Introducción al desarrollo de aplicaciones de escritorio con Swing Application Framework

Miguel Velasco (jVel)

Contenido

Introducción 5
- Tecnología 5
- La aplicación 5
- El modelo de datos 5

Desarrollo 6
- Creando el Proyecto 6
- Actualizando la información de la aplicación 8
- Creando una pantalla de edición de datos 8
- Creando las entidades JPA 9
- Accediendo a los datos 11
- Incluyendo el Driver de Oracle 21
- Añadiendo barras de progreso 23
- Creando una vista maestro-detalle 24
- Creando un diálogo para editar datos 26
- Añadiendo disparadores para cambios de propiedades 29
- Utilizando el diálogo 30
- Añadiendo un renderizador para los ComboBox 32
- Añadiendo notificación al seleccionar temas 33
- Continuando con el maestro-detalle 35
- Completando la aplicación 41

Conclusiones 42

Referencias 43

Modelo de datos 1 5
Nuevo proyecto 2 6
Nueva aplicación de escritorio 3 7
Navegador de proyectos 4 7
Propiedades de la aplicación 5 8
Nuevo panel 6 9
Creación de entidades 7 10
Creación de entidades 8 10
Creación de entidades 9 11
Nuevo entity manager 10 12
Código del entity manager 11 12
Nuevo bean 12 13
Propiedad observable 13 13
Propiedad type parameters 14 14
Código de creación de la lista 15 14
Código personalizado 16 15
Named Queries 17 15
Enlace de datos 18 16
Definición de columnas 19 16
Nuevo action 20 17
Código addState 21 17
Código deleteState 22 18
Código saveStates 23 18
Código isStateSelected 24 18
Código listSelectionListener 25 19
Código deleteState 26 19
Nuevo action 27 20
Código statesAdministration 28 20
Anotaciones secuencia 29 20
Configuración de la unidad de persistencia 30 21
Driver de Oracle 31 22
Aplicación 32 23
Código SaveStatesTask 33 24
Código saveStates 34 24
Inspección del TasksPanel 35 25
Código de creación de la lista de temas 36 25
Columnas 37 26
Diseño de TaskDialog 39 27
Inspección de TaskDialog 38 27
Enlace de datos 40 28
Creación de getter y setter 41 28
Definición del PropertyChangeSupport 43 29
Código de sabe y cancel 42 29
Código de setNota 44 30
Código de propertyChangeListeners 45 30
Código de addTask 46 30
Código de modifyTask 47 31
Código de deleteTask 48 31
Menú 49 31
Aplicación 50 32
Código de StateCellRenderer 51 32
Edición de la propiedad renderer 52 33
Código para el propertyChangeSupport 53 34

Aplicación 54 35
Columnas 55 36
Código de listSelectionListener 56 36
Navegador de librerías 57 37
Diálogo 58 37
Propiedad calendar 59 38
Código de NotasDialog 60 39
Código addNote 62 40
Código NotasDialog 61 39
Código modifyNote 63 41
Código deleteNote 64 41
Anotación deleteCascade 65 42
Código repaintNotes 66 42

Introducción

Swing Application Framework es una iniciativa que, como su nombre indica, pretende construir un framework para el desarrollo de aplicaciones de escritorio utilizando tecnología Swing. Su desarrollo se está haciendo mediante la JSR-296 que, lamentablemente, se encuentra actualmente (agosto de 2010) inactiva.

No obstante, la última versión del IDE NetBeans, la 6.9.1, viene preparada para trabajar con esta tecnología, que ofrece una forma de trabajo muy interesante para aquellos que desarrollan aplicaciones de escritorio en Java.

En este tutorial vamos a desarrollar una sencilla herramienta para gestión de tareas, fijándonos en algunas de las aportaciones del framework.

Tecnología

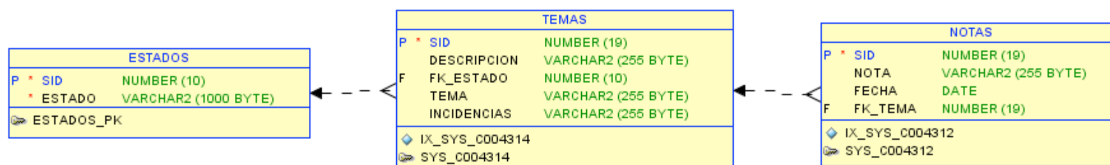
Para el desarrollo trabajaremos con NetBeans 6.9.1, y con una base de datos Oracle. La vista de la aplicación se hará lógicamente utilizando Swing, mientras que el acceso a datos lo haremos por medio de JPA.

La aplicación

Desarrollaremos una sencilla aplicación de gestión de tareas, que nos permitirá gestionar los temas que tenemos pendientes en nuestro trabajo, e ir añadiendo información con la que facilitar su seguimiento.

El modelo de datos

El modelo de datos de nuestra aplicación se compondrá de tres tablas en las que se almacenará información acerca de los temas o tareas, y de sus estados, y que nos permitirá asociar anotaciones a los temas.



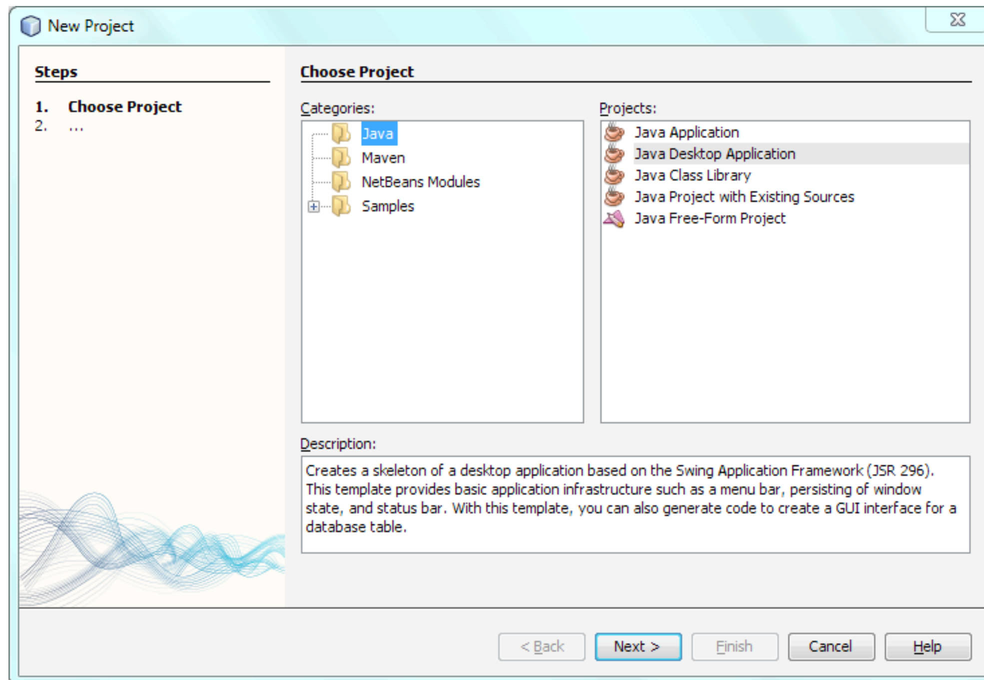
Modelo de datos 1

En los recursos adjuntos al tutorial están los scripts para generar las tablas (carpeta scripts del proyecto).

Desarrollo

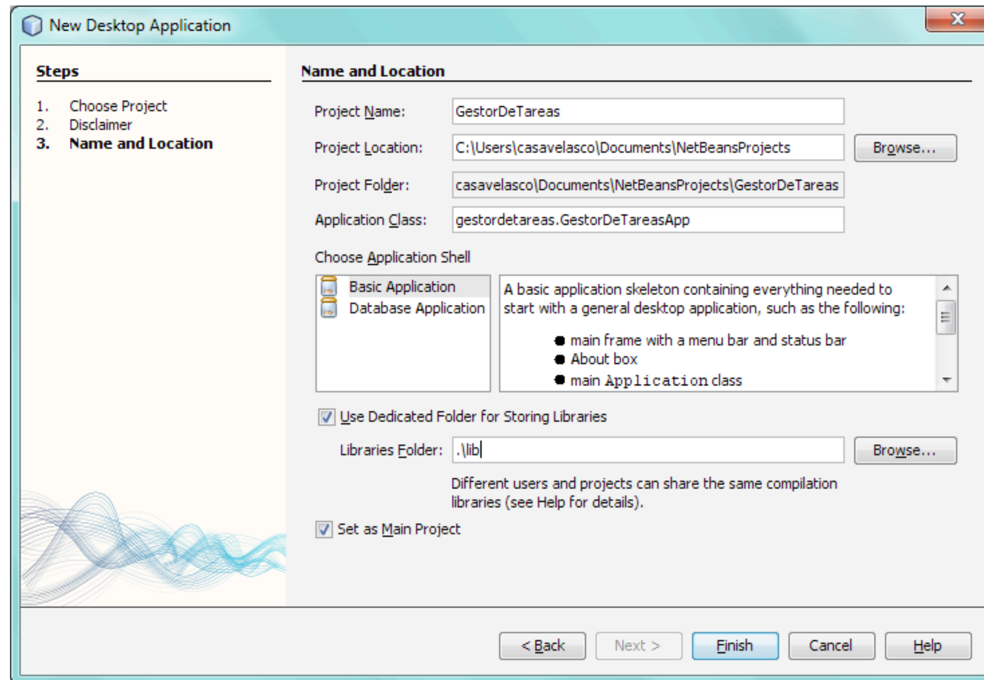
Creando el Proyecto

Para comenzar el desarrollo, lo primero que tenemos que hacer es crear un nuevo proyecto en NetBeans, accediendo a la opción File – New Project.



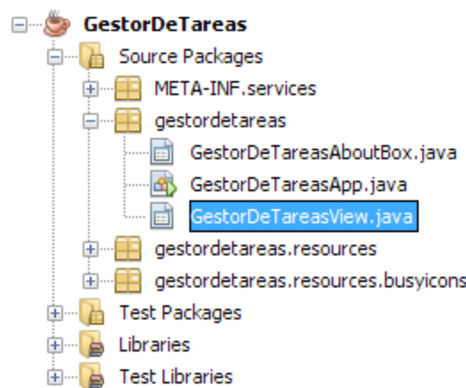
Nuevo proyecto 2

Como se puede observar, elegiremos, dentro de la categoría Java, el tipo de proyecto Java Desktop Application. En el siguiente paso, tras ver la advertencia de que el desarrollo de la JSR del SAF ha sido detenido, podemos poner un nombre y seleccionar una ubicación para nuestro proyecto. Además tenemos que seleccionar la clase principal de la aplicación que vamos a crear. Por último, podemos elegir el tipo de aplicación: básica, o de acceso a datos. Seleccionaremos la opción básica para ir creando desde cero la mayor parte de los elementos que necesitamos. Una vez visto el tutorial, sería interesante probar la opción de acceso a datos, que crea gran parte de la estructura del proyecto, facilitando mucho las cosas.



Nueva aplicación de escritorio 3

El IDE crea automáticamente tres clases: GestorDeTareasApp, GestorDeTareasView, y GestorDeTareasAbautBox.



Navegador de proyectos 4

La clase GestorDeTareasApp es la clase principal de nuestra aplicación, y hereda de la clase SingleFrameApplication de SAF. Podemos ver que contiene un método main en el que se lanza la propia aplicación, y los métodos heredados startup y configureWindow, que serán llamados por el framework. En startup hay una simple llamada al método show para mostrar la vista de nuestra aplicación: GestorDeTareasView.

Si abrimos esta otra clase con el editor de texto, veremos que hereda de la clase FrameView de SAF. Esta será la ventana de nuestra aplicación, a la que NetBeans ya le ha puesto por nosotros una barra de menú y una barra de estado.

Por último, tenemos la clase GestorDeTareasAbautBox, que es el típico diálogo con información sobre la aplicación. Además de las tres clases, podemos ver que hay otro paquete, resources, en el que encontraremos un fichero properties por cada una de ellas.

Actualizando la información de la aplicación

Cuando trabajamos con SAF, el objetivo es que todos los textos se definan en un fichero de propiedades, y nunca en las propias clases, buscando las conocidas ventajas: facilidad para la internacionalización, no necesidad de recompilar código...

