
M6.UF4.A6.P1
COMPONENTES DE
ACCESO A DATOS
JAVABEANS

Eduard Lara

INDICE

1. Javabeen
2. Propiedades
3. Ejemplo
4. Practica 1
5. Practica 2
6. Practica 3

1. JAVABEANS

¿Qué es un JavaBean?

- Es un estándar o unas normas a seguir para crear una nueva clase y que conociendo el lenguaje de programación Java, no reviste ninguna complejidad.
- Este modelo a seguir fue creado por Sun Microsystems, la cual fue adquirida por Oracle en el año 2010.
- Sun definieron un **JavaBean** como un componente software reutilizable que se puede manipular visualmente en una herramienta de construcción.
- Estos componentes no hay que confundirlos con un Enterprise JavaBean (EJB) que es una tecnología del lado del servidor incluida en Java EE.

1. JAVABEANS

Concepto de componente

- Componente: unidad de composición de aplicaciones software.
- Un nuevo paradigma: DSBC: desarrollo de software basado en componentes
→ trata de sentar las bases para el diseño y desarrollo de aplicaciones distribuidas basadas en componentes de software reutilizables
- Características:
 - Independiente de la plataforma
 - Identificable
 - Autocontenido
 - Puede ser reemplazado por otro componente
 - Con acceso solamente a través de su interfaz
 - Sus servicios (funcionalidades) no varían (puede cambiar la implementación)
 - Bien documentado. Se distribuye como un paquete

1. JAVABEANS

Concepto de componente

La forma de especificar, implementar, o empaquetar un componente depende de la tecnología utilizada. Las tecnologías basadas en componentes incluyen dos elementos:

- **Modelo de componentes:** especifica la reglas de diseño de los componentes y sus interfaces
- **Plataforma de componentes:** es la infraestructura de software requerida para la ejecución de aplicaciones basadas en componentes.

Ejemplo de tecnologías de componentes son:

- La plataforma .NET de Microsoft para sistemas Windows.
- JavaBeans y EJB de Oracle.

Nosotros veremos algo de JavaBeans

1. JAVABEANS

- Un JavaBean es un componente de software reutilizable escrito en Java.
- Características:
 - **Introspección:** mecanismo mediante el cual los propios JavaBeans proporcionan información sobre sus características. Los Beans soportan la introspección de dos formas: utilizando patrones de nombrado y proporcionando las características mediante una clase Bean Information relacionada.
 - **Manejo de eventos:** los Beans se comunican con otros mediante eventos.
 - **Propiedades:** definen las características de apariencia y comportamiento
 - **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.

1. JAVABEANS

- 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.
- Un JavaBean debe cumplir las siguientes características/reglas para ser considerado como tal y conseguir que sean realmente componentes reutilizables.
 - **Debe tener un constructor vacío** sin argumentos (puede tener más).
 - **Debe implementar la interfaz Serializable** (para poder implementar persistencia) la interfaz Externalizable. Se suelen usar para encapsular varios objetos en uno, de manera que implementando alguna de estas interfaces se puede guardar el estado de cada objeto.
 - **Todas las variables de la instancia deben ser privadas.**
 - **Debe tener métodos getters y setters para acceder a sus propiedades**
 - Los nombres de los métodos deben obedecer a ciertas convenciones de nombrado

1. JAVABEANS

POR QUÉ SE USA LA INTERFAZ SERIALIZABLE?

- Por lo general la información que se persiste debe viajar mediante la red a un servidor por lo que un objeto que se envía a guardar debe ser descompuesto en bytes, la interfaz serializable permite que un objeto sea descompuesto a bytes y que al otro lado pueda ser reconstruido.

CUANDO SE USA UN JAVABEAN?

- Un JavaBean se usa para crear un objeto y poderlo persistir en una base de datos, también permite, bien sea mostrar el objeto en la vista o capturar sus datos de la vista y posteriormente persistirlos en la base de datos.
- Por lo general un JavaBean se asocia con una tabla en la base de datos, de tal forma que las columnas de la tabla en la base de datos deben ser propiedades/atributos en un JavaBean, aunque pueden existir más propiedades, todo dependerá de las necesidades del programador.

1. JAVABEANS

CÓMO ACCEDER A UN JAVABEAN DESDE UNA PÁGINA JSP?

