una clase de utilidad usada para leer las preferencias de la aplicación de un archivo de propiedades.

Esta clase se encuentra en *Invoicing/src/org.openxava.invoicing.util*.

Listado A.29 InvoicingPreferences.java

```
package org.openxava.invoicing.util;

import java.io.*;
import java.math.*;
import java.util.*;
import org.apache.commons.logging.*;
import org.openxava.util.*;

public class InvoicingPreferences {

    private final static String
        FILE_PROPERTIES="invoicing.properties";
    private static Log log =
        LogFactory.getLog(InvoicingPreferences.class);
    private static Properties properties;

    private static Properties getProperties() {
        if (properties == null) {
            PropertiesReader reader =
                new PropertiesReader(
                    InvoicingPreferences.class,
                    FILE_PROPERTIES);
            try {
                properties = reader.get();
            }
            catch (IOException ex) {
                log.error(
                    XavaResources.getString(
                        "properties_file_error",
                        FILE_PROPERTIES),
                    ex);
                properties = new Properties();
            }
        }
        return properties;
    }

    public static BigDecimal getDefaultVatPercentage() {
        return new BigDecimal(
            getProperties().getProperty("defaultVatPercentage"));
    }
}
```

```
}
```

A.7 Pruebas JUnit

Las pruebas JUnit verifican automáticamente que la aplicación funciona correctamente. Tenemos una prueba por módulo.

Estas clases se encuentran en *Invoicing/src/org.openxava.invoicing.tests*.

Listado A.30 AuthorTest.java

```
package org.openxava.invoicing.tests;

import org.openxava.tests.*;

public class AuthorTest extends ModuleTestCase {

    public AuthorTest(String testName) {
        super(testName, "Invoicing", "Author");
    }

    public void testReadAuthor() throws Exception {
        assertEqualsInList(0, 0, "JAVIER CORCOBADO");

        execute("Mode.detailAndFirst");

        assertEquals("name", "JAVIER CORCOBADO");
        assertEqualsRowCount("products", 2);
        assertEqualsInCollection("products", 0,
            "number", "2");
        assertEqualsInCollection("products", 0,
            "description", "Arco iris de lágrimas");
        assertEqualsInCollection("products", 1, "number", "3");
        assertEqualsInCollection("products", 1,
            "description", "Ritmo de sangre");
    }

}
```

Listado A.31 CategoryTest.java

```
package org.openxava.invoicing.tests;

import org.openxava.tests.*;

public class CategoryTest extends ModuleTestCase {

    public CategoryTest(String testName) {
        super(testName, "Invoicing", "Category");
    }

    public void testCategoriesInList() throws Exception {
        assertEqualsInList(0, 0, "MUSIC");
        assertEqualsInList(1, 0, "BOOKS");
        assertEqualsInList(2, 0, "SOFTWARE");
    }
}
```

{

Listado A.32 CommercialDocumentTest.java

```
package org.openxava.invoicing.tests;

import java.text.*;
import java.util.*;
import javax.persistence.*;
import org.openxava.tests.*;
import org.openxava.util.*;
import static org.openxava.jpa.XPersistence.*;

abstract public class CommercialDocumentTest extends ModuleTestBase {

    private String number;
    private String model;

    public CommercialDocumentTest(String testName, String moduleName) {
        super(testName, "Invoicing", moduleName);
        this.model = moduleName;
    }

    public void testCreate() throws Exception {
        calculateNumber();
        verifyDefaultValues();
        chooseCustomer();
        addDetails();
        setOtherProperties();
        save();
        verifyAmountAndEstimatedProfit();
        verifyCreated();
        remove();
    }

    private void verifyDefaultValues() throws Exception {
        execute("CRUD.new");
        assertEquals("year", getCurrentYear());
        assertEquals("number", "");
        assertEquals("date", getCurrentDate());
        assertEquals("vatPercentage", "18");
    }

    private void chooseCustomer() throws Exception {
        setValue("customer.number", "1");
        assertEquals("customer.name", "FRANCISCO JAVIER PANIZA LUCAS");
    }

    private void addDetails() throws Exception {
        assertCollectionRowCount("details", 0);
        execute("Collection.new",
            "viewObject=xava_view_section0_details");
        setValue("product.number", "1");
        assertEquals("product.description",
            "Peopleware: Productive Projects and Teams");
        assertEquals("pricePerUnit",
            "31.00");
    }
}
```