Un primer paso en el desarrollo puede ser sencillamente modificar los textos de la ventana de información sobre la aplicación, cambiando el fichero GestorDeTareasApp.properties.

```
# Application global resources

Application.name = Gestor de Tareas
Application.title = Gestor de Tareas
Application.version = 1.0
Application.vendor = jVel
Application.homepage = http\://tecno-aspirinas.blogspot.com
Application.description = Una aplicación de escritorio Java para gestionar tareas, basada en Swing Application Framework.
Application.vendorId = jVel
Application.id = ${Application.name}
Application.lookAndFeel = system
```

Propiedades de la aplicación 5

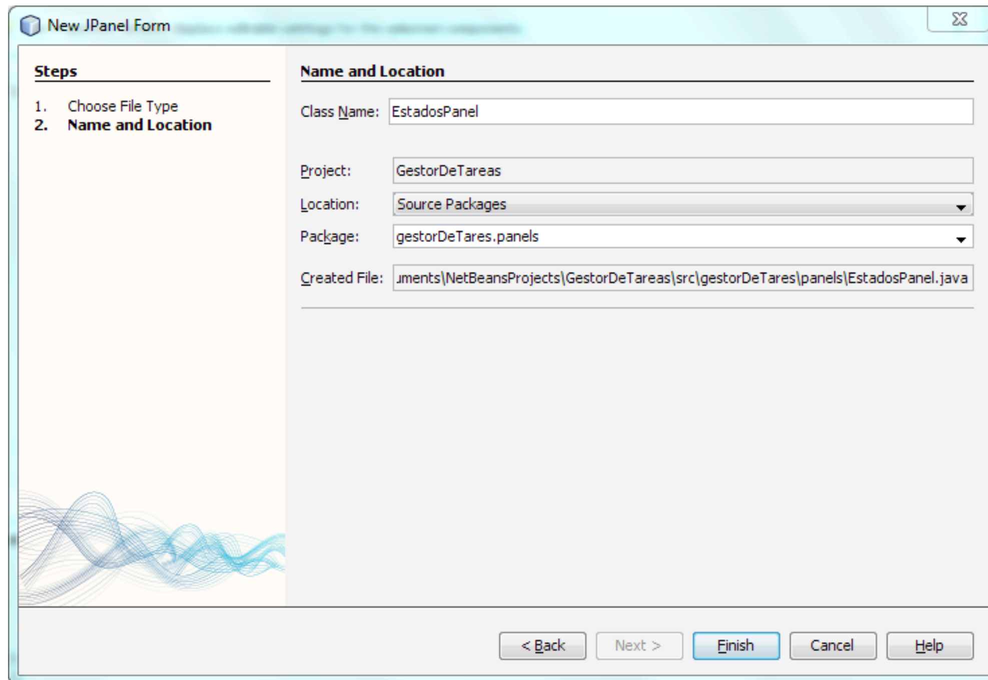
También podemos editar el fichero GestorDeTareasView.properties para poner en español los menús File y Help.

En este punto podemos ejecutar nuestra aplicación pulsando con el botón derecho sobre el proyecto y seleccionando Run, con lo que veremos nuestra ventana, y podremos acceder al diálogo de información.

Creando una pantalla de edición de datos

Vamos a empezar a añadir funcionalidad por el caso más simple, creando una pantalla que nos permita gestionar los datos de una tabla de la base de datos. Yendo al caso concreto de nuestra aplicación, crearemos una pantalla desde la que podremos gestionar los estados por los que podrían pasar las tareas.

Accedemos con el botón derecho del ratón sobre nuestro proyecto al menú, y seleccionamos la opción New – JPanel Form. En el cuadro de diálogo que aparece indicamos el nombre del nuevo panel, y el paquete en que lo queremos crear.

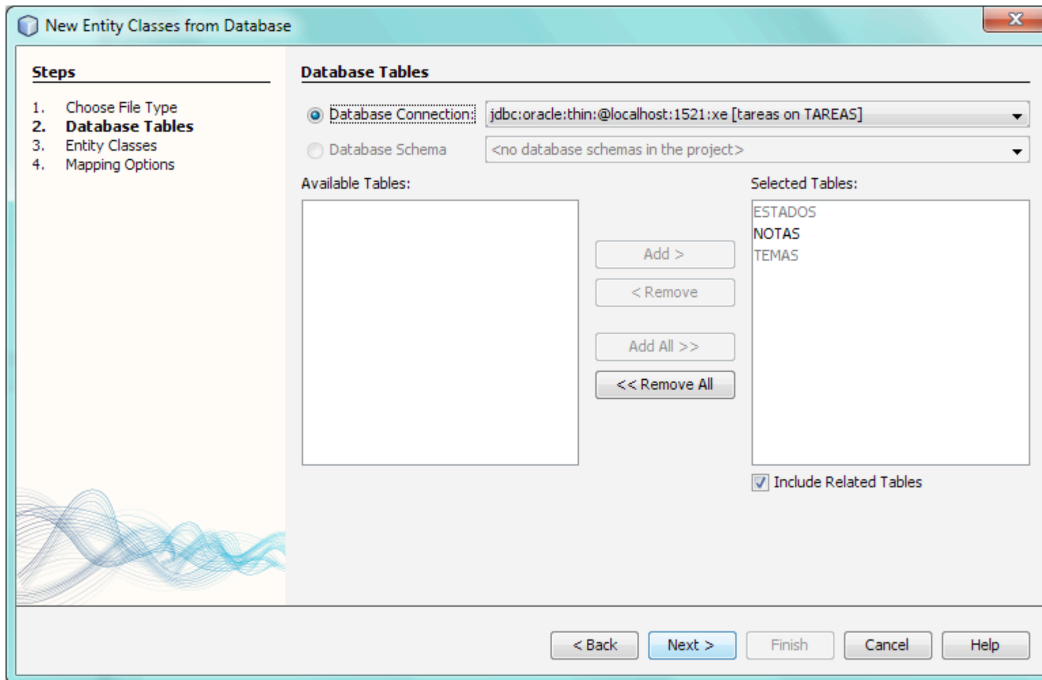


Nuevo panel 6

Ya tenemos un panel vacío para trabajar, pero antes de continuar con el desarrollo de la interfaz tenemos que preparar el acceso a datos que, como ya comentamos, se hará por medio de JPA.

Creando las entidades JPA

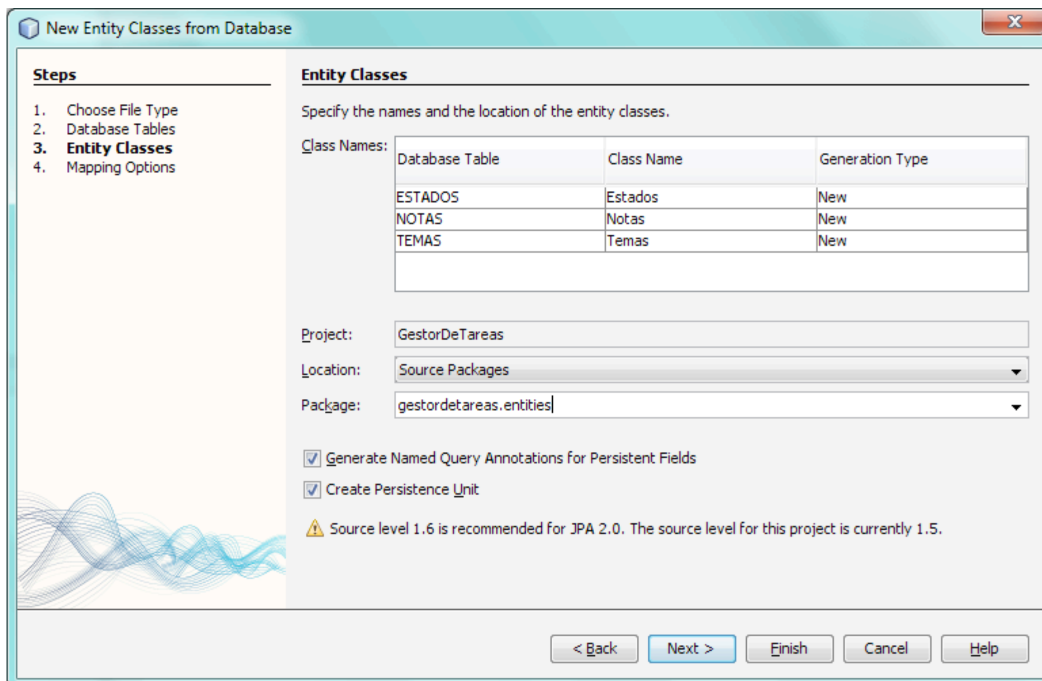
Volvemos a acceder sobre nuestro proyecto al menú New – Entity classes from Database, y en el diálogo seleccionamos la conexión a nuestra base de datos. Si no la tenemos, podemos crearla en este punto. Se nos mostrarán las tablas disponibles, y las incluiremos las tres en la selección.



Creación de entidades 7

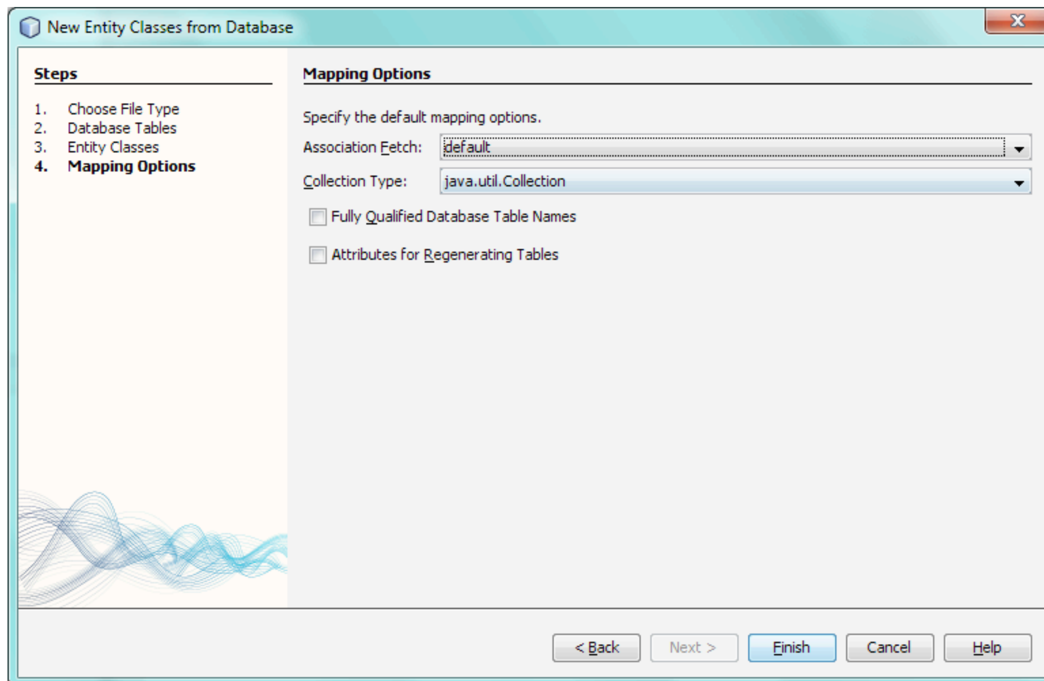
En el siguiente paso se nos preguntará qué nombre queremos dar a las clases, y la localización en que queremos crearlas (utilizaremos el paquete gestordetareas.entities).

En la sección inferior marcamos la opción para crear una unidad de persistencia.



Creación de entidades 8

Por último se nos pregunta por el modo en que queremos que se recupere la información de la base de datos, que dejaremos en default, y por el tipo de colecciones que queremos utilizar en nuestras entidades. Podemos elegir la que prefiramos.



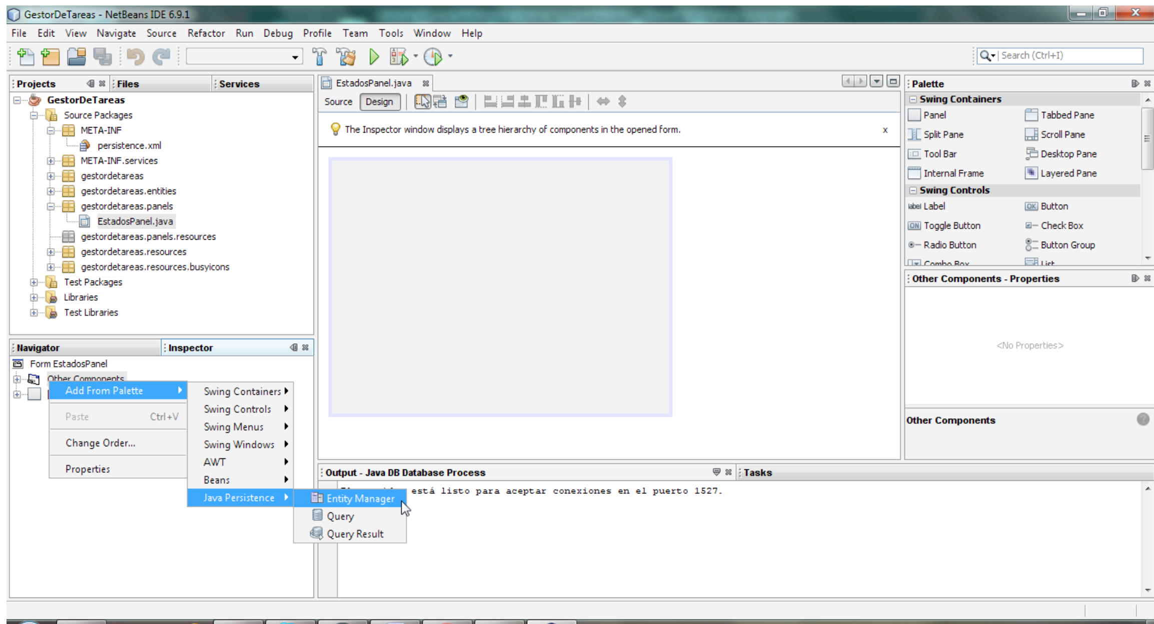
Creación de entidades 9

De esta forma hemos completado la creación de nuestras entidades y nuestra unidad de persistencia, con lo que tenemos lo que necesitamos para el acceso a datos. Si echamos un vistazo a las clases generadas, veremos que hemos obtenido una por cada tabla de la base de datos, y que cada una de ellas tiene atributos que se corresponden con las columnas de las tablas. Además, las clases contienen anotaciones JPA, mediante las que se indica la forma en que se van a persistir los datos. Durante el desarrollo iremos viendo algunos detalles más de las entidades.