- Básicamente esta es la línea que se debe incluir en una página JSP para poder declarar un bean y acceder a sus propiedades.
- Primero se le da un id, que es cualquier nombre que identifique al bean, luego se debe poner el paquete donde se encuentra la clase incluido el nombre de la clase y finalmente el alcance.
- `<jsp:useBean id="nombre_beat" class="paquete_pertenece" scope="session">`

CÓMO ACCEDER LAS PROPIEDADES DE UN JAVABEAN?

- Para esto se usa la propiedad getProperty, en el name ponemos el valor del id declarado en el useBean, la propiedad property el atributo de la clase:
- `<jsp:getProperty name="id_anterior" property="atributo_de_clase" />`
- En la práctica el acceso a un JavaBean se hace usando JSTL puro, puesto que es menos engorroso que el formato anterior

2. PROPIEDADES Y ATRIBUTOS

- Las propiedades de un Bean son los atributos que determinan su apariencia y comportamiento.
- Para acceder a las propiedades deben existir los correspondientes getter y setter
- Las propiedades pueden ser:
 - Simples
 - Indexadas
 - Ligadas
 - Restringidas

2. PROPIEDADES

Propiedades Simple

- Representan un único valor
- Si la propiedad es booleana se escribe “isNombrePropiedad()” para obtener su valor

2. PROPIEDADES

Propiedades indexadas

- Representan un array de valores a los que se accede mediante índice.
- Se deben definir métodos getter y setter para acceder al array completo y a valores individuales

```
private int[] categorias={1,2,3};  
public void setCategorias(int[] valor){ this.categorias=valor;  
public int[] getCategorias() { return this.categorias;}  
public void setCategorias(int indice, int valor) {this.categorias(indice)=valor;}  
public int getCategorias(int indice) {return this.categorias(indice);}
```

2. PROPIEDADES

Propiedades Ligadas

- Son las propiedades asociadas a eventos.
- Cuando la propiedad cambia se notifica a todos los objetos interesados en el cambio, permitiéndoles realizar alguna acción.
- Para que el Bean soporte propiedades ligadas (o 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()**

2. PROPIEDADES

JavaBean Fuente de Eventos

addPropertyChangeListener()
removePropertyChangeListener()



firePropertyChange()

JavaBean Receptor de Eventos

propertyChange()

3. EJEMPLO

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
- El problema es que cuando el stock actual de un producto sea inferior al stock mínimo se debe generar un pedido.

3. EJEMPLO

Bean Producto: es la fuente del evento que generará el evento por falta de stock

```
public class Producto implements Serializable {
    public static final String PROP_SAMPLE_PROPERTY = "sampleProperty";
    private int idproducto;
    private int stockactual;
    private int stockminimo;
    private float pvp;
    private String descripcion;
    private PropertyChangeSupport propertySupport;

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

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

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

    public PropertyChangeSupport getPropertySupport() {
        return propertySupport;
    }
}
```


3. EJEMPLO

Bean Producto: tiene una propiedad ligada denominada *stockactual* de tipo int

```
public void setPropertySupport(PropertyChangeSupport propertySupport) {
    this.propertySupport = propertySupport;
}

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

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

    if (this.stockactual < getStockminimo()) // hay que realizar pedido
    {
        propertySupport.firePropertyChange("stockactual", valorAnterior, this.stockactual);
        this.stockactual = valorAnterior; // dejamos el stock anterior, no actualizamos
    }
}
```

3. EJEMPLO

Bean Pedido: es receptor de los cambios del stock de Producto

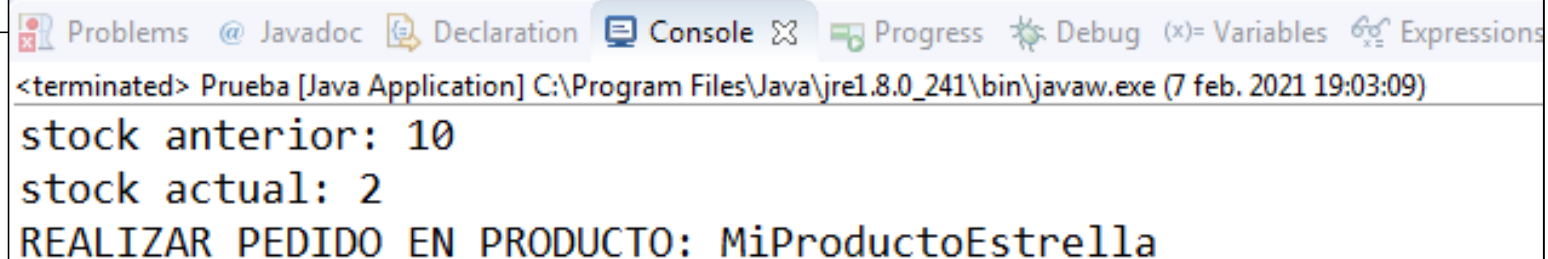
```
public class Pedido implements Serializable, PropertyChangeListener {
    private int numeroPedido;
    private Producto producto;
    private Date fecha;
    private int cantidad;

    public Pedido() { }
    public Pedido (int numeroPedido, Producto producto, Date fecha, int cantidad) {
        this.numeroPedido = numeroPedido;
        this.producto = producto;
        this.fecha = fecha;
        this.cantidad = cantidad;
    }
    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        System.out.printf("stock anterior: %d\n",evt.getOldValue());
        System.out.printf("stock actual: %d\n",evt.getNewValue());
        System.out.printf("REALIZAR PEDIDO EN PRODUCTO: %s\n",producto.getDescripcion());
    }
}
```