307 Apéndice A: Código fuente

```
        setValue("quantity", "2");
        assertEquals("amount", "62.00");
        execute("Collection.save");
        assertNoErrors();
        assertCollectionRowCount("details", 1);
        assertEquals("number", getNumber());

        assertEquals("baseAmount", "62.00");
        assertEquals("vat", "11.16");
        assertEquals("totalAmount", "73.16");

        execute("Collection.new",
            "viewObject=xava_view_section0_details");
        setValue("product.number", "2");
        assertEquals("product.description",
            "Arco iris de lágrimas");
        assertEquals("pricePerUnit",
            "15.00");
        setValue("pricePerUnit", "10.00");
        setValue("quantity", "1");
        assertEquals("amount", "10.00");
        execute("Collection.save");
        assertNoErrors();
        assertCollectionRowCount("details", 2);

        assertEquals("baseAmount", "72.00");
        assertEquals("vat", "12.96");
        assertEquals("totalAmount", "84.96");
    }

    private void verifyAmountAndEstimatedProfit() throws Exception {
        execute("Mode.list");
        setConditionValues(new String [] {
            getCurrentYear(), getNumber()
        });
        execute("List.filter");
        assertEqualsInList(0, 0, getCurrentYear());
        assertEqualsInList(0, 1, getNumber());
        assertEqualsInList(0, "amount", "84.96");
        assertEqualsInList(0, "estimatedProfit", "8.50");
        execute("Mode.detailAndFirst");
    }

    private void setOtherProperties() throws Exception {
        setValue("remarks", "This is a JUNIT test");
    }

    private void save() throws Exception {
        execute("CRUD.save");
        assertNoErrors();

        assertEquals("customer.number", "");
        assertCollectionRowCount("details", 0);
        assertEquals("remarks", "");
    }

    private void verifyCreated() throws Exception {
        assertEquals("year", getCurrentYear());
        assertEquals("number", getNumber());
        assertEquals("date", getCurrentDate());
    }
}
```

```

        assertEquals("customer.number", "1");
        assertEquals("customer.name", "FRANCISCO JAVIER PANIZA LUCAS");

        assertCollectionRowCount("details", 2);

        // Row 0
        assertEqualsInCollection("details", 0, "product.number", "1");
        assertEqualsInCollection("details", 0, "product.description",
            "Peopleware: Productive Projects and Teams");
        assertEqualsInCollection("details", 0, "quantity", "2");

        // Row 1
        assertEqualsInCollection("details", 1, "product.number", "2");
        assertEqualsInCollection("details", 1, "product.description",
            "Arco iris de lágrimas");
        assertEqualsInCollection("details", 1, "quantity", "1");

        assertEquals("remarks", "This is a JUNIT test");
    }

    public void testTrash() throws Exception {
        assertListOnlyOnePage();

        int initialRowCount = getListRowCount();
        String year1 = getValueInList(0, 0);
        String number1 = getValueInList(0, 1);
        execute("Mode.detailAndFirst");
        execute("Invoicing.delete");
        execute("Mode.list");

        assertListRowCount(initialRowCount - 1);
        assertDocumentNotInList(year1, number1);

        String year2 = getValueInList(0, 0);
        String number2 = getValueInList(0, 1);
        checkRow(0);
        execute("Invoicing.deleteSelected");
        assertListRowCount(initialRowCount - 2);
        assertDocumentNotInList(year2, number2);

        changeModule(model + "Trash");
        assertListOnlyOnePage();
        int initialTrashRowCount = getListRowCount();

        assertDocumentInList(year1, number1);
        assertDocumentInList(year2, number2);

        int row1 = getDocumentRowInList(year1, number1);
        execute("Trash.restore", "row=" + row1);
        assertListRowCount(initialTrashRowCount - 1);
        assertDocumentNotInList(year1, number1);

        int row2 = getDocumentRowInList(year2, number2);
        checkRow(row2);
        execute("Trash.restore");
        assertListRowCount(initialTrashRowCount - 2);
        assertDocumentNotInList(year2, number2);

        changeModule(model);
    }
}