Podemos volver al panel de administración de estados para continuar con su desarrollo.

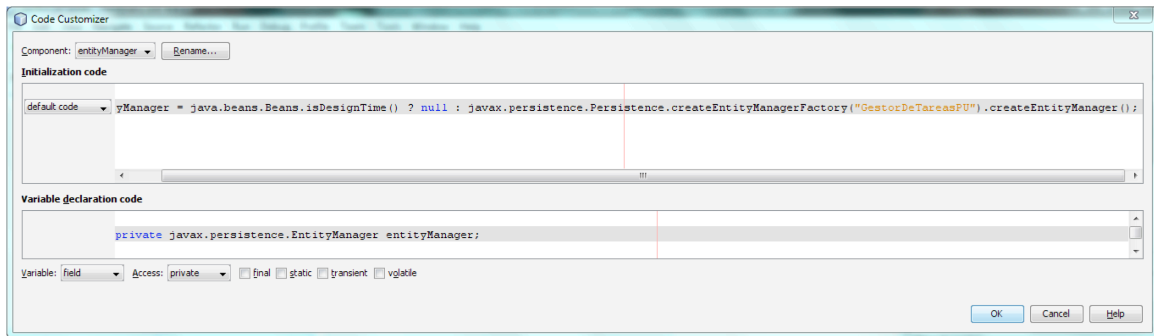
Accediendo a los datos

Lo primero que necesitamos tener en cada ventana para acceder a los datos es un gestor de entidades (entity manager). Lo creamos desde la vista diseño del panel, yendo a la paleta de inspección (Inspector), y pulsando con el botón derecho sobre Other Components – Add From Palette – Java Persistence – Entity Manager.



Nuevo entity manager 10

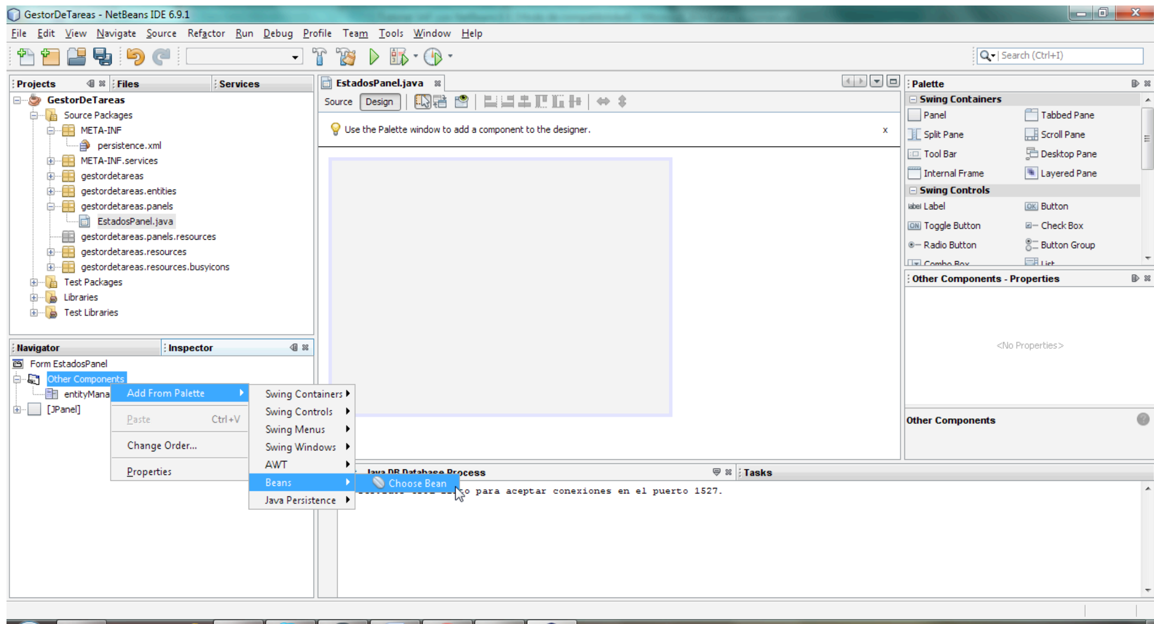
Se crea un nuevo componente, cuyo nombre cambiamos a entityManager. Podemos acceder ahora a la opción Customize Code, para ver cómo se construye este nuevo objeto.



Código del entity manager 11

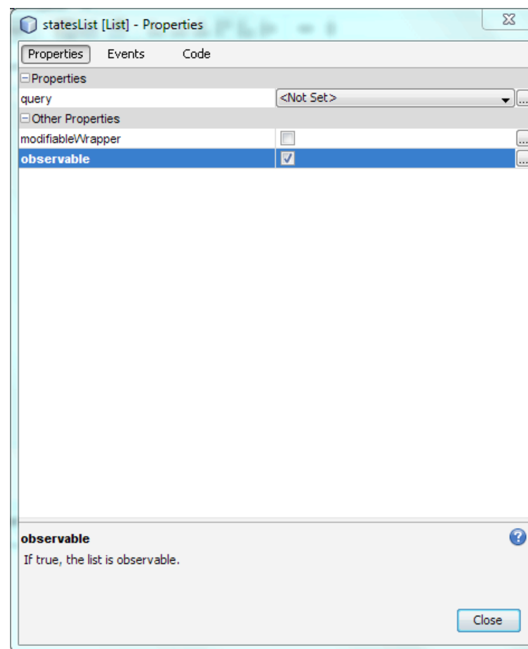
Como podemos ver, se crea una factoría pasándole el nombre que el IDE ha asignado a nuestra unidad de persistencia, GestorDeTareasPU.

Nuestra pantalla va a mostrar en primer lugar una lista con los estados almacenados en la base de datos, de modo que vamos a crear un atributo en la clase, de tipo List, que contendrá esta información. Lo hacemos accediendo a Other Components – Add From Palette – Bean – Choose Bean. En el diálogo introducimos java.util.List.



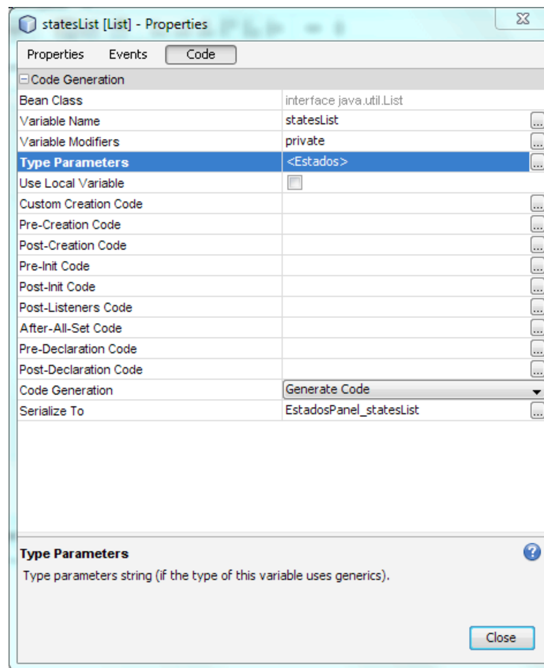
Nuevo bean 12

Cambiamos el nombre de la lista a statesList, y posteriormente accedemos a sus propiedades. En este nuevo diálogo podemos ver que tenemos la opción de hacer la lista observable. Esto implica que otros componentes podrán suscribirse como oyentes de la lista, de modo que cada vez que se agregue o elimine un elemento de la misma, serán notificados. De esta forma, los componentes de la interfaz que estén mostrando el contenido de la lista sabrán cuándo tienen que actualizarse.



Propiedad observable 13

Sabemos que la lista contendrá objetos de tipo Estados, una de las entidades que creamos anteriormente, de modo que también lo indicaremos accediendo a la pestaña Code del diálogo, e informando el campo Type Parameters con el valor <Estados>.



Propiedad type parameters 14

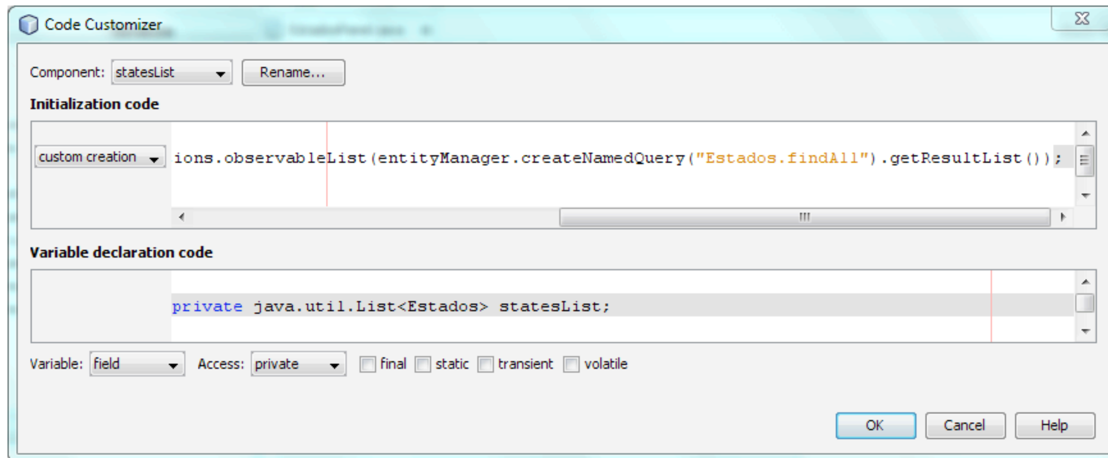
Una vez cambiada esta configuración, podemos acceder a la opción Customize Code de la lista, y ver que el IDE ha generado el código de creación por nosotros de la siguiente forma.

```
statesList = java.beans.Beans.isDesignTime() ? java.util.Collections.emptyList() :
org.jdesktop.observablecollections.ObservableCollections.observableList(((javax.persistence.Query)null).getResultList());
```

Código de creación de la lista 15

Podemos ver que la creación de la lista contiene una condición para determinar si estamos en tiempo de diseño, y utilizar una lista vacía en este caso. En teoría esto se hace para evitar problemas al trabajar con el editor de diseño, pero por limpieza del código yo suelo eliminarlo, y no he tenido problemas.

Al haber indicado que la lista es observable, se ha incluido una llamada a observableList. Sin embargo, todavía no hemos definido de qué forma se obtendrá la información. Podemos hacerlo creando un objeto de tipo Query, e indicándole a la lista que lo utilice, pero lo que haremos será incluir nosotros directamente el código para que se le pida al entityManager que cree una NamedQuery con la que obtener todos los estados de la base de datos. Tenemos que seleccionar Custom Creation en el desplegable de la izquierda para que el IDE nos permita modificar el código.



Código personalizado 16

Si compilamos el proyecto en este punto, obtendremos un error porque no se reconoce la clase Estados. Lo solucionamos accediendo al código del panel, e importándola. Además, podemos encontrarnos otro error en la línea en la que inicializamos la lista de estados, debido a que el paquete org.jdesktop.observablecollections no se encuentre en el classpath. Este problema lo solucionamos incluyendo la librería beansbindings a nuestro proyecto, accediendo a las propiedades, y dentro de la sección librería a Add Library – Import. Una vez hecho esto tenemos el proyecto compilando.

Las NamedQueries que tenemos disponibles en el proyecto son las que el IDE ha creado de forma automática en cada entidad, que podemos consultar en las anotaciones presentes antes de la cabecera de cada una de las clases.