3. EJEMPLO

En el programa principal se agrega Pedido como PropertyChangeListener de Producto, es decir el Javabean que será avisado cuando se dispare el evento por falta de stock:

```
public static void main(String[] args) {  
    Producto producto = new Producto(1, "MiProductoEstrella",10,3,16);  
    Pedido pedido = new Pedido();  
    pedido.setProducto(producto);  
  
    producto.addPropertyChangeListener (pedido);  
    producto.setStockactual(2);  
}
```



Problems Javadoc Declaration Console Progress Debug (x)= Variables Expressions

<terminated> Prueba [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe (7 feb. 2021 19:03:09)

stock anterior: 10
stock actual: 2
REALIZAR PEDIDO EN PRODUCTO: MiProductoEstrella

4. PRACTICA 1

Paso 1. Crea un JavaBean Empleado que representa el Empleado de una compañía.

- Cuatro propiedades: NIF, nombre, cargo que ocupa y sueldo
- Dos constructores:
 - Empleado(): Otorga a los campos cargo y sueldo los valores de “Junior” y 1000€ respectivamente
 - Empleado(NIF, nombre) : llama al constructor anterior y otorga los valores de NIF y nombre pasados como argumentos

```
public class Empleado implements Serializable {  
    private String NIF;  
    private String nombre;  
    private String cargo;  
    private float sueldo;  
    private PropertyChangeSupport propertySupport;  
  
    public Empleado() {  
        this.propertySupport = new PropertyChangeSupport(this);  
        this.cargo="Junior";  
        this.sueldo=1000;  
    }  
    public Empleado(String NIF, String nombre) {  
        this.propertySupport = new PropertyChangeSupport(this);  
        this.NIF=NIF;  
        this.nombre=nombre;  
    }  
}
```

4. PRACTICA 1

Paso 2. Cuando se modifica el valor de “cargo” del Bean se comprueba que la modificación realizada no ponga el atributo a “NULL” o lo deje en blanco. En caso de no ser ni lo uno ni lo otro, se notifica el cambio de la propiedad a un Bean oyente. De la misma manera, cuando se modifica el valor de “sueldo”, el Bean Empleado comprueba que la modificación realizada sea superior a 0. En caso de serlo, se notifica el cambio de la propiedad al Bean oyente

```
public void setCargo(String cargo) {  
    if(cargo != null && !cargo.equals("")) {  
        try{  
            propertySupport.firePropertyChange("cargo", this.cargo, cargo);  
            this.cargo = cargo;  
        }catch (ArithmeticException e){  
            e.printStackTrace();  
        }  
    }  
}
```

```
public void setSueldo(float sueldo) {  
    if(sueldo >= 0){  
        try {  
            propertySupport.firePropertyChange("sueldo", this.sueldo, sueldo);  
            this.sueldo = sueldo;  
        }catch (ArithmeticException e){  
            e.printStackTrace();  
        }  
    }  
}
```

4. PRACTICA 1

Paso 3. Crea el Bean oyente PanelEmpleado. Es un Bean que puede leer las propiedades de sueldo y cargo del Bean Empleado. Dispone de:

- Dos atributos:
 - limiteVariacionSueldo: representa un límite de la variación del sueldo de un empleado ya sea un incremento o un decremento. Se trata de un valor entero entre 10 y 50.
 - listaDeCargos: representa una lista de cargos asignables al trabajador. Se trata de un vector de Strings o similar. Deben existir todos los getters y setters necesarios según hemos visto en teoría. Por defecto, se inicializa con los valores: (“Junior”, “SemiSenior”, “Analista”, “CEO”)

