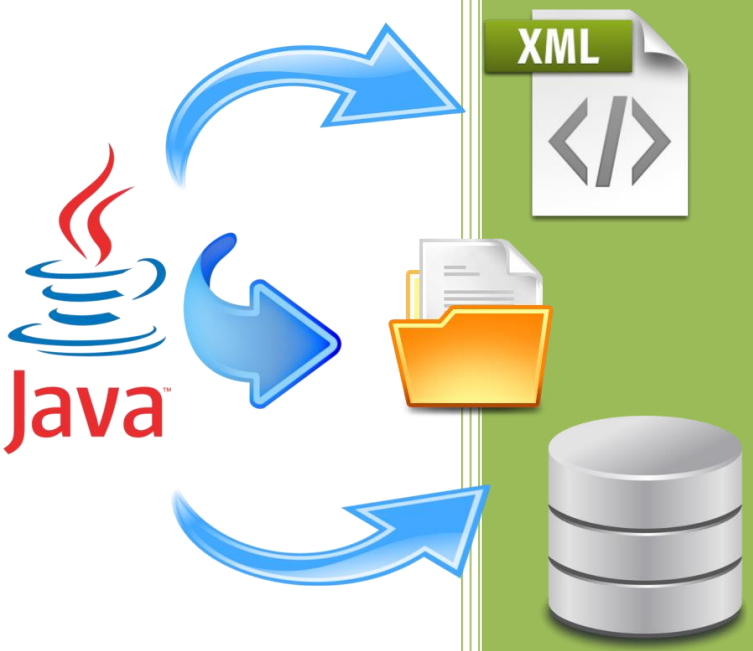
Desarrollo de Aplicaciones Multiplataforma

Informática y Comunicaciones

IES Valle del Jerte
Plasencia
informatica.iesvalledeljerteplasencia.es

ACCESO A DATOS

UT.6. Programación de componentes de acceso a datos



Ana Arribas Arjona

2º DAM

1. INTRODUCCIÓN

Los continuos avances en la Informática y las Telecomunicaciones están haciendo cambiar la forma en la que se desarrollan actualmente las aplicaciones software. Los modelos de programación existentes se ven incapaces de manejar la complejidad de los requisitos de los sistemas abiertos y distribuidos. Surgen nuevos paradigmas de programación como la programación orientada a componentes que supone una "extensión" de la programación orientada a objetos y está basada en la noción de COMPONENTE. El *Desarrollo de Software Basado en Componentes (DSBC)*, trata de sentar las bases para el diseño y desarrollo de aplicaciones distribuidas basadas en componentes software reutilizables.

1.1. Concepto de componente

Existen muchas definiciones de componentes software, pero una de las más difundidas es la de Szyperski, 1998:

"Un componente es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio".

El **objetivo** del desarrollo basado en componentes es construir aplicaciones mediante ensamblado de módulos software reutilizables, que han sido diseñados previamente con independencia de las aplicaciones en las que van a ser utilizados.

1.1.1. Características

Existen algunas características claves para que un elemento pueda ser catalogado como componente:

- **Independiente de la plataforma**: Hardware, Software, Sistema Operativo.
- **Identificable:** Debe tener una identificación que permita acceder fácilmente a sus servicios y que permita su clasificación.
- **Autocontenido:** Un componente no debe requerir de la utilización de otros para llevar a cabo la función para la cual fue diseñado.
- **Puede ser remplazado por otro componente:** Se puede remplazar por nuevas versiones u otro componente que lo mejore.
- **Con acceso solamente a través de su interfaz**: Una interfaz define el conjunto de operaciones que un componente puede realizar; estas operaciones se llaman también servicios o responsabilidades. Las interfaces proveen un mecanismo para interconectar componentes y controlar las dependencias entre ellos.
- **Sus servicios no varían**: Las funcionalidades ofrecidas en su interfaz no deben variar, pero su implementación sí.
- **Bien documentado**: Un componente debe estar correctamente documentado para facilitar su búsqueda si se quiere actualizar, integrar con otros, adaptarlo, etc.

- **Es genérico**: Sus servicios deben servir para varias aplicaciones.
- **Reutilizado dinámicamente:** Puede ser cargado en tiempo de ejecución en una aplicación.

Ejemplo de tecnologías de componentes son las siguientes:

- La plataforma .NET de Microsoft para sistemas Windows. Representa una nueva etapa en la evolución de *COM (Component Object Model),* la plataforma de componentes de Microsoft. Con ella, Microsoft impulsa la idea de "industrializar" el software utilizando tecnologías de componentes.
- *JavaBeans* y *EJB (Enterprise JavaBeans)* de Oracle Corporation (inicialmente desarrollado por Sun Microsystems). Es la tecnología de componentes basada en Java que funciona en cualquier plataforma.

A lo largo de este tema trabajaremos con los JavaBeans, la tecnología de componentes basada en Java.

1.1.2. Ventajas e Inconvenientes

Un componente es una pieza de software que describe o ejecuta funciones específicas a través de una interfaz bien definida. Se pueden juntar o combinar varios componentes para llevar a cabo una tarea. El uso de componentes aporta una serie de ventajas:

- Reutilización de software.
- Disminuye la complejidad del software. Se pueden ir probando trozos de componentes antes de probar el conjunto de todos los componentes ensamblados.
- Mejora el mantenimiento del sistema, los errores son más fáciles de detectar.
- Incrementa la calidad del software ya que un componente puede ser construido y luego mejorado.

El actual diseño basado en componentes también tiene limitaciones ya que solo existen en algunos campos como las GUIs y no siempre se pueden encontrar los componentes adecuados para cada proyecto. A esto hay que añadir la falta de estándares y la falta de procesos de certificación que garanticen la calidad de los componentes.