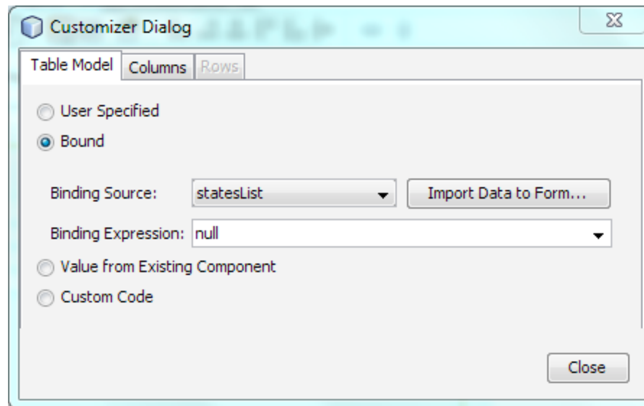
```
@Entity
@Table(name = "ESTADOS")
@NamedQueries({
    @NamedQuery(name = "Estados.findAll", query = "SELECT e FROM Estados e"),
    @NamedQuery(name = "Estados.findBySid", query = "SELECT e FROM Estados e WHERE e.sid = :sid"),
    @NamedQuery(name = "Estados.findByEstado", query = "SELECT e FROM Estados e WHERE e.estado = :estado")})
```

Named Queries 17

Si fuese necesario podríamos añadir nuevas consultas, pero en este tutorial no lo haremos.

Vamos a comenzar a añadir componentes a nuestra interfaz. En primer lugar vamos a mostrar una tabla con los estados almacenados en la base de datos. Nos vamos a la vista diseño de la clase EstadosPanel, y desde la paleta arrastramos un JTable. Le cambiamos el nombre a statesTable, y accedemos a la opción Table Contents, con el botón derecho sobre ella.

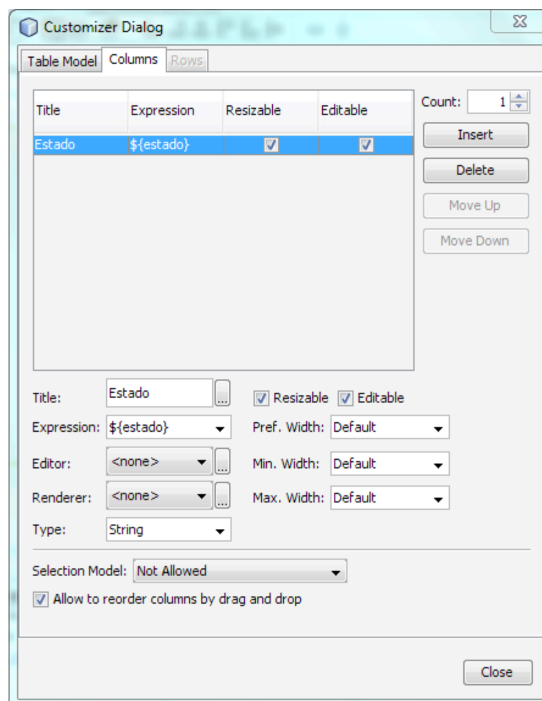
En primer lugar tenemos que enlazar la tabla con la lista de estados que hemos definido, seleccionando Bound, y statesList como Binding Source, tal como se ve en la siguiente imagen.



Enlace de datos 18

A continuación accedemos a la siguiente pestaña para definir las columnas que tendrá la tabla. Insertaremos una única columna que mostrará la descripción del estado. El campo expression podemos rellenarlo seleccionando directamente de la lista desplegable, que nos mostrará el contenido de los objetos de tipo Estados. Si la lista está vacía lo más probable es que se deba a que nos hemos saltado el paso anterior en el que indicábamos la propiedad Type Parameters para el objeto statesList.

También tenemos la posibilidad de definir otros aspectos como qué tipo de selección se permite en la lista, o si los campos serán editable. En este caso usamos todos los valores por defecto.



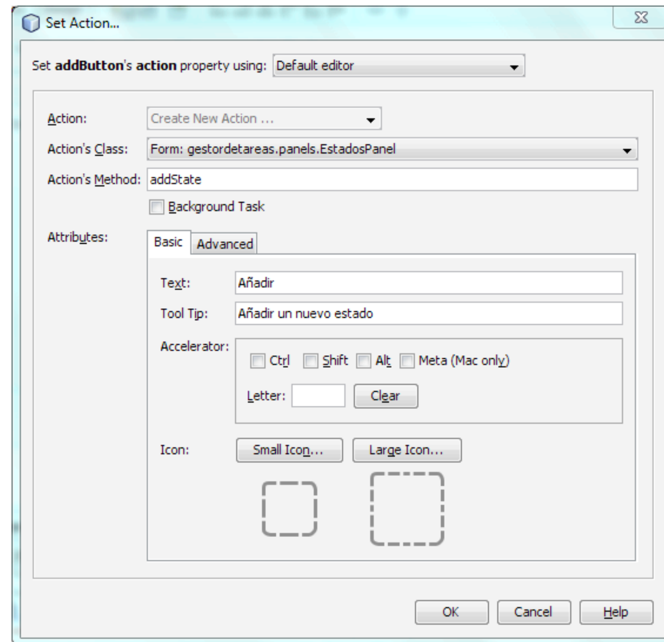
Definición de columnas 19

A continuación vamos a incluir tres botones a la interfaz. Dos de ellos nos servirán para añadir y eliminar elementos a la lista, y otro para guardar los cambios. Los arrastramos de

la paleta, los situamos debajo de la lista, y los renombramos addButton, deleteButton y saveButton.

Una vez renombrados, vamos a asignarle una acción a cada uno de ellos. De esta forma definiremos directamente el texto, la tecla de acceso rápido, el método a invocar al pulsarlo, etc.

Para asignar una acción a un botón basta con hacer doble clic sobre él, y aparece el diálogo correspondiente. Para el botón añadir, lo rellenamos de la siguiente forma:



Nuevo action 20

Para poder escribir en el campo Action´s Method tenemos que seleccionar la opción Create New Action en el campo Action. Una vez pulsado el botón Ok, el IDE nos lleva al editor textual, habiendo creado un nuevo método etiquetado como @Action, en el que tenemos que meter el código a ejecutar cuando se pulse el botón.

Lo que queremos hacer es crear un nuevo estado, y añadirlo a la lista.

```
@Action
public void addState() {
    statesList.add(new Estados());
}
```

Código addState 21

En este punto cobra importancia el hecho de haber definido la lista de estados como observable. Gracias a ello, como ya se ha comentado, al añadir un elemento a la lista, ésta notifica a sus oyentes de que ha sido modificada. La tabla, al haber sido enlazada con la

lista, se ha agregado como oyente y, por tanto, recibirá la notificación de que hay un nuevo estado, y se refrescará para mostrarlo.

Si al ejecutar la aplicación y pulsar este botón la tabla no se ve refrescada, lo más probable es que sea porque la lista no es observable. Hay que tener en cuenta que al marcar esta opción en las propiedades del objeto, lo que hace el IDE es encapsular la creación del objeto con una llamada al método observableList; por tanto, debemos tener cuidado de no eliminarla al personalizar el código.

Continuamos con los siguientes botones, cuya configuración se puede ver a continuación.

```
@Action (enabledProperty="stateSelected")
public void deleteState() {
    int n = JOptionPane.showConfirmDialog(null, "¿Está seguro de que desea eliminar el estado?",
            "Aviso", JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null);
    if (n == JOptionPane.YES_OPTION){
        int index = statesTable.convertRowIndexToModel(statesTable.getSelectedRow());
        Estados estado = statesList.get(index);
        statesList.remove(index);
        entityManager.remove(estado);
    }
}
```

Código deleteState 22

```
@Action
public void saveStates() {
    entityManager.getTransaction().begin();
    Iterator <Estados> iterator = statesList.iterator();
    while (iterator.hasNext()){
        entityManager.merge(iterator.next());
    }
    entityManager.getTransaction().commit();
}
```

Código saveStates 23

Como puede observarse, se ha añadido confirmación por parte del usuario para la eliminación de estados. En cualquier caso, los cambios efectuados no serán persistidos en la base de datos, a menos que el usuario pulse el botón guardar.

Para completar la pantalla, vamos a incluir un control que impida pulsar el botón eliminar si no hay un elemento seleccionado en la tabla. Para ello, las acciones tienen una propiedad enabledProperty. Con ella, el botón se hará oyente de la propiedad especificada, y se habilitará o deshabilitará convenientemente cada vez que ésta cambie.

Creamos un método isStateSelected de la siguiente forma.

```
public boolean isStateSelected(){
    return statesTable.getSelectedRow() != -1;
}
```

Código isStateSelected 24

Y añadimos un selectionListener a la tabla, de modo que cada vez que la selección se cambie, se notifique a los oyentes del cambio de esta propiedad. Incluimos este código en el constructor de la clase, después de la llamada al método initComponents.

```
/** Creates new form StatesPanel */
public StatesPanel() {
    initComponents();

    statesTable.getSelectionModel().addListSelectionListener(
            new ListSelectionListener() {

                public void valueChanged(ListSelectionEvent e) {
                    firePropertyChange("stateSelected", !isStateSelected(), isStateSelected());
                }
            });
}
```

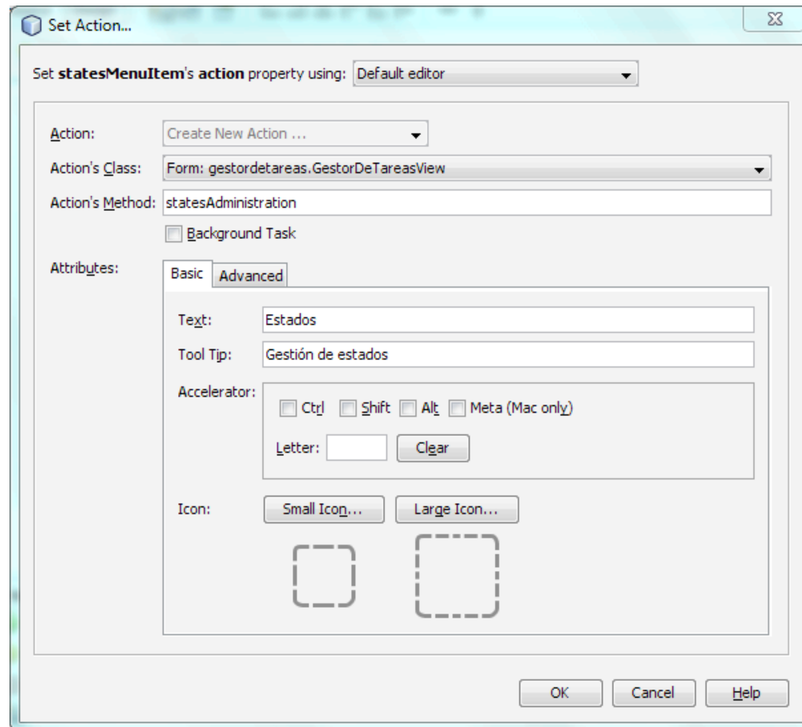
Código listSelectionListener 25

```
@Action (enabledProperty="stateSelected")
public void deleteState() {
    int n = JOptionPane.showConfirmDialog(null, "Are you sure?", "Warning",
            JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null);
    if (n == JOptionPane.YES_OPTION) {
        int index = statesTable.convertRowIndexToModel(statesTable.getSelectedRow());
        Estados estado = statesList.get(index);
        statesList.remove(index);
        entityManager.remove(estado);
    }
}
```

Código deleteState 26

Por último, añadimos a la anotación @Action del método para eliminar estados esta propiedad.

Hemos terminado el desarrollo de este panel. Ahora vamos a incluir una opción de menú en nuestra aplicación para poder acceder a él. Abrimos la clase GestorDeTareasView en modo diseño, y sobre la barra de menús elegimos la opción Add Menu. Cambiamos el nombre y texto del nuevo elemento a administrationMenu y Administración, respectivamente. Dentro del nuevo menú añadimos a su vez un elemento con Add from Palette – Menu Item. Le cambiamos el nombre a statesMenuItem, y posteriormente le asignamos una acción con la opción Set Action. Volvemos a tener el mismo cuadro de diálogo de siempre, y creamos un nuevo método, statesAdministration.