4. PRACTICA 1

Paso 4. El Bean PanelEmpleado dispone de dos constructores:

- PanelEmpleado(): Otorga al atributo limiteVariacionSueldo el valor de 10
- PanelEmpleado(int valor) : otorga al atributo limiteVariacionSueldo el valor indicado en el argumento.

```
public class PanelEmpleado implements Serializable, PropertyChangeListener {  
  
    private float limiteVariacionSueldo;  
    private ArrayList<String> listaDeCargos = new ArrayList<String>();  
  
    public PanelEmpleado() {  
        this.limiteVariacionSueldo=10;  
    }  
    public PanelEmpleado(float limiteVariacionSueldo) {  
        this.limiteVariacionSueldo = limiteVariacionSueldo;  
    }  
}
```

4. PRACTICA 1

Paso 5. Si el Bean PanelEmpleado recibe la notificación de una actualización en el valor del sueldo de un empleado:

- Calculará el porcentaje que varía el sueldo actual respecto al anterior.
- Si el porcentaje es superior al valor de limiteVariacionSueldo, PanelEmpleado generará una excepción que pueda ser capturada y mostrada por el Bean Empleado, mostrando un mensaje que detalle el error.
- A su vez, el Bean Empleado no podrá modificar el sueldo

Paso 6. Si el Bean PanelEmpleado recibe una notificación de que el atributo cargo de Empleado ha sido modificado:

- Comprobará que el nuevo cargo se encuentre dentro de la listaDeCargos.
- Si no está en la lista, generará una excepción que será capturada por el Bean Empleado mostrando un mensaje de error
- Si sucede esto, el cargo del empleado no podrá ser modificado

4. PRACTICA 1

```
@Override
public void propertyChange(PropertyChangeEvent propertyChangeEvent){
    System.out.println(propertyChangeEvent.getPropertyName());
    System.out.println(" old value " + propertyChangeEvent.getOldValue());
    System.out.println(" new value" + propertyChangeEvent.getNewValue());
    System.out.println(" getLimiteVariacionSueldo() " + getLimiteVariacionSueldo());
    if(propertyChangeEvent.getPropertyName().equals("sueldo")){

        if(){
            throw new ArithmeticException("El saldo esta fuera de rango ");
        }
    }else if(propertyChangeEvent.getPropertyName().equals("cargo")) {

        if() {
            throw new ArithmeticException("El cargo seleccionado no se encuentra disponible ");
        }
    }
}
```

4. PRACTICA 1

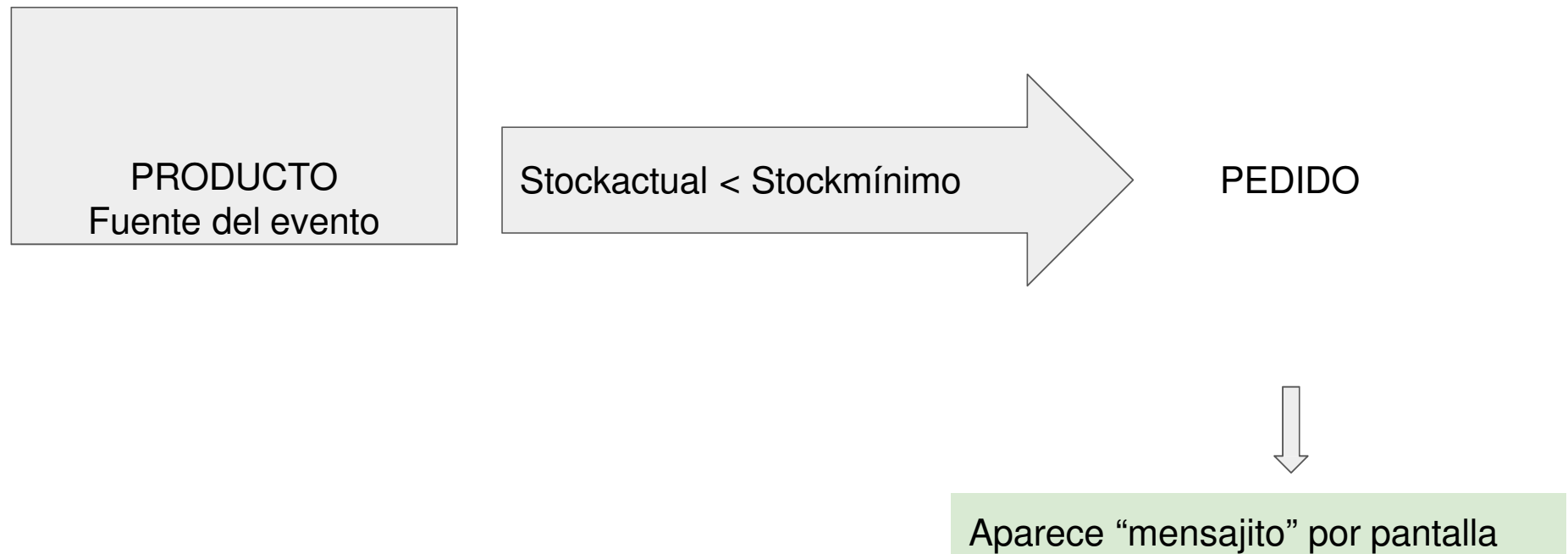
Paso 7. Finalmente, crea una clase llamada PruebasFinales que contendrá el main. Define un objeto Empleado y un objeto PanelEmpleado. Vincula ambos objetos y realiza un pequeño juego de pruebas para comprobar que ambos componentes funcionan correctamente.

```
public static void main(String[] args) {  
    Empleado e = new Empleado();  
    PanelEmpleado pe = new PanelEmpleado(50);  
    e.addPropertyChangeListener(pe);  
  
    System.out.println("Cambiando el sueldo a algo invalido 1501");  
    e.setSueldo(1501);  
  
    System.out.println("Cambiando el sueldo a algo invalido 499");  
    e.setSueldo(499);  
}
```