2. JavaBeans

Un JavaBean es un componente de software reutilizable que está escrito en lenguaje Java. Puede ser manipulado visualmente mediante herramientas de desarrollo Java. Ejemplos de JavaBeans son las librerías gráficas AWT *(Abstract Window Toolkit),* API de Java que permite hacer aplicaciones con componentes GUI (como ventanas, botones, barras, campos de texto, etc.); y SWT *(Standard Widget Toolkit*) de Eclipse.

A menudo nos referimos a un JavaBean como un Bean. Han de cumplir las siguientes características:

- **Introspección.** Mecanismo mediante el cual los propios JavaBeans proporcionan información sobre sus características (propiedades, métodos y eventos). Los Beans soportan la introspección de dos formas: utilizando patrones de nombrado (que son como reglas para nombrar las características del Bean) y proporcionando las características mediante una clase *Bean Information* relacionada.
- **Manejo de eventos.** Los Beans se comunican con otros Bean utilizando los eventos. Un Bean puede tener interés en recibir y responder a eventos enviados por otro Bean.
- **Propiedades.** Las propiedades definen las características de apariencia y comportamiento de un Bean que pueden ser modificadas durante el diseño de los componentes.
- **Persistencia.** Permite a los Beans almacenar su estado y restaurarlo posteriormente. Se basa en la serialización.
- **Personalización.** Los programadores pueden alterar la apariencia y conducta del Bean durante el diseño. La personalización se soporta de dos formas: utilizando editores de propiedades o utilizando personalizadores de Beans más sofisticados.

Un **JavaBean** es una clase Java que se define a través de las propiedades que expone, los métodos que ofrece y los eventos que atiende o genera. Su definición requiere ciertas reglas:

- Debe tener un constructor sin argumentos, aunque puede tener más de uno.
- Debe implementar la interfaz **Serializable** (para poder implementar **persistencia).**
- Sus propiedades deben ser accesibles mediante métodos **get** y **set.**
- Los nombres de los métodos deben obedecer a ciertas convenciones de nombrado.

Aunque los JavaBean se diseñaron para ser utilizados y manipulados por herramientas visuales, en este tema los trataremos como componentes de acceso a base de datos; y veremos cómo el mismo componente lo podemos usar para acceder a diferentes bases de datos.

No debemos confundir los *JavaBeans*, diseñados para crear componentes reusables que suelen usarse en herramientas de desarrollo IDE y en componentes visuales (aunque no exclusivamente), pensados para ser procesos locales; con la especificación *Enterprise Java Beans (EJB)* que describe un modelo de componentes del lado servidor, los componentes se almacenan en un servidor para ser invocados por los clientes.

2.1. Propiedades y Atributos

Las propiedades de un Bean son los atributos que determinan su apariencia y comportamiento. Por ejemplo, un *Producto* puede tener las siguientes propiedades: *descripción, pvp, stockactual, idproducto,* etc. Para acceder a las mismas se utilizan los métodos *getter* y *setter* de la siguiente forma:

```
public TipoPropiedad getNombrePropiedad() { ... }

public void setNombrePropiedad (TipoPropiedad valor) { ... }
```

Las propiedades pueden ser **simples, indexadas, ligadas** o **restringidas**.

PROPIEDADES SIMPLES:

Las propiedades simples representan un único valor; por ejemplo, si un atributo se llama *pvp* y es de tipo *float* los métodos *getter* y *setter* son los siguientes:

```
private float pvp;

public float getPvp() { return pvp; }
public void setPvp(float pvp) { this.pvp = pvp; }
```

Si la propiedad es booleana se escribe **isNombrePropiedad()** para obtener su valor:

```
private boolean conectado = false;

public boolean isConectado(){ return conectado; }
public void setConectado(boolean conectado) { this.conectado = conectado;}
```

PROPIEDADES INDEXADAS:

Las propiedades indexadas representan un array de valores a los que se accede mediante un índice. Se deben definir métodos *getter* y *setter* para acceder al array completo y a valores individuales. Ejemplo:

```
private int[] categorias = {1,2,3};

//métodos get y set para acceder al array completo
public void setCategorias(int[] valor){ categorías = valor; }
public int[] getCategorias(){ return categorías; }

//métodos get y set para acceder a valores individuales
public void setCategorias(int indice, int valor){
categorias[indice]=valor;}

public int getCategorias(int indice){ return categorias[indice];
}
```

PROPIEDADES LIGADAS:

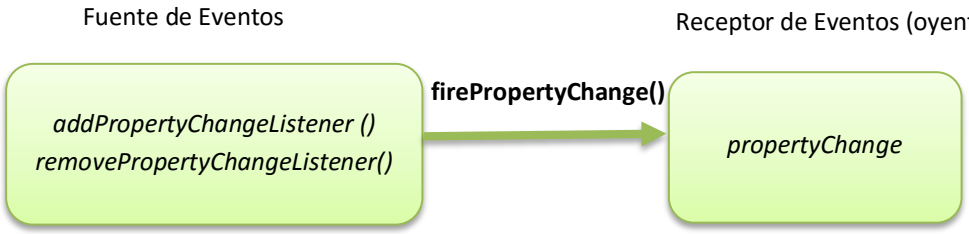
Son las propiedades asociadas a eventos. Cuando la propiedad cambia se notifica a todos los objetos interesados la naturaleza del cambio, permitiéndoles realizar alguna acción. Para que el Bean soporte propiedades ligadas (también llamadas compartidas), debe mantener una lista de los receptores de la propiedad y alertar a dichos receptores cuando cambie la propiedad; para ello proporciona una serie de métodos de la clase **PropertyChangeSupport.** La lista de receptores se mantiene gracias a los métodos ***addPropertyChangeListener()*** y ***removePropertyChangeListener().*** Un Bean que produce eventos (llamado Bean fuente) tiene la siguiente forma:

```
public class BeanFuente implements Serializable {

  private PropertyChangeSupport propertySupport;

  //en el constructor se indica el objeto al que se le quiere dar soporte
     public BeanFuente () {
     propertySupport = new PropertyChangeSupport(this); }

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

     public void removePropertyChangeListener(PropertyChangeListener
     listener) { propertySupport.removePropertyChangeListener(listener);

}
```