```

309 Apéndice A: Código fuente

```
        assertListRowCount(initialRowCount);
        assertDocumentInList(year1, number1);
        assertDocumentInList(year2, number2);
    }

    private void assertListOnlyOnePage() throws Exception {
        assertListNotEmpty();
        assertTrue("Must be less than 10 rows to run this test",
                   getListRowCount() < 10);
    }

    private void assertDocumentNotInList(String year, String number)
        throws Exception
    {
        assertTrue(
            "Document " + year + "/" + number + " must not be in list",
            getDocumentRowInList(year, number) < 0);
    }

    protected void assertDocumentInList(String year, String number)
        throws Exception
    {
        assertTrue(
            "Document " + year + "/" + number + " must be in list",
            getDocumentRowInList(year, number) >= 0);
    }

    protected int getDocumentRowInList(String year, String number)
        throws Exception
    {
        int c = getListRowCount();
        for (int i=0; i<c; i++) {
            if (year.equals(getValueInList(i, 0)) &&
                number.equals(getValueInList(i, 1)))
            {
                return i;
            }
        }
        return -1;
    }

    private void remove() throws Exception {
        execute("Invoicing.delete");
        assertNoErrors();
    }

    private String getCurrentYear() {
        return new SimpleDateFormat("yyyy").format(new Date());
    }

    private String getCurrentDate() {
        return DateFormat.getDateInstance(
            DateFormat.SHORT).format(new Date());
    }

    private void calculateNumber() {
        Query query = getManager().
            createQuery(
                "select max(i.number) from " +
```

```

        model +
        " i where i.year = :year");
query.setParameter("year", Dates.getYear(new Date()));
Integer lastNumber = (Integer) query.getSingleResult();
if (lastNumber == null) lastNumber = 0;
number = Integer.toString(lastNumber + 1);
}

private String getNumber() {
    return number;
}

protected void assertCustomerInList(String customerNumber)
    throws Exception
{
    assertValueForAllRows(3, customerNumber);
}

protected void assertValueForAllRows(int column, String value)
    throws Exception
{
    assertListNotEmpty();
    int c = getListRowCount();
    for (int i=0; i<c; i++) {
        if (!value.equals(
            getValueInList(i, column)))
        {
            fail("Column " + column + " in row " + i + " is not " + value);
        }
    }
}

}

```

Listado A.33 CustomerOrdersTest.java

```

package org.openxava.invoicing.tests;

import org.openxava.tests.*;

public class CustomerOrdersTest extends ModuleTestCase {

    public CustomerOrdersTest(String testName) {
        super(testName, "Invoicing", "CustomerOrders");
    }

    public void testLimitingDataVisibility() throws Exception {
        login("lulu@invoicing.com", "lulu");

        assertListNotEmpty();
        int rowCount = getListRowCount();
        for (int row=0; row<rowCount; row++) {
            assertValueInList(row, "customer.name", "LULU SEMUA");
        }

        execute("CustomerOrders.new");
        assertNoEditable("customer.number");
    }
}

```

3.11 Apéndice A: Código fuente

```
        assertEquals("customer.name", "LULU SEMUA");

    logout();
}

}
```

Listado A.34 CustomerTest.java

```
package org.openxava.invoicing.tests;

import org.openxava.tests.*;

public class CustomerTest extends ModuleTestCase {

    public CustomerTest(String testName) {
        super(testName,
              "Invoicing",
              "Customer");
    }

    public void testCreateReadUpdateDelete() throws Exception {

        // Create
        execute("CRUD.new");
        setValue("number", "77");
        setValue("name", "JUNIT Customer");
        setValue("address.street", "JUNIT Street");

        setValue("address.zipCode", "77555");
        setValue("address.city", "The JUNIT city");
        setValue("address.state", "The JUNIT state");

        execute("CRUD.save");
        assertNoErrors();
        assertEquals("number", "");
        assertEquals("name", "");
        assertEquals("address.street", "");
        assertEquals("address.zipCode", "");
        assertEquals("address.city", "");
        assertEquals("address.state", "");

        // Read
        setValue("number", "77");
        execute("CRUD.refresh");
        assertEquals("number", "77");
        assertEquals("name", "JUNIT Customer");
        assertEquals("address.street", "JUNIT Street");
        assertEquals("address.zipCode", "77555");
        assertEquals("address.city", "The JUNIT city");
        assertEquals("address.state", "The JUNIT state");

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

```

    // Verify if modified
    setValue("number", "77");
    execute("CRUD.refresh");
    assertEquals("number", "77");
    assertEquals("name", "JUNIT Customer MODIFIED");

    // Delete
    execute("Invoicing.delete");
    assertMessage("Customer deleted successfully");
}