5. PRACTICA 2

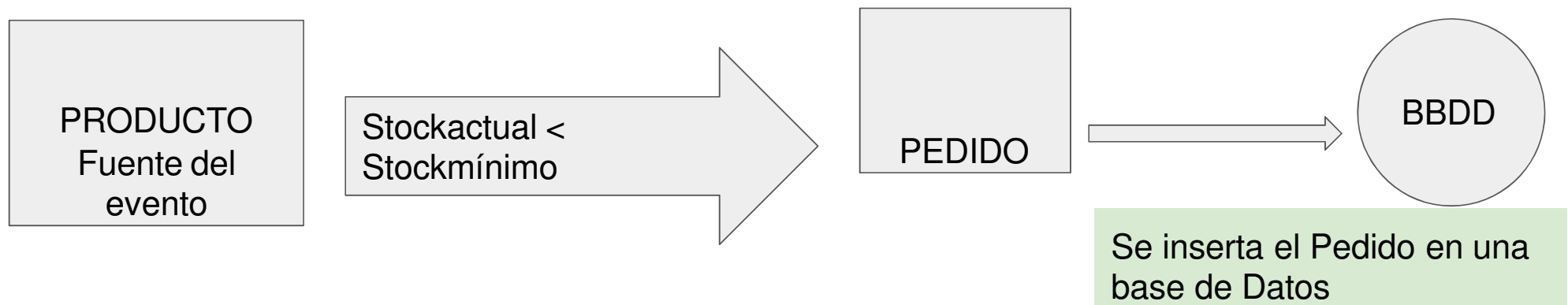
- Vamos a montar una aplicación por componentes utilizando JavaBeans.
- Esta aplicación nos servirá para introducir el acceso a datos desde un JavaBean
- Como punto de partida tomaremos el ejemplo de la sesión anterior donde teníamos dos tipos de objetos: Pedido y Producto.
- Producto era el **JavaBean Fuente de Eventos** y Pedido era el **JavaBean Receptor de Eventos**.
- Cuando se modificaba un el stock de un producto (producto.setStockactual) se generaba un evento. Este evento era capturado por un pedido y aparecía un mensajito.

5. PRACTICA 2



5. PRACTICA 2

- Vamos a modificar el *propertyChange* de Pedido para que en vez de generar un “mensajito” inserte el pedido en una base de datos.
- Utilizaremos la base de datos de objetos *Neodatis*.



5. PRACTICA 2

- Se ha de crear una clase que contenga todos los métodos que permitan abrir la base de datos, cerrar, guardar ventas, pedidos y productos en la base de datos.
- En concreto, tendrá:
 - Propiedad de tipo `org.neodatis.odb.ODB`
 - Un constructor que abrirá la base de datos: `this.odb = ODBFactory.open("Producto_Ped.BD");`
 - Un método para cerrar la base de datos
 - Un método para insertar un producto
 - Un método para insertar un pedido
 - Un método para insertar una venta.