Para que nuestro Bean receptor pueda oir los eventos de cambio de propiedad, debe implementar el método ***propertyChange()*** de la interfaz **PropertyChangeListener.** El Bean fuente llama a este método de notificación de todos los receptores de su lista de receptores. La notificación no es mas que un objeto **PropertyChangeEvent** que encapsula la propiedad que ha cambiado (su antiguo y nuevo valor). El receptor cada vez que reciba este objeto lo examinará y podrá descubrir el cambio. Un Bean receptor (oyente) de un cambio de propiedad tiene la forma:

```
   public class BeanReceptor implements Serializable, PropertyChangeListener
{

      public void propertyChange(PropertyChangeEvent evt) {
         System.out.println("Valor anterior: "+ evt.getOldValue());
         System.out.println("Valor actual: "+ evt.getNewValue());
      }
```

Al Bean que tiene la propiedad ligada se le conoce como **fuente** y al que recibe la notificación de los cambios como **receptor**. La siguiente figura muestra los métodos que deben implementar los Bean fuente y receptor de eventos cuando utilizan propiedades ligadas, el lanzamiento del evento se realiza mediante el método ***firePropertyChange().***



PROPIEDADES RESTRINGIDAS:

Son similares a las propiedades ligadas, pero en este caso los objetos a los que se les notifica el cambio del valor de la propiedad pueden vetarlo si no se ajusta a unas determinadas características; la propiedad que puede dar lugar a un veto se le llama **restringida**. Se deben proporcionar dos métodos de registro para los receptores;

```
public void addVetoableChangeListener(VetoableChangeListener listener);
public void removeVetoableChangeListener(VetoableChangeListener listener);
```

El método ***addVetoableChangeListener()*** hace que el objeto oyente (listener) sea capaz de capturar eventos de tipo **PropertyVetoEvent** y el método ***removeVetoableChangeListener()*** hace que el objeto oyente deje de escuchar los eventos de veto. Para definir estos métodos se puede crear un objeto **VetoableChangeSupport** para conseguir un soporte sencillo a fin de definir los métodos anteriores:

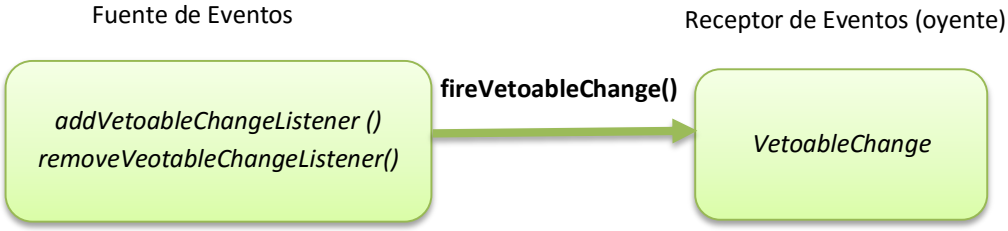
```
public class BeanFuente implements Serializable {
    private VetoableChangeSupport soporteVeto;
    . . . . .
    //en el constructor se indica el objeto al que se le quiere dar soporte
    public BeanFuente () {
    soporteVeto = new VetoableChangeSupport(this); }
    . . . . .
    public void addVetoableChangeListener(VetoableChangeListener listener) {
    soporteVeto.addVetoableChangeListener(listener);
    }
    public void removeVetoableChangeListener(VetoableChangeListener
    listener){ soporteVeto.removeVetoableChangeListener(listener);
    }

}
```

Para que el Bean receptor pueda oir los eventos debe implementar el método ***vetoableChange()*** de la interfaz **VetoableChangeListener.** El método recibe una copia del objeto **PropertyChangeEvent** y lanza excepciones del tipo ***PropertyVetoException*** (mediante la instrucción **throw)** cuando el cambio de valor en la propiedad no se pueda realizar debido a que no cumple una condición. El lanzamiento de estas excepciones tiene esta sintaxis: *throw new PropertyVetoException(String mensaje, PropertyChangeEvent evento):*

```
public class BeanReceptor implements Serializable, VetoableChangeListener {

    . . . . .
  public void vetoableChange(PropertyChangeEvent evt) throws
  PropertyVetoException{ //comprobación de las condiciones

    . . . . .
  //se lanza excepción si no se aprueba el cambio
  throw new PropertyVetoException ("mensaje",evt);
    }

    . . . . .
 }
```

El objeto de excepción lanzado permite obtener el mensaje (método *getMessage()*) y el evento que desencadenó la excepción (*getPropertyChangeEvent()*). El Bean que tiene la propiedad restringida captura la excepción y asigna a la propiedad el valor anterior, sin tener que notificar a todas las partes interesadas el nuevo cambio de valor.

La siguiente figura muestra los métodos que deben implementar los Bean fuente y receptor de eventos cuando utilizan propiedades restringidas, el lanzamiento del evento se realiza mediante el método ***fireVetoableChange()***. Las propiedades restringidas son más complejas de implementar que las ligadas.



2.2. Eventos

Los Beans utilizan los eventos para comunicarse con otros Beans. Para lanzar eventos **PropertyChangeEvent** a los oyentes, se utiliza el método ***firePropertyChange()*** cada vez que se cambia el valor de la propiedad. A este método se le pasan tres parámetros: el nombre de la propiedad, el valor antiguo y el valor nuevo. Los valores antiguos y nuevos deben de ser objetos *(Object),* lo que obliga a utilizar las clases envoltorio Integer, Double, Boolean, Char, etc. para poder pasar valores de tipos básicos.