Nuevo action 27

Con este código hacemos que la ventana de nuestra aplicación muestre el panel que acabamos de crear.

```
@Action
public void statesAdministration() {
    this.setComponent(new EstadosPanel());
    this.getFrame().pack();
}
```

Código statesAdministration 28

Antes de poder ejecutar nuestra aplicación, nos queda un último paso. La base de datos se ha diseñado de forma que los registros tienen que tener un identificador, que se debe obtener de una secuencia Oracle. Por tanto, tenemos que añadir a nuestras entidades anotaciones para indicarles que esto debe ser así. Sobre la definición de los atributos sid, que son las claves primarias, añadimos el siguiente código.

```
@Id
@Basic(optional = false)
@Column(name = "SID")
@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ_ESTADOS")
@SequenceGenerator(name="seqEstados",sequenceName="SEQ_ESTADOS", allocationSize=1, initialValue=1)
private Long sid;
```

Anotaciones secuencia 29

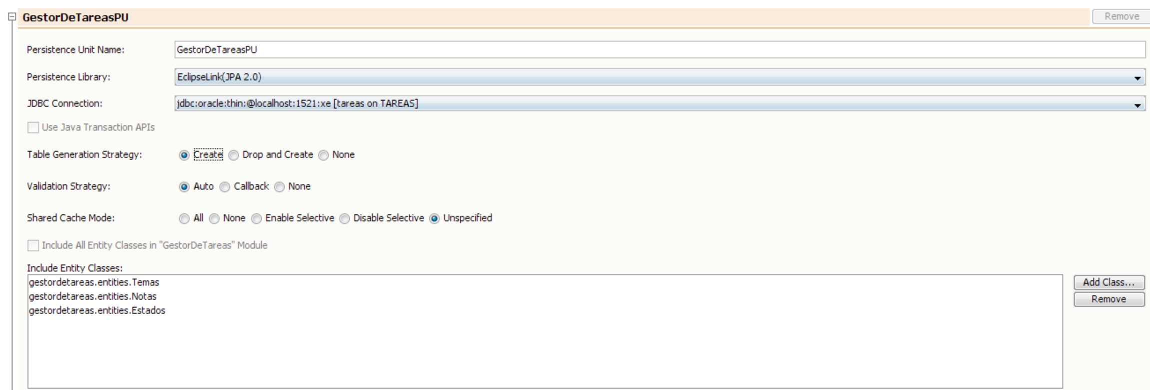
Esto habría que repetirlo para las tres entidades, cambiando el nombre de la secuencia (atributos generator, name y sequenceName).

A la hora de trabajar con este proyecto, hemos seguido un enfoque en el que primero hemos diseñado la base de datos, y a partir de ahí hemos generado las entidades y el resto del código. Sin embargo, también existe la posibilidad de hacerlo al contrario: podemos generar los elementos de base de datos a partir de las anotaciones de nuestras entidades. Vamos a ver este funcionamiento con la creación de las secuencias, que todavía no existen en la base de datos.

Si recordamos, cuando creamos las entidades a partir de las tablas de la base de datos, creamos también nuestra unidad de persistencia. En ese momento, en nuestro proyecto se creó un nuevo fichero, denominado persistence.xml, en el que se guarda la información relativa a la conexión a la base de datos, y las entidades involucradas en nuestro proyecto.

Este fichero se ubica dentro de la carpeta META-INF, y podemos verlo tanto en modo textual como gráfico con NetBeans. Además de la conexión a la base de datos, la librería de persistencia, o las entidades, en este fichero se puede definir la estrategia de generación de tablas. Esto es, podemos decirle a JPA si queremos que genere en la base de datos los elementos necesarios.

Actualmente el valor es None, pero vamos a cambiarlo a Create, para que al ejecutar nuestro proyecto se creen las secuencias que hemos definido en las entidades.



Configuración de la unidad de persistencia 30

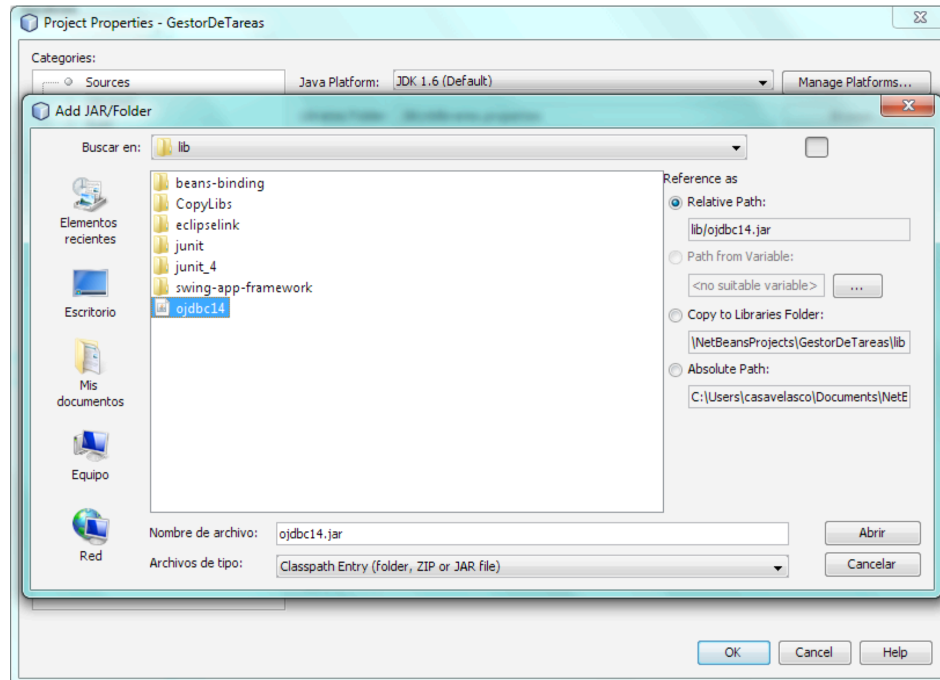
Ahora sí, podemos lanzar el programa, y ver los resultados. Cabe la posibilidad de que algunos de los textos que hemos definido en las acciones no se vean reflejados en la interfaz. Si es así, tendremos que ejecutar un Clean and Build sobre el proyecto antes de ejecutarlo. Si aún así siguen sin verse los textos correctos, lo más fácil es ir a los componentes con problemas y corregirlos utilizando la opción Edit Text, aunque lo más limpio sería ir a los ficheros de propiedades, y eliminar las entradas sobrantes.

Incluyendo el Driver de Oracle

Al ejecutar la aplicación accediendo a datos por primera vez, se producirá el siguiente error:

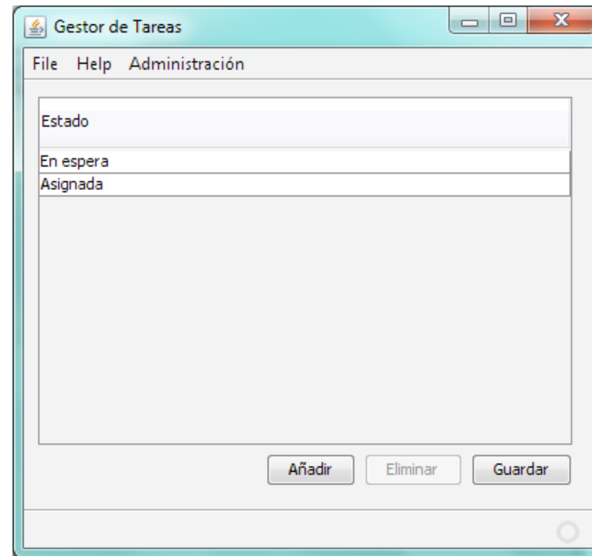
Configuration error. Class [oracle.jdbc.OracleDriver] not found.

Lógicamente se debe a que no hemos añadido el driver de Oracle a nuestro proyecto. En este caso accedemos a la sección librerías, dentro de las propiedades, y le indicamos dónde se encuentra dicho driver, que no viene con la distribución de NetBeans, por lo que tendremos que ponerlo nosotros mismos en la carpeta lib de nuestro proyecto.
Se puede encontrar por ejemplo en las instalaciones de Oracle, jDeveloper o SQLDeveloper, o bien buscarlo en Internet.



Driver de Oracle 31

El resultado al ejecutar es el siguiente.



Aplicación 32

Añadiendo barras de progreso

Todavía podemos mejorar un poco la presentación de nuestra aplicación, haciendo uso de la barra de estado que NetBeans añadió cuando creamos el proyecto para mostrar el progreso de la acción de guardado, que es la que puede ocasionar mayor demora.

Para ello tenemos que crear una clase que extienda de org.jdesktop.application.Task, en la que implementaremos los métodos doInBackground y succeeded. En el primero implementaremos la tarea, y en el último podemos incluir algún tipo de postprocesamiento.

La nueva clase la crearemos como una clase interna en el panel de estados, de la siguiente forma.

```
private class SaveStatesTask extends org.jdesktop.application.Task<Object, Void> {
    SaveStatesTask(org.jdesktop.application.Application app){
        super(app);
    }

    @Override protected Object doInBackground(){
        setMessage("Guardando los datos...");
        entityManager.getTransaction().begin();
        Iterator<Estados> iterator = statesList.iterator();
        while (iterator.hasNext()){
            entityManager.merge(iterator.next());
        }
        entityManager.getTransaction().commit();
        return null;
    }

    @Override protected void succeeded(Object result){
        setMessage("Datos guardados correctamente");
    }
}
```

Código SaveStatesTask 33

En el método doInBackground llevamos a cabo el guardado de la información, y hacemos uso del método setMessage para informar a través de la barra de estado de la tarea que se está realizando en cada momento.

Para hacer que esta tarea se ejecute al pulsar el botón de guardado, modificamos el action saveStates para que devuelva una instancia de esta clase.

```
@Action
public Task saveStates() {
    return new SaveStatesTask(org.jdesktop.application.Application.getInstance(gestordetareas.GestorDeTareasApp.class));
}
```

Código saveStates 34

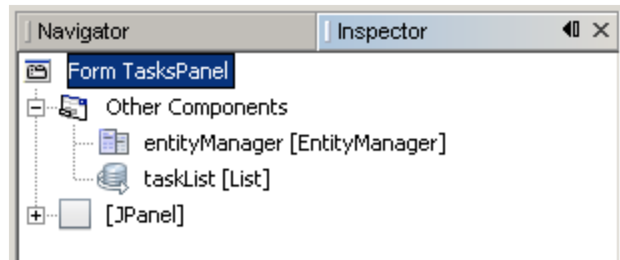
De esta forma podemos dar por terminada la ventana para gestionar los estados de las tareas. Ahora vamos a pasar a un caso algo más complejo, en el que haremos una interfaz desde la que podremos gestionar las tares y asociarle anotaciones.

Creando una vista maestro-detalle

Creamos un nuevo JPanel en el paquete paneles, y lo nombramos TareasPanel. Igual que hicimos anteriormente, tenemos que crear un entityManager, que nos servirá para realizar las peticiones a la base de datos.

La primera parte de la ventana es muy similar a la anterior, ya que volveremos a crear una tabla para mostrar un listado de objetos, y controles para añadir, eliminar, etc., con la diferencia de que en este caso trabajaremos con Temas en lugar de Estados.

Creamos una lista desde la vista de inspección, la marcamos como observable, e indicamos que contendrá objetos de tipo Temas. Accedemos a la edición textual, e importamos la clase Temas para evitar errores de compilación.



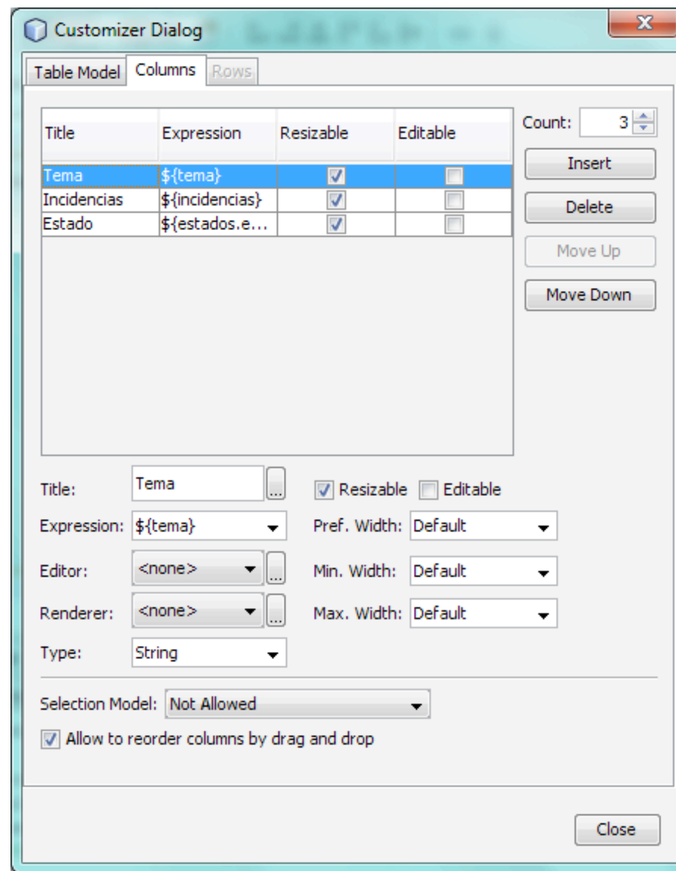
Inspección del TasksPanel 35

Seguimos personalizando el código de inicialización de la lista, que quedaría como sigue.

```
org.jdesktop.observablecollections.ObservableCollections.observableList(entityManager.createNamedQuery("Temas.findAll").getResultList());
```

Código de creación de la lista de temas 36

Y arrastramos un JTable desde la paleta, al que nombraremos taskTable, y cuyo contenido enlazamos con la lista taskList. Por último, añadiremos las columnas que queremos mostrar.