}

```

Listado A.35 InvoiceTest.java

```

package org.openxava.invoicing.tests;

public class InvoiceTest extends CommercialDocumentTest {

    public InvoiceTest(String testName) {
        super(testName, "Invoice");
    }

    public void testAddOrders() throws Exception {
        assertListNotEmpty();
        execute("List.orderBy", "property=number");
        execute("Mode.detailAndFirst");
        String customerNumber = getValue("customer.number");
        deleteDetails();
        assertCollectionRowCount("details", 0);
        assertEquals("baseAmount", "0.00");
        execute("Sections.change", "activeSection=1");
        assertCollectionRowCount("orders", 0);
        execute("Invoice.addOrders",
               "viewObject=xava_view_section1_orders");
        assertCustomerInList(customerNumber);
        assertEqualsForAllRows(5, "Yes");
        String firstOrderBaseAmount = getValueInList(0, 8);
        int ordersRowCount = getListRowCount();
        execute("AddOrdersToInvoice.add", "row=0");
        assertMessage("1 element(s) added to Orders of Invoice");
        assertCollectionRowCount("orders", 1);
        execute("Sections.change", "activeSection=0");
        assertCollectionNotEmpty("details");
        assertEquals("baseAmount", firstOrderBaseAmount);
        execute("Sections.change", "activeSection=1");
        execute("Invoice.addOrders",
               "viewObject=xava_view_section1_orders");
        assertListRowCount(ordersRowCount - 1);
        execute("AddToCollection.cancel");
        checkRowCollection("orders", 0);
        execute("Collection.removeSelected",
               "viewObject=xava_view_section1_orders");
        assertCollectionRowCount("orders", 0);
    }
}

```

313 Apéndice A: Código fuente

```
private void deleteDetails() throws Exception {
    int c = getCollectionRowCount("details");
    for (int i=0; i<c; i++) {
        checkRowCollection("details", i);
    }
    execute("Collection.removeSelected",
            "viewObject=xava_view_section0_details");
}

}
```

Listado A.36 OrderTest.java

```
package org.openxava.invoicing.tests;

import java.math.*;
import java.util.*;
import javax.persistence.*;
import org.openxava.invoicing.model.*;
import org.openxava.jpa.*;
import org.openxava.util.*;

public class OrderTest extends CommercialDocumentTest {

    public OrderTest(String testName) {
        super(testName, "Order");
    }

    public void testSetInvoice() throws Exception {
        assertListNotEmpty();
        execute("List.orderBy", "property=number");
        execute("Mode.detailAndFirst");
        assertEquals("delivered", "false");
        execute("Sections.change", "activeSection=1");
        assertEquals("invoice.number", "");
        assertEquals("invoice.year", "");
        execute("Order.searchInvoice",
                "keyProperty=invoice.number");
        execute("List.orderBy", "property=number");
        String year = getValueInList(0, "year");
        String number = getValueInList(0, "number");
        execute("ReferenceSearch.choose", "row=0");
        assertEquals("invoice.year", year);
        assertEquals("invoice.number", number);

        // Not delivered order cannot have invoice
        execute("CRUD.save");
        assertErrorsCount(1);
        setValue("delivered", "true");
        execute("CRUD.save");
        assertNoErrors();

        // Order with invoice cannot be deleted
        execute("Mode.list");
        execute("Mode.detailAndFirst");
        execute("Invoicing.delete");
        assertErrorsCount(1);
    }
}
```

```

// Restoring original values
setValue("delivered", "false");
setValue("invoice.year", "");
setValue("invoice.number", "");
execute("CRUD.save");
assertNoErrors();
}

public void testCreateInvoiceFromOrder() throws Exception {
    // Looking for the order
    searchOrderSusceptibleToBeInvoiced();
    assertEquals("delivered", "true");
    int orderDetailsCount = getCollectionRowCount("details");

    execute("Sections.change", "activeSection=1");
    assertEquals("invoice.year", "");
    assertEquals("invoice.number", "");

    // Creating the invoice
    execute("Order.createInvoice");
    String invoiceYear = getValue("invoice.year");
    assertTrue("Invoice year must have value",
        !IsEmptyString(invoiceYear));
    String invoiceNumber = getValue("invoice.number");
    assertTrue("Invoice number must have value",
        !IsEmptyString(invoiceNumber));
    assertMessage("Invoice " + invoiceYear + "/" + invoiceNumber +
        " created from current order");
    assertCollectionRowCount("invoice.details",
        orderDetailsCount);