El siguiente ejemplo muestra el método *set* para la propiedad *stockactual* en el Bean fuente. Si el stock actual es menor que el stock mínimo (es decir hay un cambio en la propiedad que nos interesa controlar) se lanza un evento mediante el método ***firePropertyChange(),*** al que se le envían los 3 parámetros ya indicados anteriormente:

```
public void setStockactual(int valorNuevo) {
int valorAnterior = stockactual;
```

```
stockactual = valorNuevo;
    if (stockactual < getStockminimo()) //hay que realizar pedido
       propertySupport.firePropertyChange("stockactual", valorAnterior,
                                                    stockactual);
```

El Bean receptor de eventos de tipo **PropertyChangeEvent** debe implementar el método *propertyChange()* de la interfaz **PropertyChangeListener.** Este método es llamado cuando se modifica el valor de la propiedad. El evento **PropertyChangeEvent** que recibe este método proporciona varios métodos:

- *Object getOldValue():* Obtiene el antiguo valor de la propiedad.
- *Object getNewValue():* Obtiene el nuevo valor de la propiedad.
- *String getPropertyName():* Obtiene el nombre de la propiedad que cambió (puede ser null si cambiaron varias propiedades a la vez).

En el caso de las propiedades restringidas, para lanzar eventos **PropertyVetoEvent** a los oyentes, se utiliza el método ***fireVetoableChange()*** que también requiere tres parámetros: el nombre de la propiedad que cambia, el valor antiguo y el valor nuevo de la propiedad; este método debe capturar excepciones *PropertyVetoException:*

```
public void setStockactual(int  valorNuevo)
    {
try {
     int valorAnterior = stockactual;
     stockactual = valorNuevo;
     if (stockactual < getStockminimo()) //hay que realizar pedido
        soporteVeto.fireVetoableChange("stockactual", valorAnterior,
                                                    stockactual);
    } catch(PropertyVetoException pve ){
      //Código que se ejecuta en caso de excepción

}
```

El Bean receptor de eventos de tipo **PropertyVetoEvent** debe implementar el método *VetoableChange()* de la interfaz **VetoableChangeListener.**

2.3. Persistencia del componente

Mediante el mecanismo de persistencia un Bean es capaz de almacenar su estado en un momento determinado y recuperarlo posteriormente. Para que un Bean sea persistente debe implementar la interfaz **Serializable** usando las librerías APIs de serialización de Java; entonces, todos los valores del Bean serán trasladados a una cadena de bytes que se almacenarán en un fichero. Esta cadena contendrá la información suficiente para reconstruir el Bean almacenado en su último estado. Cuando se carga el componente en el programa se hace a partir del fichero serializado.

A la hora de implementar la interfaz **Serializable** hay que tener en cuenta que:

1. Las clases que implementan **Serializable** deben tener un constructor sin argumentos.
2. Todos los campos excepto **static** y **transient** son serializados. Para especificar los campos que no queremos serializar utilizaremos el modificador **transient:**

```
transient int campo;
```

Hemos de tener en cuenta que se deben guardar aquellas características del Bean que le permitan reincorporarse al estado en que se encontraba.

3. Herramienta para el desarrollo de componentes

En este apartado vamos a crear dos Beans. El primer Bean (fuente) de nombre **Producto** tiene una propiedad ligada denominada stockactual de tipo int. El segundo Bean (receptor) de nombre **Pedido** está interesado en los cambios de dicha propiedad cuando sea inferior al stock mínimo. Supongamos que el problema que se trata de resolver es que cuando el stock actual de un producto sea inferior al stock minimo se debe generar un pedido. Crearemos dos clases con los siguientes atributos y que implementarán las siguientes interfaces:

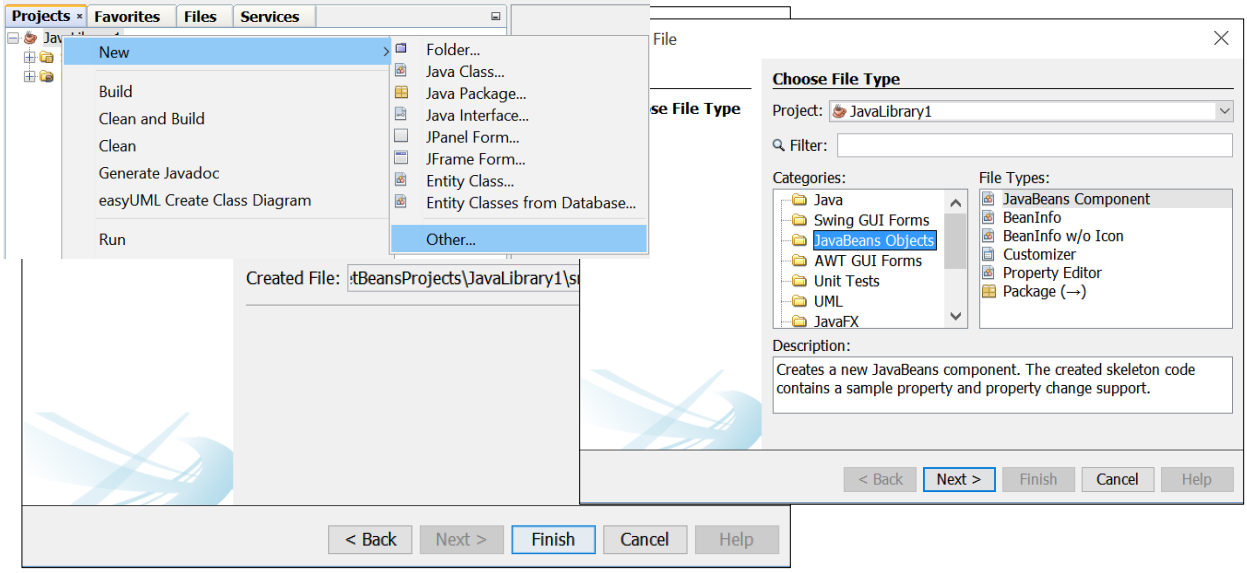
CLASE PRODUCTO, IMPLEMENTS SERIALIZABLE. FUENTE DE EVENTOS	**CLASE PEDIDO, IMPLEMENTS SERIALIZABLE, PROPERTYCHANGELISTENER. RECEPTOR DE EVENTOS**