Columnas 37

En este caso nos vamos a permitir que el usuario modifique los datos directamente en la tabla, sino que vamos a crear un cuadro de diálogo para introducir la información, por lo que hemos desmarcado los checks Editable de todas las columnas.

Lo siguiente que haremos será crear botones para añadir, modificar y eliminar tareas, y asociarles sus respectivas acciones, todavía sin código. Además podemos ir definiendo la propiedad taskSelected, que nos indicará si hay un registro seleccionado en la tabla de tareas, y nos servirá para definir si los botones modificar y eliminar estarán activados.

Recordamos que los pasos son: crear el oyente en la lista, definir un método isTaskSelected, y poner la propiedad enabledProperty a las acciones.

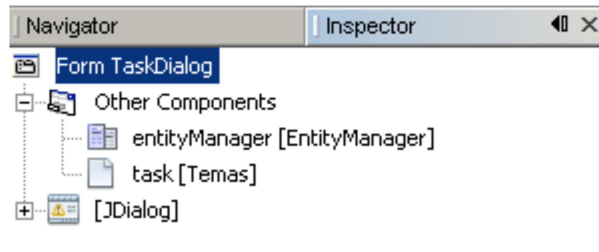
Creando un diálogo para editar datos

Vamos a continuar creando el diálogo para la edición de tareas. Seleccionamos New - JDialog Form en el menú, lo nombramos como TemasDialog, y lo ubicamos en el paquete gestordetareas.paneles.dialogos.

El IDE crea el diálogo con un método main, que podemos eliminar en el editor textual, ya que no lo necesitamos.

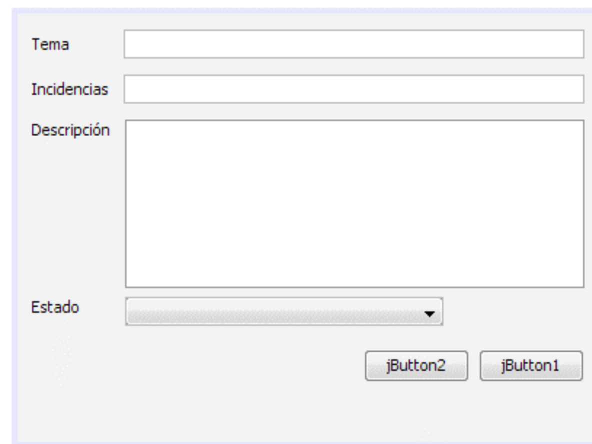
Lo que vamos a tener ahora es una interfaz que mostrará la información de una instancia de la clase Temas en distintos tipos de componentes Swing. Por tanto, vamos a definir en la clase un atributo de tipo Temas, que será la instancia que se está creando o editando, y vamos a enlazar los distintos componentes de la interfaz con sus propiedades.

Como siempre, lo primero que tenemos que hacer es crear un objeto entityManager, y posteriormente creamos el objeto task, de tipo Temas. Tenemos que usar la misma opción del menú que para crear las listas (añadir un Bean, y escribir el nombre completo de la clase).



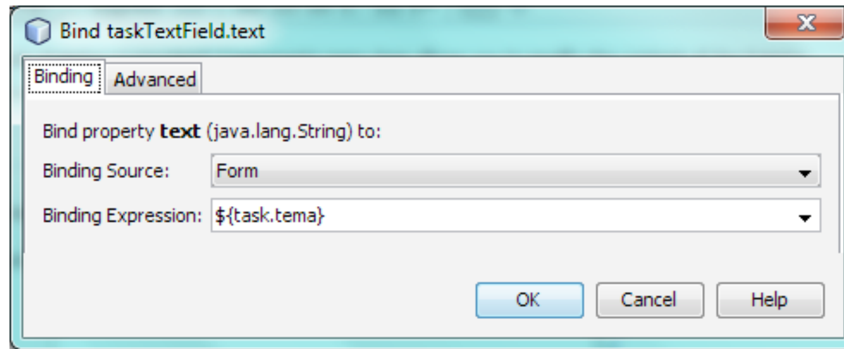
Inspección de TaskDialog 38

Ahora vamos a arrastrar todos los componentes que queremos presentar. Las tareas tienen el tema, las incidencias, la descripción y el estado. Para editarlos usaremos dos TextField, un TextArea, y un ComboBox, respectivamente. Además incluimos un botón para guardar y otro para cancelar, con sus respectivas acciones. La interfaz quedaría como sigue.



Diseño de TaskDialog 39

El siguiente paso es enlazar el valor mostrado en los campos de texto con las propiedades de la tarea. Para ellos hacemos clic derecho sobre cada uno de ellos, y accedemos a Bind – text. Como Binding Source elegimos en todos los casos el valor Form, y como Binding Expression la propiedad del objeto task que corresponda en cada caso.



Enlace de datos 40

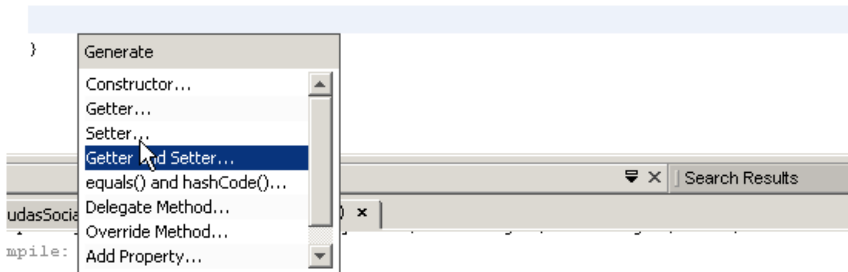
En el caso del ComboBox, tenemos que enlazar dos propiedades: los elementos seleccionables, y el seleccionado. Para poder mostrar el primero, vamos a crear una lista en la clase, statesList, igual que las veces anteriores, con la diferencia de que en este caso no es necesario que sea observable. Personalizaremos el código de creación para que contenga todos los estados guardados en la base de datos.

Una vez que la tenemos, enlazamos la propiedad elements del ComboBox con la nueva lista. En el campo Binding Expression no es necesario poner nada.

La propiedad selectedElements podemos enlazarla exactamente igual que los textos, con la diferencia de que en Binding Expression elegimos el objeto estados, que está dentro de task y aparece representado como una carpeta por no ser un objeto básico.

Vamos a crear otra propiedad en el diálogo, de tipo boolean, que servirá para indicar a otras pantallas si las modificaciones se han confirmado o no. Esto es, de si se ha pulsado el botón guardar o cancelar.

En este caso accedemos directamente al código de la clase, y añadimos un atributo al final de las declaraciones. Le creamos también un get y un set, mediante el diálogo Insert Code, accesible con el botón derecho.



Creación de getter y setter 41

También vamos a añadir un get y un set para la propiedad task, de forma que antes de abrir el diálogo podamos indicar con qué tarea tiene que trabajar, para cuando estemos modificando.

Ahora añadimos el siguiente código para los métodos invocados por los botones, teniendo en cuenta que ya sabemos cómo hacer que se invoquen en una tarea, si nos interesa.

```
@Action
public void save() {
    entityManager.merge(task);

    this.setEntityConfirmed(true);
    this.setVisible(false);
}

@Action
public void cancel() {
    this.setEntityConfirmed(false);
    this.setVisible(false);
}
```

Código de sabe y cancel 42

Por último, vamos a señalar que el diálogo creado es modal, lo que podemos hacer pinchando sobre el nodo JDialog en la vista de inspección, y accediendo a la vista de propiedades. Simplemente marcamos esta opción.

Añadiendo disparadores para cambios de propiedades

Otro punto importante para que la aplicación funcione correctamente, es que las entidades gestionadas deben notificar los cambios que se producen en sus propiedades.

Al crear el diálogo para modificar datos, hemos enlazado los componentes de la interfaz con las propiedades de un objeto. Por ejemplo, en el diálogo TemasDialog, el campo de texto para las incidencias se enlaza con la propiedad incidencias del objeto task, de tipo Temas. Sin embargo, con la actual clase Temas, cuando el valor de la propiedad incidencias se modifique, la interfaz no se verá afectada. Para que el cambio se refleje en el área de texto es necesario que la clase lance un aviso de propiedad cambiada.

Esto lo conseguimos añadiendo un atributo de tipo PropertyChangeSupport (que marcaremos como transient para indicar a JPA que no debe ser persistido), y modificando los métodos set para que se informe del cambio a los oyentes registrados.

```
@Transient
private PropertyChangeSupport changeSupport = new PropertyChangeSupport(this);
```

Definición del PropertyChangeSupport 43

```java
public void setNota(String nota) {
    String oldNota = this.nota;
    this.nota = nota;
    changeSupport.firePropertyChange("nota", oldNota, nota);
}
```

Código de setNota 44

También será necesario, por tanto, añadir métodos para que los oyentes puedan registrarse.

```java
public void addPropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.removePropertyChangeListener(listener);
}
```

Código de propertyChangeListeners 45

Utilizando el diálogo

Una vez completado el diálogo, volvemos al TareasPanel, y completamos el código de los métodos que dejamos pendientes.

El método añadir mostrará el diálogo, y comprobará si los datos se han confirmado o cancelado. En caso de confirmación añadirá la nueva tarea a su lista, la grabará en la base de datos, y modificará la selección de la tabla para forzar su repintado.

```java
@Action
public void addTask() {
    JFrame mainFrame = GestorDeTareasApp.getApplication().getMainFrame();
    TemasDialog td = new TemasDialog(mainFrame, true);
    td.setLocationRelativeTo(mainFrame);
    td.setTask(new Temas());
    td.setVisible(true);
    if (td.isEntityConfirmed()){
        taskList.add(td.getTask());
        entityManager.persist(td.getTask());
        entityManager.getTransaction().begin();
        entityManager.getTransaction().commit();
    }
    int row = taskTable.getRowCount()-1;
    if (row >= 0){
        taskTable.clearSelection();
        taskTable.setRowSelectionInterval(row, row);
        taskTable.scrollRectToVisible(taskTable.getCellRect(row, 0, true));
    }
}
```

Código de addTask 46

El método modificar funciona de forma muy similar, pero antes de hacer visible el diálogo llamará al método setTask para indicarle que se va a editar la tarea seleccionada en la tabla.

```
@Action (enabledProperty="taskSelected")
public void modifyTask() {
    int selected = taskTable.getSelectedRow();
    JFrame mainFrame = GestorDeTareasApp.getApplication().getMainFrame();
    TemasDialog td = new TemasDialog(mainFrame, true);
    td.setLocationRelativeTo(mainFrame);
    td.setTask(taskList.get(taskTable.convertRowIndexToModel(selected)));
    td.setVisible(true);
    if (td.isEntityConfirmed()){
        entityManager.merge(td.getTask());
        entityManager.getTransaction().begin();
        entityManager.getTransaction().commit();
        taskList.set(taskTable.convertRowIndexToModel(selected), td.getTask());
    }
    else{
        entityManager.refresh(td.getTask());
    }

}
```

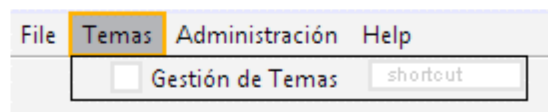
Código de modifyTask 47

El método eliminar pedirá confirmación, y borrará el elemento de la tabla, eliminando la selección posteriormente.

```
@Action (enabledProperty="taskSelected")
public void deleteTask() {
    int selected = taskTable.getSelectedRow();
    int n = JOptionPane.showConfirmDialog(null, "¿Está seguro de que desea eliminar el tema?", "Aviso",
            JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null);
    if (n == JOptionPane.YES_OPTION){
        Temas toRemove = taskList.get(taskTable.convertRowIndexToModel(selected));
        entityManager.remove(toRemove);
        entityManager.getTransaction().begin();
        entityManager.getTransaction().commit();
        taskList.remove(toRemove);
        taskTable.clearSelection();
    }
}
```

Código de deleteTask 48

Para poder probar lo que llevamos desarrollado hasta el momento, tenemos que dar acceso a la nueva pantalla a través del menú. Abrimos la clase GestorDeTareasView, y añadimos un nuevo menú como se ve en la imagen. El Action es igual que el que creamos para la pantalla de administración, pero cambiando el panel a mostrar.