    // Restoring the order for running the test the next time
    setValue("invoice.year", "");
    setValue("invoice.number", "");
    assertEquals("invoice.number", "");
    assertCollectionRowCount("invoice.details", 0);
    execute("CRUD.save");
    assertNoErrors();
}

public void testSearchInvoiceFromOrder() throws Exception {
    execute("CRUD.new");
    setValue("customer.number", "1");
    execute("Sections.change", "activeSection=1");
    execute("Order.searchInvoice",
        "keyProperty=invoice.number");
    assertCustomerInList("1");
    execute("ReferenceSearch.cancel");
    execute("Sections.change", "activeSection=0");
    setValue("customer.number", "2");
    execute("Sections.change", "activeSection=1");
    execute("Order.searchInvoice",
        "keyProperty=invoice.number");
    assertCustomerInList("2");
}

public void testOnChangeInvoice() throws Exception {
    execute("CRUD.new");
    assertEquals("customer.number", "");
    execute("Sections.change", "activeSection=1");
}

```

315 Apéndice A: Código fuente

```
execute("Order.searchInvoice",
        "keyProperty=invoice.number");

execute("List.orderBy", "property=customer.number");
String customer1Number = getValueInList(0, "customer.number");
String invoiceYear1 = getValueInList(0, "year");
String invoiceNumber1 = getValueInList(0, "number");
execute("List.orderBy", "property=customer.number");
String customer2Number = getValueInList(0, "customer.number");
String customer2Name = getValueInList(0, "customer.name");

assertNotEquals("Must be invoices of different customer",
                customer1Number, customer2Number);

execute("ReferenceSearch.choose", "row=0");
execute("Sections.change", "activeSection=0");
assertValue("customer.number", customer2Number);
assertValue("customer.name", customer2Name);

execute("Sections.change", "activeSection=1");
setValue("invoice.year", invoiceYear1);
setValue("invoice.number", invoiceNumber1);

assertError("Customer Nº " + customer1Number + " of invoice " +
            invoiceYear1 + "/" + invoiceNumber1 +
            " does not match with Customer Nº " +
            customer2Number + " of the current order");

assertValue("invoice.year", "");
assertValue("invoice.number", "");
assertValue("invoice.date", "");
}

public void testHidesCreateInvoiceFromOrderWhenNotApplicable() throws Exception {
    searchOrderUsingList(
        "delivered = true and invoice <> null");
    assertNoAction("Order.createInvoice");

    execute("Mode.list");

    searchOrderUsingList(
        "delivered = false and invoice = null");
    assertNoAction("Order.createInvoice");

    execute("CRUD.new");
    assertNoAction("Order.createInvoice");
}

public void testCreateInvoiceFromSelectedOrders() throws Exception {
    assertOrder(2010, 9, 2, 362);
    assertOrder(2010, 10, 1, 126);

    execute("List.orderBy", "property=number");
    checkRow(
        getDocumentRowInList("2010", "9"))
    );
    checkRow(
        getDocumentRowInList("2010", "10"))
}
```

```

);

execute("Order.createInvoiceFromSelectedOrders");

String invoiceYear = getValue("year");
String invoiceNumber = getValue("number");
assertMessage("Invoice " + invoiceYear + "/" + invoiceNumber +
    " created from orders: [2010/9, 2010/10]");
assertCollectionRowCount("details", 3);

assertValue("baseAmount", "488.00");
execute("Sections.change", "activeSection=1");
assertCollectionRowCount("orders", 2);
assertValueInCollection("orders", 0, 0, "2010");
assertValueInCollection("orders", 0, 1, "9");
assertValueInCollection("orders", 1, 0, "2010");
assertValueInCollection("orders", 1, 1, "10");

assertAction("CurrentInvoiceEdition.save");
assertAction("CurrentInvoiceEdition.return");

checkRowCollection("orders", 0);
checkRowCollection("orders", 1);
execute("Collection.removeSelected",
    "viewObject=xava_view_section1_orders");
assertNoErrors();

execute("CurrentInvoiceEdition.return");
assertDocumentInList("2010", "9");
assertDocumentInList("2010", "10");
}

public void testCreateInvoiceFromOrderExceptions() throws Exception {
    assertCreateInvoiceFromOrderException(
        "delivered = true and invoice <> null",
        "Impossible to create invoice: the order already has an invoice"
    );