`private String descripción;`
`private int idproducto;`
`private int stockactual;`
`private int stockminimo;`
`private float pvp;`
	`private int numeropedido;`
	`private int idproducto;`
	`private Date fecha;`
	`private int cantidad;`
	`private boolean pedir;`

3.1. Crear JavaBeans con Netbeans

Abrimos NetBeans, nos vamos a crear un nuevo proyecto de tipo *Biblioteca* (*Library*). Se selecciona **File→New Project→ Java ClassLibrary** y pulsamos el botón Next; y en la siguiente pantalla si lo deseamos cambiamos el nombre del proyecto, en nuestro caso dejaremos el nombre de *JavaLibrary1*.

Una vez creado el proyecto vamos a crearnos los JavaBeans. Nos ponemos encima del nombre del proyecto, y con el botón derecho del ratón pulsamos en **New→Other… →JavaBeans Object**; a continuación, escribimos el nombre del Bean (Producto) y el nombre del paquete del que formará parte, en el ejemplo MisBeans. Pulsamos el botón Finish



Se crea el ***Bean Producto*** con un código por defecto. Dentro del código aparece una propiedad ejemplo de nombre *sampleProperty* con sus métodos *get* y *set* correspondientes, se crea un objeto PropertyChangeSupport de nombre *propertySupport* para conseguir un soporte sencillo a fin de definir los métodos anteriores *addPropertyChangeListener()* y *removePropertyChangeListener().* El código es el siguiente:

```
/*
 * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package MisBeans;

import java.beans.*;
import java.io.Serializable;

/**
 *
 * @author Ana Arribas
 */
public class Producto implements Serializable {

    public static final String PROP_SAMPLE_PROPERTY = "sampleProperty";

    private String sampleProperty;
```

```
    private PropertyChangeSupport propertySupport;

    public Producto() {
        propertySupport = new PropertyChangeSupport(this);
    }

    public String getSampleProperty() {
        return sampleProperty;
    }

    public void setSampleProperty(String value) {
        String oldValue = sampleProperty;
        sampleProperty = value;
            propertySupport.firePropertyChange(PROP_SAMPLE_PROPERTY,
     oldValue, sampleProperty);
    }

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

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

}
```

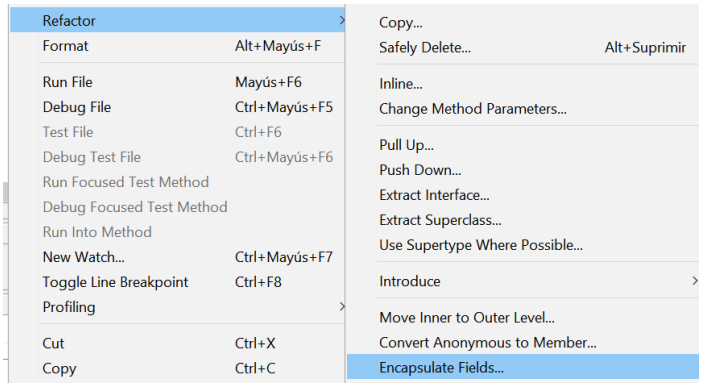
Modificamos el código del JavaBean. Eliminamos el atributo *sampleProperty* y sus métodos `get` y `set`. Y añadimos los siguientes atributos:

```
private String description;
private int idproducto;
private int stockactual;
private int stockminimo;
private double pvp;
```

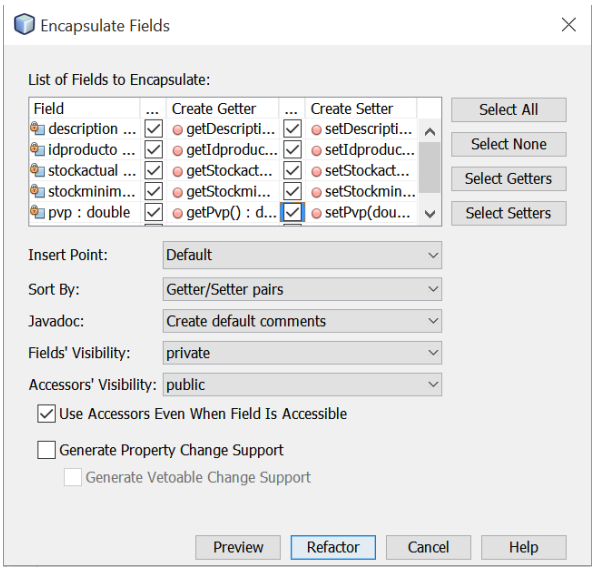
Añadimos el siguiente constructor:

```
public Producto(int idproducto,String descripcion, int stockactual,int
stockminimo, double pvp) {
propertySupport = new PropertyChangeSupport(this);
this.idproducto=idproducto;
this.description=descripcion;
this.stockactual=stockactual;
this.stockminimo=stockminimo;
this.pvp=pvp;
    }
```

Y generamos los getter y setter para los atributos de forma automática. Nos ponemos encima del código, pulsamos botón derecho **Refactor→Encapsulate Fields**



Y seleccionamos los getter y setter de los atributos a crear y pulsamos **Refactor**:



Y cambiamos el el método **`setStockactual()`** por el siguiente, para generar un evento si el stock actual es menor que el mínimo (se usa el método **`firePropertyChange()`**):

```
    /**
     * @param stockactual the stockactual to set
     */
    public void setStockactual(int valorNuevo) {
        int valorAnterior = stockactual;
        stockactual = valorNuevo;
        if (stockactual < getStockminimo())
        propertySupport.firePropertyChange("stockactual", valorAnterior,
stockactual);

    }
```