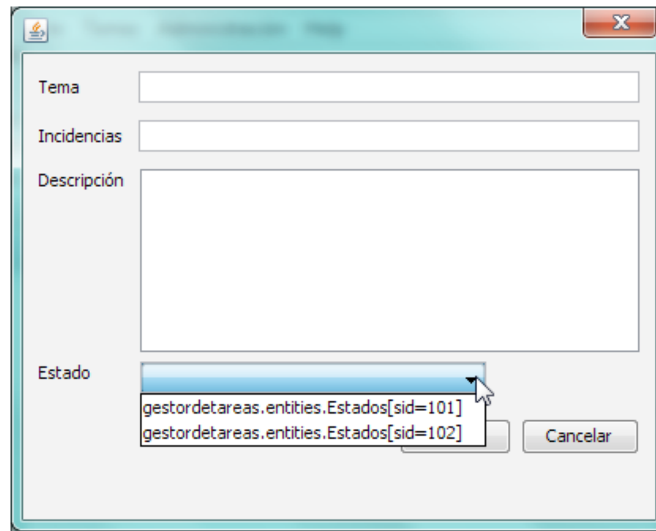


Menú 49

Al ejecutar podemos ver que el funcionamiento es correcto, pero hay algunos detalles que hemos dejado pendientes, y vamos a ir viendo a continuación.

Añadiendo un renderizador para los ComboBox

Cuando estamos creando una nueva tarea, el desplegable de estados muestra un número de valores adecuado (en la imagen de abajo dos, que son los estados almacenados en la base de datos), pero lo que vemos es el nombre de la clase y el sid del objeto. Lógicamente lo interesante sería ver el nombre del estado, para lo que tenemos que añadir un renderizador al ComboBox.



Aplicación 50

Vamos a crear una clase interna en TemasDialog, que extienda de DefaultListCellRenderer, y que sobrescriba el método getListCellRendererComponent, que es el que da el valor a mostrar cuando se dibuja el ComboBox. El código queda como sigue.

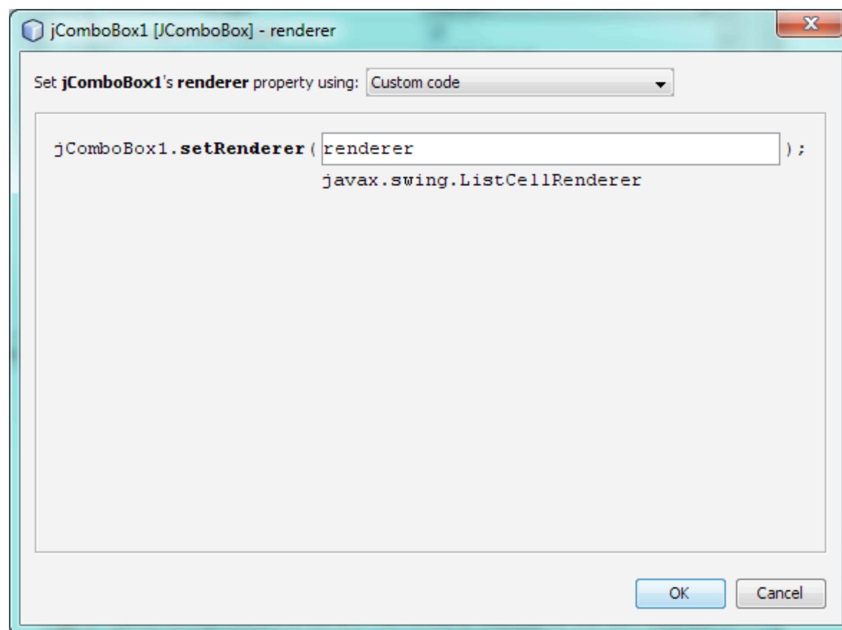
```
public class StateCellRendered extends DefaultListCellRenderer {

@Override
public Component getListCellRendererComponent(
        JList list, Object value, int index, boolean isSelected, boolean cellHasFocus) {
    super.getListCellRendererComponent(list, value, index, isSelected, cellHasFocus);
    if (value instanceof Estados) {
        setText(((Estados)value).getEstado());
    }
    return this;
}
```

Código de StateCellRenderer 51

Como se puede ver, se comprueba que el valor sea una instancia de la clase Estados, y en ese caso se pone en el texto el valor de la propiedad estado. En este caso podríamos ahorrarnos la comprobación del tipo de value, dado que nuestro ComboBox sólo tendrá

Estados, pero esta estructura puede ser útil cuando un ComboBox tenga elementos de distinto tipo o, algo más habitual, cuando vayamos a usar el mismo renderizador para distintos ComboBox.

Para que los valores se muestren correctamente, falta definir un atributo de tipo StateCellRenderer en la clase TemasDialog, e indicar al ComboBox que lo use. Desde el editor textual creamos el atributo, al que llamaremos renderer, y lo añadimos al ComboBox como se ve en la imagen (vamos a las propiedades del ComboBox, accedemos al diálogo de la propiedad renderer, elegimos Custom Code, y escribimos el nombre de nuestro renderizador.



Edición de la propiedad renderer 52

Añadiendo notificación al seleccionar temas

Otro error que podemos detectar, es que al tratar de modificar una tarea no se muestran los valores en el diálogo, y vamos a explicar el porqué.

Lo que hemos hecho ha sido enlazar el valor de los componentes de la interfaz (de sus propiedades text, selectedElement...) con el valor de un determinado atributo del objeto task. Si nos fijamos en el código del diálogo, en el método initComponents, este enlace se materializa mediante una llamada al método bind del objeto bindingGroup, al que se han ido añadiendo las definiciones de los enlaces de datos. Tal como está definido el código, en el momento en que se lleva a cabo el enlace (llamada a bind), el objeto task será siempre un objeto recién creado y vacío, dado que se inicia al comienzo del método initComponents con una llamada al constructor sin parámetros de la clase Temas. Por tanto, en el momento de enlazar los datos, las propiedades del objeto task están vacías.

Posteriormente, tras crear el diálogo desde la clase TareasPanel, hemos hecho una llamada al método setTask de TemasDialog, en el que cambiamos el objeto task, pero como el enlace de datos se hizo en el constructor, no vemos cambios en la interfaz.

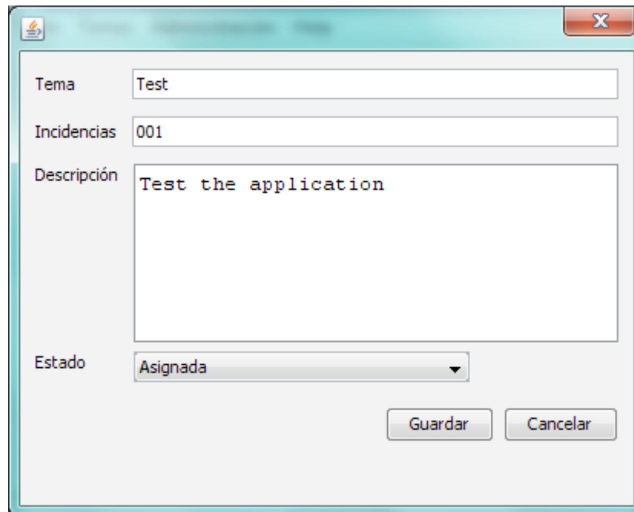
Al realizar el enlace entre el componente de la interfaz y una propiedad de un objeto, cada uno de los elementos enlazados se agrega como oyente del otro, pero somos nosotros los que tenemos que lanzar la notificación cada vez que se produzca un cambio. En este caso, tenemos que notificar que la propiedad task de la clase TemasDialog ha cambiado, para que los componentes de la interfaz sepan que tienen que actualizarse. Por tanto, agregamos a la clase TemasDialog un elemento de tipo PropertyChangeSupport, modificamos el método setTask para que notifique el cambio de la propiedad task, e incluimos métodos para agregarse y eliminarse como oyentes.

```
private PropertyChangeSupport propertyChangeSupport = new PropertyChangeSupport(this);

public Temas getTask() {
    return task;
}

public void setTask(Temas task) {
    Temas oldTask = this.task;
    this.task = task;
    propertyChangeSupport.firePropertyChange("task", oldTask, task);
}

public void addPropertyChangeListener(PropertyChangeListener listener) {
    propertyChangeSupport.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener(PropertyChangeListener listener) {
    propertyChangeSupport.removePropertyChangeListener(listener);
}
```

Código para el propertyChangeSupport 53

Queda un detalle importante. Para que esto funcione el objeto task no puede ser inicializado previamente, por lo que tenemos que modificar el código de creación mediante la opción Customize Code para que se le asigne un valor nulo.

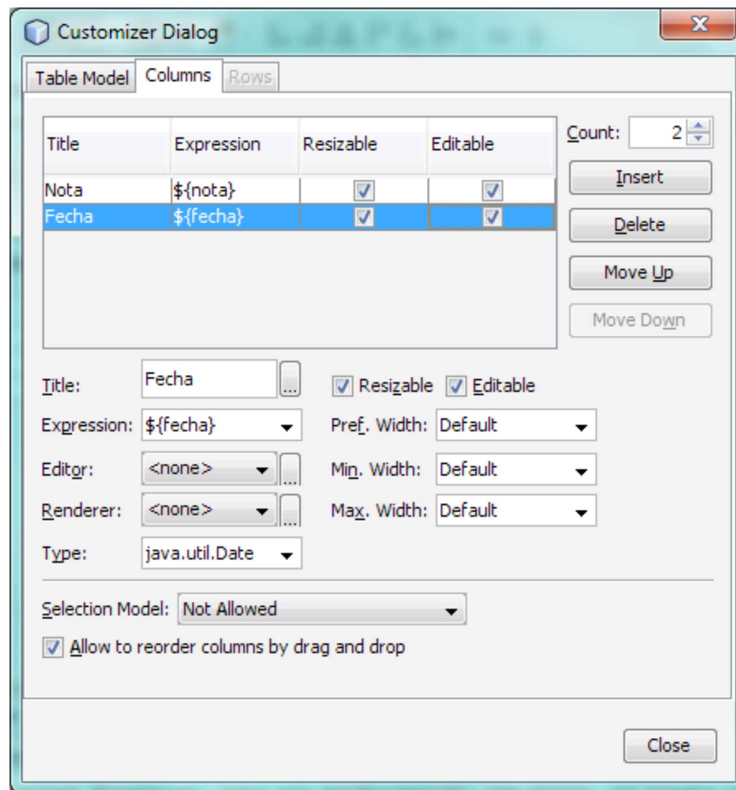
Si ejecutamos la aplicación ahora, podemos ver que se han solucionado los problemas presentados. También se aprecia que el diálogo no tiene título, por lo que se lo damos accediendo a sus propiedades, por ejemplo seleccionándolo en la ventana de inspección.



Aplicación 54

Continuando con el maestro-detalle

Vamos a continuar el desarrollo mostrando una segunda tabla en la ventana principal, en la que veremos las anotaciones asociadas a cada una de las tareas. Arrastramos el componente JTable desde la paleta, debajo de la tabla de tareas y sus botones. Sobre la marcha accedemos con el botón derecho a la opción Table Contents..., y enlazamos el contenido de la tabla con la taskTable como BindingSource, y selectedElement.notasCollection como Binding Expression. Cambiamos también el nombre de la tabla a notesTable, e indicamos las columnas que vamos a querer mostrar.



Columnas 55

Añadimos ahora tres botones para añadir, modificar y eliminar notas, y les asignamos sus respectivos nombres y acciones: addNote, modifyNote y deleteNote. La acción addNote sólo debería estar habilitada cuando hubiese una tarea seleccionada. Como ya creamos anteriormente una propiedad que indica si es así, añadimos este valor para la acción. En cuanto a las acciones para modificar y eliminar, se deberían poder ejecutar cuando haya un registro seleccionado en la tabla de notas. Creamos por tanto una nueva propiedad noteSelected con el siguiente código en el constructor de TareasPanel.

```
notesTable.getSelectionModel().addListSelectionListener(
        new ListSelectionListener() {

            public void valueChanged(ListSelectionEvent e) {
                firePropertyChange("noteSelected", !isNoteSelected(), isNoteSelected());
            }
        });
```

Código de listSelectionListener 56

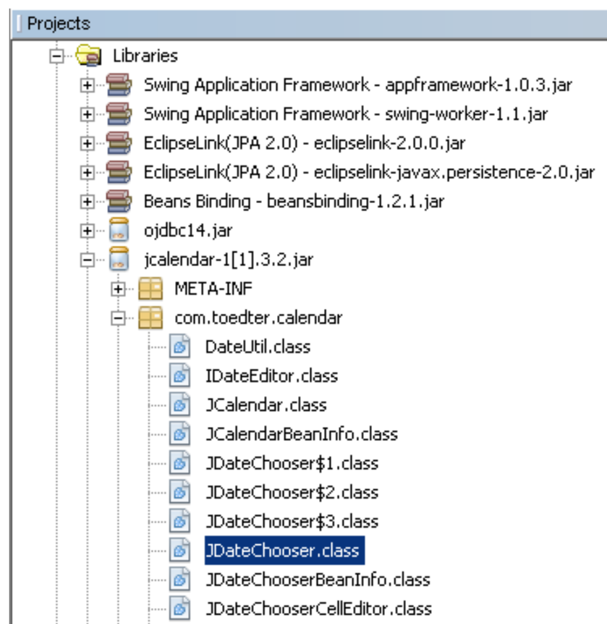
El método isNoteSelected devolverá cierto siempre que el índice seleccionado en la tabla notesTable sea distinto de -1.