    assertCreateInvoiceFromOrderException(
        "delivered = false and invoice = null",
        "Impossible to create invoice: the order is not delivered yet"
    );
}

private void assertCreateInvoiceFromOrderException(
    String condition, String message) throws Exception
{
    Order order = findOrder(condition);
    int row = getDocumentRowInList(
        String.valueOf(order.getYear()),
        String.valueOf(order.getNumber())
    );
    checkRow(row);
    execute("Order.createInvoiceFromSelectedOrders");
    assertError(message);
    uncheckRow(row);
}

private void assertOrder(
    int year, int number, int detailsCount, int baseAmount)
}

```

317 Apéndice A: Código fuente

```
{  
    Order order = findOrder("year = " + year + " and number=" + number);  
    assertEquals("To run this test the order " +  
        order + " must have " + detailsCount + " details",  
        detailsCount, order.getDetails().size());  
    assertTrue("To run this test the order " +  
        order + " must have " + baseAmount + " as base amount",  
        order.getBaseAmount().compareTo(new BigDecimal(baseAmount)) == 0);  
}  
  
private void searchOrderSusceptibleToBeInvoiced() throws Exception {  
    searchOrderUsingList("o.delivered = true and o.invoice = null");  
}  
  
private void searchOrderUsingList(String condition) throws Exception {  
  
    Order order = findOrder(condition);  
    String year = String.valueOf(order.getYear());  
    String number = String.valueOf(order.getNumber());  
    setConditionValues(new String [] { year, number });  
    execute("List.filter");  
    assertEqualsRowCount(1);  
    execute("Mode.detailAndFirst");  
    assertEquals("year", year);  
    assertEquals("number", number);  
}  
  
private Order findOrder(String condition) {  
    Query query = XPersistence.getManager().createQuery(  
        "from Order o where o.deleted = false and "  
        + condition);  
    List orders = query.getResultList();  
    if (orders.isEmpty()) {  
        fail("To run this test you must have some order with " + condition);  
    }  
    return (Order) orders.get(0);  
}  
}
```

Listado A.37 ProductTest.java

```
package org.openxava.invoicing.tests;  
  
import java.math.*;  
import org.openxava.invoicing.model.*;  
import org.openxava.tests.*;  
import static org.openxava.jpa.XPersistence.*;  
  
public class ProductTest extends ModuleTestCase {  
  
    private Author author;  
    private Category category;  
    private Product product1;  
    private Product product2;  
  
    public ProductTest(String testName) {  
        super(testName, "Invoicing", "Product");  
    }
```

```
}

protected void setUp() throws Exception {
    createProducts();
    super.setUp();
}

protected void tearDown() throws Exception {
    super.tearDown();
    removeProducts();
}

public void testRemoveFromList() throws Exception {
    setConditionValues(
        new String[] { "", "JUNIT" });
    setConditionComparators(
        new String[] { "=", "contains_comparator" });
    execute("List.filter");
    assertListRowCount(2);
    checkRow(1);
    execute("Invoicing.deleteSelected");
    assertListRowCount(1);
}

public void testChangePrice() throws Exception {
    // Searching the product1
    execute("CRUD.new");
    setValue("number", Integer.toString(product1.getNumber()));
    execute("CRUD.refresh");
    assertEquals("price", "10.00");

    // Changing the price
    setValue("price", "12.00");
    execute("CRUD.save");
    assertNoErrors();
    assertEquals("price", "");

    // Verifying
    setValue("number", Integer.toString(product1.getNumber()));
    execute("CRUD.refresh");
    assertEquals("price", "12.00");
}

private void createProducts() {
    // Creating the Java objects
    author = new Author();
    author.setName("JUNIT Author");
    category = new Category();
    category.setDescription("JUNIT Category");

    product1 = new Product();
    product1.setNumber(900000001);
    product1.setDescription("JUNIT Product 1");
    product1.setAuthor(author);
    product1.setCategory(category);
    product1.setPrice(new BigDecimal("10"));

    product2 = new Product();
    product2.setNumber(900000002);
    product2.setDescription("JUNIT Product 2");
}
```

319 Apéndice A: Código fuente

```
product2.setAuthor(author);
product2.setCategory(category);
product2.setPrice(new BigDecimal("20"));