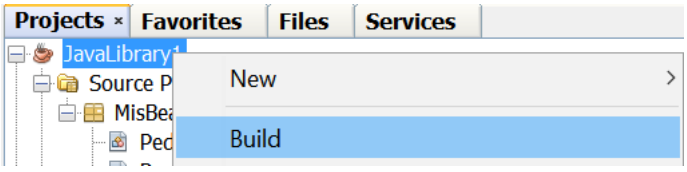
Creamos (como antes) el JavaBean ***Pedido*** en el paquete *MisBeans.* Eliminamos todo el código de la clase y añadimos **`PropertyChangeListener`** a la cláusula *implements;* será necesario sobreescribir el método **`propertyChange()`.** En este método es donde indicamos las acciones a realizar cuando el stock actual es menor que el stock mínimo, debe quedar algo parecido a esto:

```
/*
 * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package MisBeans;
```

```
import java.beans.*;
import java.io.Serializable;
import java.util.Date;

/**
 *
 * @author Ana Arribas
 */
public class Pedido implements Serializable, PropertyChangeListener {

    private int numeropedido;
    private int idproducto;
    private Date fecha;
    private int cantidad;
    private boolean pedir;//se usa para controlar si hay que hacer pedido

    @Override
    public void propertyChange(PropertyChangeEvent pce) {
        System.out.println("Stock anterior: " + pce.getOldValue());
        System.out.println("Stock actual: " + pce.getNewValue());
        setPedir(true); //hay que hacer pedido

    }

    public Pedido() {
    }

    public Pedido(int numeropedido, int idproducto, java.sql.Date fecha,
int cantidad) {
        this.numeropedido = numeropedido;
        this.idproducto = idproducto;
        this.fecha = fecha;
        this.cantidad = cantidad;
    }

    /**
     * @return the numeropedido
     */
    public int getNumeropedido() {
        return numeropedido;
    }

    /**
     * @param numeropedido the numeropedido to set
     */
    public void setNumeropedido(int numeropedido) {
        this.numeropedido = numeropedido;
    }

    /**
     * @return the idproducto
     */
    public int getIdproducto() {
        return idproducto;
    }

    /**
     * @param idproducto the idproducto to set
     */
    public void setIdproducto(int idproducto) {
```

```
        this.idproducto = idproducto;
    }

    /**
     * @return the fecha
     */
    public Date getFecha() {
        return fecha;
    }

    /**
     * @param fecha the fecha to set
     */
    public void setFecha(Date fecha) {
        this.fecha = fecha;
    }

    /**
     * @return the cantidad
     */
    public int getCantidad() {
        return cantidad;
    }

    /**
     * @param cantidad the cantidad to set
     */
    public void setCantidad(int cantidad) {
        this.cantidad = cantidad;
    }

    /**
     * @return the pedir
     */
    public boolean isPedir() {
        return pedir;
    }

    /**
     * @param pedir the pedir to set
     */
    public void setPedir(boolean pedir) {
        this.pedir = pedir;
    }

}
```

Una vez que tenemos los JavaBeans, generamos el JAR. Para ello, pulsamos con el botón derecho del ratón en el proyecto y a continuación en la opción de menú **Build**:



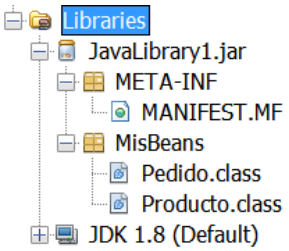
Si todo va bien, se generará el archivo ***JavaLibrary1.jar*** en la carpeta **dist** del proyecto, en el ejemplo en:

```
C:\Users\Ana Arribas\Documents\NetBeansProjects\JavaLibrary1\dist
```

Si posteriormente modificamos los JavaBean pulsamos en la opción **Clean and Buid.**

Ahora vamos a probar los componentes, para ello nos vamos a crear un proyecto Java, por ejemplo, **PruebaJavaBean** con una clase con el mismo nombre e incluimos *JavaLibrary1.jar*. Para ello pulsamos con el botón derecho del ratón en **Libraries→ Add JAR/Folder**, localizamos el archivo jar y los añadimos.

Una vez realizado, al abrir el nodo *Libraries* podemos ver la librería incluida con el paquete MisBeans creado anteriormente y la carpeta META-INF con el archivo MANIFEST.MF que se genera al crear el JAR.



Para detectar si hay que realizar un pedido en el producto, utilizamos el método `addPropertyChangeListener()` del objeto Producto para agregar a la lista de oyentes el objeto Pedido, que es el que está interesado en el cambio del stock: **`producto.addPropertyChangeListener(pedido)`**. El programa quedaría de la siguiente forma:

```
package pruebajavabean;

import MisBeans.Pedido;
import MisBeans.Producto;
/**
 *
 * @author Ana Arribas
 */
public class PruebaJavaBean {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
      Producto producto= new Producto(1,"Bombilla led 40w",10,3, 1.5);
      Pedido pedido= new Pedido();

      producto.addPropertyChangeListener(pedido);
      producto.setStockactual(2);
      if(pedido.isPedir()){
            System.out.println("Realizar pedido en producto: "+
producto.getDescription());
       }
      else
      {System.out.println("No es necesario realizar un pedido en producto:
"+ producto.getDescription());}
    }
   }
```

Lo ejecutamos y el resultado sería:

```
Stock anterior: 10
Stock actual: 2
Realizar pedido en producto: Bombilla led 40w
```

ACTIVIDADES

1. Crea un proyecto en Eclipse para probar la librería *(JavaLibrary1.jar).* Define varios productos y cambia el stock actual en alguno de ellos. Visualiza por cada producto su stock anterior y actual.