Antes de añadir código para las acciones vamos a crear un nuevo diálogo para la edición de notas, de la misma forma que hicimos para las tareas. En el paquete dialogs, vamos a la opción del menú New – JdialogForm. Lo nombramos como NotasDialog.

Añadimos un entityManager y un objeto de tipo Notas a los componentes, asegurándonos de cambiar el código de construcción de éste último para que se le asigne un valor nulo.

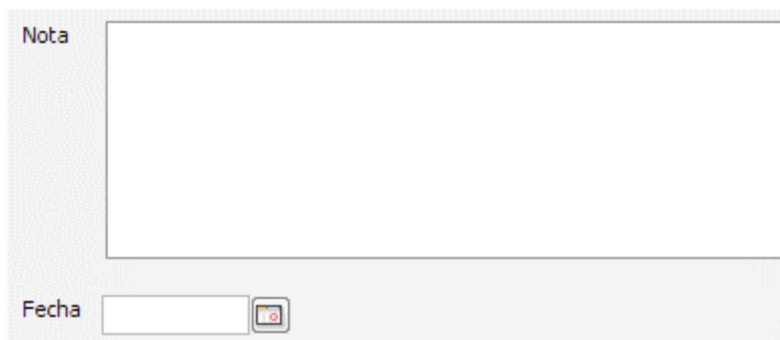
Vamos añadiendo los elementos de la interfaz, renombrándolos, y enlazándolos directamente con su correspondiente valor del objeto note, como hicimos en el diálogo para las tareas. En este caso tendremos un JTextArea para el texto de la nota, y un editor de fechas. Usaremos un JDateChooser, que es un fantástico componentes desarrollado por Kai Toedter y que podéis encontrar en su Web (http://www.toedter.com/).

Para poder utilizar este componente copiamos el jar en la carpeta lib del proyecto, y accedemos a sus propiedades. En la opción Libraries pulsamos el botón Add Jar/Folder, y elegimos el jar. Una vez aceptado, en el navegador del proyecto podemos desplegar la carpeta Libraries – jcalendar-1... - com.toedter.calendar, y ver la clase JDateChooser.



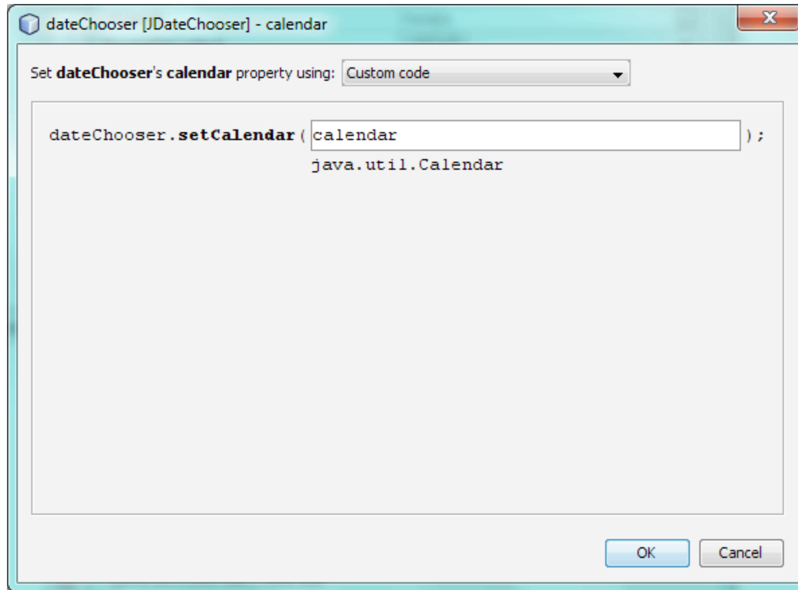
Navegador de librerías 57

Ya no tenemos más que arrastrar este elemento sobre nuestro diálogo, y situarlo donde queramos.



Diálogo 58

Renombramos el elemento como dateChooser. Vamos a crear un atributo de tipo Calendar en la clase NotasDialog, que será el que utilice el selector para controlar la fecha. Lo nombramos como calendar, y lo asignamos a la propiedad correspondiente del selector.



Propiedad calendar 59

Además, añadimos los botones para guardar y cancelar. El código de las acciones es equivalente al del diálogo de las tareas, añadiendo alguna línea para el control de la fecha, y tenemos que añadir también a la clase el método isNoteConfirmed, y el campo propertyChangeSupport para notificar los cambios en la anotación, además de hacer el diálogo modal.

```
private PropertyChangeSupport propertyChangeSupport = new PropertyChangeSupport(this);
private boolean noteConfirmed = false;
private Calendar calendar = Calendar.getInstance();

@Action
public void save() {
    note.setFecha(dateChooser.getDate());
    entityManager.merge(note);

    this.setNoteConfirmed(true);
    this.setVisible(false);
}

@Action
public void cancel() {
    this.setNoteConfirmed(false);
    this.setVisible(false);
}

public void addPropertyChangeListener(PropertyChangeListener listener) {
    propertyChangeSupport.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener(PropertyChangeListener listener) {
    propertyChangeSupport.removePropertyChangeListener(listener);
}

public boolean isNoteConfirmed(){
    return noteConfirmed;
}
```

Código de NotasDialog 60

```
public void setNoteConfirmed(boolean noteConfirmed){
    this.noteConfirmed = noteConfirmed;
}

public Notas getNote() {
    return note;
}

public void setNote(Notas note) {
    Notas oldNote = this.note;
    this.note = note;
    if (note.getFecha() != null){
        calendar.setTime(note.getFecha());
        dateChooser.setCalendar(calendar);
    }
    propertyChangeSupport.firePropertyChange("note", oldNote, this.note);
}
```

Código NotasDialog 61

Con estos cambios ya podemos volver a la ventana de gestión de tareas para escribir el código con el que se tratan las anotaciones. Comenzamos con el botón para añadir notas. Cuando pulsemos este botón la aplicación mostrará el diálogo para la edición de notas y, en caso de que se pulse el botón Aceptar, añadirá la nota al listado de notas de la tarea.

Lo último que hacemos tras asociar la nota con la tarea es modificar la selección en la tabla de tareas, con el objetivo de que se refresque la tabla de notas. Si no introdujésemos este código el usuario no vería la nueva nota hasta hacer manualmente el cambio de selección. Como este fragmento se utilizará más de una vez, lo sacamos a un método repaintNotes.

```
@Action (enabledProperty="taskSelected")
public void addNote() {
    int row = taskTable.getSelectedRow();
    Temas task = taskList.get(taskTable.convertRowIndexToModel(row));
    JFrame mainFrame = GestorDeTareasApp.getApplication().getMainFrame();
    NotasDialog noteDialog = new NotasDialog(mainFrame, true);
    Notas note = new Notas();
    note.setTemas(task);
    noteDialog.setNote(note);
    noteDialog.setLocationRelativeTo(mainFrame);
    noteDialog.setVisible(true);
    if (noteDialog.isNoteConfirmed()){
        Collection<Notas> cn = task.getNotasCollection();
        entityManager.persist(note);
        cn.add(note);
        entityManager.getTransaction().begin();
        entityManager.getTransaction().commit();
        repaintNotes(row);
    }
}
```

Código addNote 62

Seguimos con el botón para modificar, cuyo código sería el siguiente.

```
@Action (enabledProperty="noteSelected")
public void modifyNote() {
    int index = taskTable.getSelectedRow();
    Temas task = taskList.get(taskTable.convertRowIndexToModel(index));
    Collection<Notas> notesCollection = task.getNotasCollection();
    Notas note = (Notas)(notesCollection.toArray()[notesTable.getSelectedRow()]);

    JFrame mainFrame = GestorDeTareasApp.getApplication().getMainFrame();
    NotasDialog noteDialog = new NotasDialog(mainFrame, false);
    noteDialog.setLocationRelativeTo(mainFrame);
    noteDialog.setNote(note);
    noteDialog.setVisible(true);
    if (noteDialog.isNoteConfirmed()){
        entityManager.getTransaction().begin();
        entityManager.getTransaction().commit();
    }
    else{
        entityManager.refresh(noteDialog.getNote());
    }
    repaintNotes(index);
}
```

Código modifyNote 63

En este caso, cuando el usuario pulse el botón cancelar en el diálogo le diremos al entityManager que refresque la nota editada para desechar los cambios que se hubiesen realizado.

Por último, añadimos el método para eliminar una nota.

```
@Action (enabledProperty="noteSelected")
public void deleteNote() {
    int index = taskTable.getSelectedRow();
    Temas task = taskList.get(taskTable.convertRowIndexToModel(index));
    Collection<Notas> notesCollection = task.getNotasCollection();
    Notas toRemove = (Notas)(notesCollection.toArray()[notesTable.getSelectedRow()]);
    int n = JOptionPane.showConfirmDialog(null, "¿Está seguro de que desea eliminar la nota?",
            "Aviso", JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null);
    if (n == JOptionPane.YES_OPTION) {
        notesCollection.remove(toRemove);
        entityManager.remove(toRemove);
        entityManager.getTransaction().begin();
        entityManager.getTransaction().commit();
        repaintNotes(index);
    }
}
```

Código deleteNote 64

Completando la aplicación

En este punto tenemos la aplicación operativa, aunque queda algún detalle por pulir.

El primero de ellos es que si tratamos de eliminar una tarea con anotaciones asociadas se produce un error que nos indica que no se puede realizar la operación existiendo registros secundarios. La solución por la que vamos a optar es hacer que cada vez que se elimine

una tarea se eliminen en cascada sus notas, para lo que añadimos una anotación en la clase Temas.

```
@OneToMany(mappedBy = "temas", cascade=CascadeType.REMOVE)
private Collection<Notas> notasCollection;
```

Anotación deleteCascade 65

Por otro lado, si creamos algunas tareas y les asociamos notas, puede parecer que todo funciona correctamente. Sin embargo, si asociamos una nota a la primera tarea de la lista, veremos que para que se refresque la tabla de notas es necesario modificar la selección de la tabla de tareas. Esto se debe a que hemos tratado de forzar el refresco realizando un cambio de selección por código, pero siempre pasábamos a seleccionar la fila 0, y después de nuevo la que se estaba tratando. Lógicamente, cuando estamos tratando la primera fila, este cambio no surte ningún efecto.

Para resolver este problema vamos a utilizar una solución poco elegante, pero práctica. En el método para refrescar la tabla de notas, meteremos un elemento en la tabla de tareas, lo marcaremos como seleccionado, y después lo borraremos. De esta forma nos aseguramos que siempre se recargará la tabla con las anotaciones.

El código del método es el siguiente.

```
private void repaintNotes(int row){
    taskList.add(0, new Temas());
    taskTable.setRowSelectionInterval(0, 0);
    taskList.remove(0);
    if (row >= 0){
        taskTable.setRowSelectionInterval(row, row);
    }
}
```

Código repaintNotes 66

Con estos últimos cambios tenemos nuestra aplicación completa.

Conclusiones

Aunque su desarrollo se encuentra actualmente parado, el Swing Application Framework ofrece una interesante base para el desarrollo de aplicaciones de escritorio en Java. Además, el IDE NetBenas ofrece las herramientas necesarias para trabajar con él de una forma sencilla y cómoda.

En este tutorial nos hemos centrado en ver cómo trabajar en este entorno para construir una sencilla aplicación con acceso a una base de datos, en la que se ha trabajado con los principales conceptos relacionados, como entidades JPA, bindings, Actions, etc., ofreciendo una base para el desarrollo de aplicaciones más complejas y de mayor tamaño.

De cara al futuro, y teniendo en cuenta que Sun ha descartado por el momento seguir evolucionando SAF, existen otras alternativas a las que se debería prestar atención. Por un lado, existen frameworks alternativos, en su mayoría desarrollados por la comunidad como Better SAF, que parece haber tenido actividad en los últimos meses. Por otro, tenemos a JavaFX, que también permite la creación de aplicaciones Java de escritorio, con un aspecto más moderno.

Referencias

Por último un poco de información adicional, en inglés, mayormente.

- http://netbeans.org/kb/docs/java/gui-saf.html – Introducción a SAF, primeros conceptos y ejemplos.
- http://netbeans.org/kb/docs/java/gui-db-custom.html – Guía paso a paso para crear una aplicación con SAF, accediendo a base de datos.
- http://www.java.net/blog/150559 – Blog de Alexander Potochkin, último líder de SAF.
- http://jcp.org/en/jsr/detail?id=296 – La JSR 296, Swing Application Framework.
- https://appframework.dev.java.net/ - La página del framework.
- http://kenai.com/projects/bsaf – La del proyecto hijo.