// Marking as persistent objects
getManager().persist(author);
getManager().persist(category);
getManager().persist(product1);
getManager().persist(product2);

// Commit changes to the database
commit();
}

public void testISBNValidator() throws Exception {
    execute("CRUD.new");
    setValue("number", Integer.toString(product1.getNumber()));
    execute("CRUD.refresh");
    assertEquals("description", "JUNIT Product 1");
    assertEquals("isbn", "");

    setValue("isbn", "1111");
    execute("CRUD.save");
    assertError("1111 is not a valid value for Isbn of " +
               "Product: ISBN does not exist");

    setValue("isbn", "1234367890");
    execute("CRUD.save");
    assertError("1234367890 is not a valid value for Isbn of " +
               "Product: ISBN does not exist");

    setValue("isbn", "0932633439");
    execute("CRUD.save");
    assertNoErrors();
}

private void removeProducts() {
    remove(product1, product2, author, category);
    commit();
}

private void remove(Object ... entities) {
    for (Object entity: entities) {
        getManager().remove(getManager().merge(entity));
    }
}
}
```

A.8 Definición de la aplicación

Aunque OpenXava genera un módulo por defecto por cada entidad en el código, tenemos la posibilidad de refinarlos o definir nuevos usando el archivo *application.xml*.

El archivo *application.xml* se encuentra en *Invoicing/xava*.

Listado A.38 application.xml

```

<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE application SYSTEM "dtds/application.dtd">

<application name="Invoicing">

    <default-module>
        <controller name="Invoicing"/>
    </default-module>

    <module name="ConsultInvoice">
        <env-var name="XAVA_SEARCH_ACTION"
            value="CRUD.searchReadOnly"/>
        <model name="Invoice"/>
        <controller name="Print"/>
    </module>

    <module name="CustomerOrders">
        <env-var name="XAVA_SEARCH_ACTION" value="CRUD.searchReadOnly"/>
        <model name="Order"/>
        <tab name="CurrentCustomer"/>
        <controller name="CustomerOrders"/>
        <controller name="Print"/>
    </module>

    <module name="InvoiceTrash">
        <env-var name="XAVA_LIST_ACTION"
            value="Trash.restore"/>
        <model name="Invoice"/>
        <tab name="Deleted"/>
        <controller name="Trash"/>
        <mode-controller name="ListOnly"/>
    </module>

    <module name="OrderTrash">
        <env-var name="XAVA_LIST_ACTION" value="Trash.restore"/>
        <model name="Order"/>
        <tab name="Deleted"/>
        <controller name="Trash"/>
        <mode-controller name="ListOnly"/>
    </module>

    <module name="CurrentInvoiceEdition">
        <model name="Invoice"/>
        <controller name="CurrentInvoiceEdition"/>
        <mode-controller name="Void"/>
    </module>