2. Tenemos las siguientes clases:

- *Cliente*(IdCliente, nombre, apellidos, saldodebe, saldohaber)
- *Préstamo*(IdPrestamo, idCliente, fechaprestamo, interes)

Crea los javabeans necesarios para implementar la siguiente funcionalidad:

- Tenemos una aplicación de servicios financieros, y cuando el saldodebe de los clientes es superior al saldohaber le sugerimos la realización de un prestámo financiero.

3. Tenemos las siguientes clases:

- *Libro* (ISBN, Titulo, Autor, Editorial, NumPag)
- *Socio* (Codsocio, Nombre, Apellidos, dirección, teléfono, sancionado)
- *Préstamo* (CodPrestamo, ISBN, CodSocio, FechaPrestamo, FechaDevolución)

Crea los javabeans necesarios para implementar la siguiente funcionalidad:

- Tenemos una biblioteca, con una serie de libros, los socios pueden hacer préstamos de libros y tienen que devolver el libro en un plazo de 15 días. Si se devuelve el libro pasado los 15 días el socio será sancionado (campo booleano de la clase Socio).

4. Queremos gestionar los avisos de ocupación del parking situado en la puerta de Talavera. Para dicha gestión guardaremos la siguiente información:

- Parking: Nombre, dirección, plazas libres, plazas ocupadas.
- Aparcamiento: matricula, fecha de entrada y fecha de salida.

Los avisos a mostrar en el cartel luminoso a la entrada del parking serán:

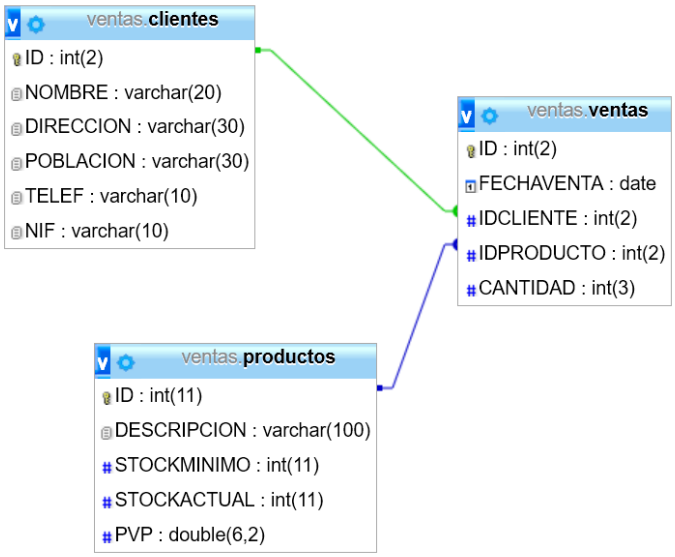
- *PLAZAS LIBRES*: Cuando el número de plazas libres es mayor que el de plazas ocupadas.

- *COMPLETO:* Cuando no haya plazas libres.
- *POCAS PLAZAS*: cuando el número de plazas ocupadas sea el 80% del total.

Crea lo componentes (javabeans) necesarios para gestionar los avisos del parking, una vez creado los componentes créate un proyecto nuevo para probarlo (varias entradas y salidas del parking)

3.2. Uso de un componente para acceder a bases de datos SQL

En el siguiente ejemplo vamos a desarrollar un componente que nos permitirá acceder a una base de datos basada en SQL. Las tablas que usaremos se llaman PRODUCTOS, VENTAS y CLIENTES, tal como se muestra a continuación:



Nos crearemos un Bean (BaseDatosBean) para acceder a la BD Ventas y que contendrá los siguientes atributos:

```
private static Connection conexion; //objeto Connection para conectarse a la BD
private String urldb; //url para acceder a la BD
private String usuario; //nombre de usuario
private String clave; //clave del usuario
private String driver; //driver para la conexión a la BD
private boolean crearConexion; //atributo para saber si la conexión está creada
```

Y métodos:

Desde el constructor con parámetros se inician los atributos para la conexión a la BD.

- **`public void`** `setCrearConexion()`*:* carga el driver y establece la conexión con la BD.

- **`public boolean`** `getCrearConexion()`: devuelve el valor del atributo *crearConexion.*
- **`public Connection`** `getConexion()`: devuelve el objeto *conexion.*
- **`public void`** `cerrarConexion()`: cierra la conexión a la base de datos.
- **`public int`** `obtenerUltimoID(String tabla)`*:* obtiene el último identificador de la tabla que recibe como parámetro.
- **`public int`** `insertarProducto(Producto producto)`: inserta una venta, recibe un objeto Venta.
- **`ArrayList<Producto>`** `consultaProductos(String consulta)`*:* devuelve una lista de productos consultados.

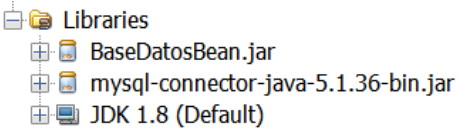
El código completo de la clase **BaseDatos.java** es:

```
package MisBeans;

import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;

/**
 *
 * @author Ana Arribas
 */
public class BaseDatos {

    private static Connection conexion; //objeto Connection para conectarse
a la BD
    private String urldb; //url para acceder a la BD
    private String usuario; //nombre de usuario
    private String clave; //clave del usuario
    private String driver; //driver para la conexión a la BD
    private boolean crearConexion; //atributo para saber si la conexión
está creada

//Constructores
    public BaseDatos() {
    }

    public BaseDatos(String urldb, String usuario, String clave, String
driver) {
        this.urldb = urldb;
        this.usuario = usuario;
        this.clave = clave;
        this.driver = driver;
    }

    public void setCrearConexion()//carga el driver y establece la conexión
con la BD
```

```
    {
        crearConexion = false;
        try {
            Class.forName(driver);
            conexion = DriverManager.getConnection(urldb, usuario, clave);
            crearConexion = true;
        } catch (Exception e) {
            System.out.println("PROBLEMAS EN LA CONEXION ....");
            e.printStackTrace();
        }
    }

    // devuelve el valor del atributo crearConexion
    public boolean getCrearConexion() {
        return crearConexion;
    }

// devuelve el objeto conexión.
    public Connection getConexion() {
        return conexion;
    }
    public void cerrarConexion() {// cierra la conexión a la BD
        try {
            conexion.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }

    }

//Obtener ultimo identificador de la tabla que recibe
    public int obtenerUltimoID(String tabla) {
        int id = 0;
        String consulta = "SELECT MAX(ID) FROM " + tabla;
        try {
            Statement st = getConexion().createStatement();
            ResultSet rs = st.executeQuery(consulta);
            rs.next();
            id = rs.getInt(1) + 1;
            rs.close();
            st.close();
        } catch (SQLException e) {
            System.out.println("PROBLEMAS AL OBTENER MAXIMO ID EN " +
tabla);
            e.printStackTrace();
        }
        return id;
    } //fin obtener último ID


    //Insertar un producto
    public int insertarProducto(Producto prod) {
        int filas = 0;

        String sql = "INSERT INTO PRODUCTOS VALUES (?,?,?,?,?)";
        try {
            PreparedStatement ps = getConexion().prepareStatement(sql);
            ps.setInt(1, prod.getIdproducto()); // idproducto
            ps.setString(2, prod.getDescription());// description
            ps.setInt(3, prod.getStockactual()); // stockactual
            ps.setInt(4, prod.getStockminimo()); // stockminimo
```

```
            ps.setDouble(5, prod.getPvp()); // pvp

            filas = ps.executeUpdate();   // filas afectadas
            ps.close();
        } catch (SQLException e) {
            System.out.println("ERROR AL INSERTAR PRODUCTO");
            e.printStackTrace();
        }
        return filas;
    }

    //Devuelve una lista de productos consultados
    public ArrayList<Producto> consultaProductos(String consulta) {
        ArrayList<Producto> lista = new ArrayList<Producto>();
        try {
            Statement st = getConexion().createStatement();
            ResultSet rs = st.executeQuery(consulta);

            while (rs.next()) {
                Producto p = new Producto(rs.getInt(1), rs.getString(2),
                        rs.getInt(3), rs.getInt(4), rs.getFloat(5));
                lista.add(p);
            }
            rs.close();
            st.close();
        } catch (SQLException e) {
            System.out.println("PROBLEMAS AL CONSULTAR PRODUCTOS");
            e.printStackTrace();
        }
        return lista;
    }

}
```

Compilamos la clase y generamos el jar, para probar el componente nos crearemos un nuevo proyecto donde agregaremos el driver de MySQL y el JAR del componente:



El código de la clase para probar el componente sería:

```
package pruebabd;

import MisBeans.BaseDatos;
import MisBeans.Producto;
import java.util.ArrayList;

/**
 *
 * @author Ana Arribas
 */
public class PruebaBD {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        String urldb = "jdbc:mysql://localhost/ventas";
```

```
        String usuario = "root";
        String clave = "";
        String driver = "com.mysql.jdbc.Driver";

        //Se crea un objeto BaseDatos
        BaseDatos bd = new BaseDatos(urldb, usuario, clave, driver);
        bd.setCrearConexion();//se crea la conexión a la BD
        if (bd.getCrearConexion()) {
            System.out.println("Conectado");

       System.out.println("==========================================");
       System.out.println("LISTA INICIAL DE PRODUCTOS");
            VerProductos(bd);
            InsertarProducto(bd);
        }
        bd.cerrarConexion();
    }

// Visualizar los productos
    private static void VerProductos(BaseDatos bd) {
        ArrayList<Producto> lista = new ArrayList<Producto>();
        lista = bd.consultaProductos("SELECT * FROM PRODUCTOS");
        if (lista != null) {
            for (int i = 0; i < lista.size(); i++) {
                Producto p = (Producto) lista.get(i);
                System.out.println("ID=>" + p.getIdproducto() + ": " +
p.getDescription()
                        + " Stock: " + p.getStockactual() + " Pvp: " +
p.getPvp() + "Stock Minimo:"
                        + p.getStockminimo());

            }
        }
    }

    private static void InsertarProducto(BaseDatos bd)
    {
      int id=bd.obtenerUltimoID("Productos");
      Producto producto= new Producto(id,"detergente",10,30,7.10);
      int filas=bd.insertarProducto(producto);
      System.out.println("Filas insertadas: "+ filas);

    }

}
```

ACTIVIDADES

5. A partir del componente anterior (*BaseDatosBean*) modifícalo para añadirle las siguientes funcionalidades:

- Alta de nuevos clientes.
- Consulta de clientes a partir de los campos NIF y NOMBRE.
- Realizar nuevas ventas, para lo cual habrá que consultar y actualizar el stock de los productos que se compran.
- Consultar las ventas de un determinado cliente.

- Consultar el total vendido en un determinado mes.

Una vez añadidas las nuevas funcionalidades, créate un proyecto para probarlas.

6. Create un nuevo proyecto para probar el componente BaseDatosBean sobre la BD de Oracle (deberás crearte la BD de Ventas en Oracle).

7. (opcional) Queremos controlar la climatización del CPD (Centro de Proceso de Datos) del Instituto. El sistema de climatización está formado por los siguientes componentes:

- **Termostato** (temperatura, máxima, mínima)
- **Climatizador** (Marca, modelo, modo)

El funcionamiento sería el siguiente:

- Si la temperatura ambiente es mayor que la temperatura máxima (28 ºC) se activa el modo aire acondicionado (F) del climatizador.
- Si la temperatura ambiente es inferior a la temperatura mínima (4 ºC) se activa el modo calefacción(C) del climatizador.

Mediante JavaBeans, simula el funcionamiento del sistema de climatización del CPD.