</application>

```

A.9 Definición de los controladores

Un controlador es una colección de acciones. Aunque OpenXava incluye un conjunto de controladores listo para usar, hemos definido los nuestros propios para la aplicación Invoicing en *controllers.xml*.

321 Apéndice A: Código fuente

El archivo *controllers.xml* se encuentra en *Invoicing/xava*.

Listado A.39 controllers.xml

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE controllers SYSTEM "dtds/controllers.dtd">
<controllers>

<env-var
  name="XAVA_SEARCH_ACTION"
  value="Invoicing.searchExcludingDeleted"/>

<object name="invoicing_currentInvoiceKey"
  class="java.util.Map"
  scope="global"/>

<controller name="Invoicing">
  <extends controller="Typical"/>

  <action name="delete"
    mode="detail" confirm="true"
    class="org.openxava.invoicing.actions.InvoicingDeleteAction"
    image="images/delete.gif"
    keystroke="Control D"/>

  <action name="searchExcludingDeleted"
    hidden="true"
    class=
      "org.openxava.invoicing.actions.SearchExcludingDeletedAction"/>

  <action name="deleteSelected" mode="list" confirm="true"
    class="org.openxava.invoicing.actions.InvoicingDeleteSelectedAction"
    keystroke="Control D"/>

  <action name="deleteRow" mode="NONE" confirm="true"
    class="org.openxava.invoicing.actions.InvoicingDeleteSelectedAction"
    image="images/delete.gif"
    in-each-row="true"/>

</controller>

<controller name="Trash">
  <action name="restore" mode="list"
    class=
      "org.openxava.invoicing.actions.InvoicingDeleteSelectedAction">
    <set property="restore" value="true"/>
  </action>
</controller>

<controller name="Order">
  <extends controller="Invoicing"/>

  <action name="createInvoice" mode="detail"
    class=
      "org.openxava.invoicing.actions.CreateInvoiceFromOrderAction"/>

  <action name="createInvoiceFromSelectedOrders"
    mode="list"
```

```
        class=
    "org.openxava.invoicing.actions.CreateInvoiceFromSelectedOrdersAction"
    />

    <action name="searchInvoice"
        class="org.openxava.invoicing.actions.SearchInvoiceFromOrderAction"
        hidden="true" image="images/search.gif"/>

</controller>

<controller name="CurrentInvoiceEdition">

    <action name="load"
        class=
            "org.openxava.invoicing.actions.LoadCurrentInvoiceAction"
        hidden="true" on-init="true"/>

    <action name="save"
        class="org.openxava.invoicing.actions.SaveInvoiceAction"
        keystroke="Control S"/>

    <action name="return"
        class="org.openxava.actions.ReturnPreviousModuleAction"/>

</controller>

<controller name="Invoice">
    <extends controller="Invoicing"/>

    <action name="addOrders"
        class="org.openxava.invoicing.actions.GoAddOrdersToInvoiceAction"
        hidden="true" image="images/create_new.gif"/>

</controller>

<controller name="AddOrdersToInvoice">
    <extends controller="AddToCollection"/>

    <action
        name="add"
        class="org.openxava.invoicing.actions.AddOrdersToInvoiceAction"/>

</controller>

<controller name="CustomerOrders">

    <action name="new"
        class="org.openxava.invoicing.actions.NewOrderForCurrentUserAction"
        image="images/new.gif"
        keystroke="Control N"/>

    <action name="save" mode="detail"
        class="org.openxava.actions.SaveAction"
        image="images/save.gif"
        keystroke="Control S"/>

</controller>

</controllers>
```

A.10 Internacionalización

Los archivos de internacionalización (i18n) contienen el texto de las etiquetas y mensajes usados en la aplicación.

Estos archivos se encuentran en *Invoicing/i18n*.

Listado A.40 Invoicing-labels_en.properties

```
Invoicing=Invoicing
Product.views.Simple.number=Product Nº
Product.views.Simple.description=Product
```

Listado A.41 Invoicing-messages_en.properties

```
order_must_be_delivered=Order {0}/{1} must be delivered in order to be added to
an Invoice
cannot_delete_order_with_invoice=An order with an invoice cannot be deleted
invoice_created_from_order=Invoice {0} created from current order
impossible_create_invoice_order_already_has_one=Impossible to create invoice:
the order already has an invoice
impossible_create_invoice_order_is_not_delivered=Impossible to create invoice:
the order is not delivered yet
impossible_create_invoice=Impossible to create invoice
impossible_create_invoice_order_not_exist=Impossible to create invoice: the
order does not exist yet
invoice_created_from_orders=Invoice {0} created from orders: {1}
impossible_create_invoice_orders_not_specified=Impossible to create invoice:
orders not specified
impossible_copy_details_to_invoice=Impossible to copy details from order to
invoice
invoice_customer_not_match=Customer Nº {0} of invoice {1} does not match with
Customer Nº {2} of the current order
```


Aprender más

apéndice B

Este apéndice contiene algunos recursos para aprender más sobre OpenXava.

Guía de referencia

Una referencia completa de toda la sintaxis y semántica de OpenXava. Está incluida en la distribución de OpenXava y en el wiki:

http://openxava.wikispaces.com/reference_es

Disponible en inglés, español, francés, chino y ruso.

API

La documentación del API de todas las clases y anotaciones de OpenXava se incluye en la distribución de OpenXava y en el sitio web:

<http://www.openxava.org/OpenXavaDoc/apidocs/>

Wiki

La documentación oficial de OpenXava está en un wiki público:

<http://openxava.wikispaces.com/>

Foros

Puedes aprender mucho de otros desarrolladores OpenXava en los foros:

Foro de ayuda: <http://sourceforge.net/projects/openxava/forums/forum/437013>

Foro de debate: <http://sourceforge.net/projects/openxava/forums/forum/437015>

Sitio de OpenXava

En el sitio de OpenXava tendrás las últimas noticias, descargas, recursos y vínculos útiles:

<http://www.openxava.org>