5. PRACTICA 2

1. Genera el código de la clase “BaseDatos”. Debe contener todos los métodos mencionados. Trabaja con *Neodatis*.
2. Genera un programa que llamarás “LlenarProductos”. Insertará productos en la base de datos. Utilizará los métodos de la clase “BaseDatos”. Los productos a insertar son estos:

Producto p1 = new Producto(1, "Duruss Cobalt", 10, 3, 220);

Producto p2 = new Producto(2, "Varlion Avant Carbon", 5, 2, 176);

Producto p3 = new Producto(3, "Star Vie Pyramid R50", 20, 5, 193);

Producto p4 = new Producto(4, "Dunlop Titan", 8, 3, 85);

Producto p5 = new Producto(5, "Vision King", 7, 1, 159);

Producto p6 = new Producto(6, "Slazenger Reflex Pro", 5, 2, 80);

5. PRACTICA 2

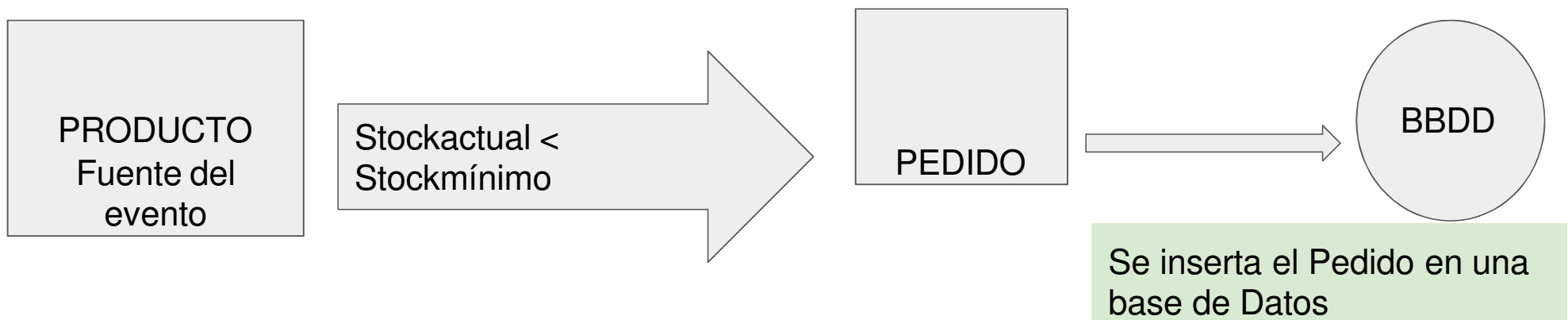
3. Genera un programa que llamarás “VerProductos” que permita visualizar los productos y sus características de la base de datos.
4. Modifica el *propertyChange* de la clase Pedido para que, cuando se produzca un evento desde Producto - es decir, cuando Stockactual supere el Stockmínimo - se inserte un pedido en la base de datos. Se debe considerar:
 - a. El número del pedido (propiedad *numeroPedido*) a insertar en la base de datos se debe de generar. Para ello, se utilizará el número de Pedido más alto más uno de entre los Pedidos que ya están almacenados en la base de datos (Es decir, reproducir un “auto_increment” mediante programación).
 - b. El campo fecha del pedido debe ser la fecha actual.
5. Modifica el programa “VerProductos” para que permita visualizar los Pedidos además de los productos.

5. PRACTICA 2

6. Escribe un programa que testee todos los componentes. Deberá de:
 - a. Mostrar productos
 - b. Mostrar pedidos
 - c. Disparar la generación de un pedido y su posterior almacenamiento en la base de datos. Para ello, lógicamente, modificará el Stockactual de algún producto de tal forma que quede por debajo de su Stockmínimo.
 - d. Volver a mostrar los productos. NOTA: el Stockactual del producto que ha provocado la generación del pedido NO se modificará. Si, en cambio, la modificación del Stockactual no provoca la generación del pedido (es decir, el Stockactual modificado no queda por debajo del Stockmínimo del producto) entonces sí que debe modificarse el Stockactual
($Stockactual = Stockactual - cantidad$)
 - e. Volver a mostrar los pedidos.

6. PRACTICA 3

En esta sesión, vamos a completar la aplicación que iniciamos en la anterior. La idea es generar un nuevo componente que llamaremos Venta. También modificaremos algunos de los componentes que ya tenemos para que se comuniquen adecuadamente con los objetos de tipo Venta. Teníamos:



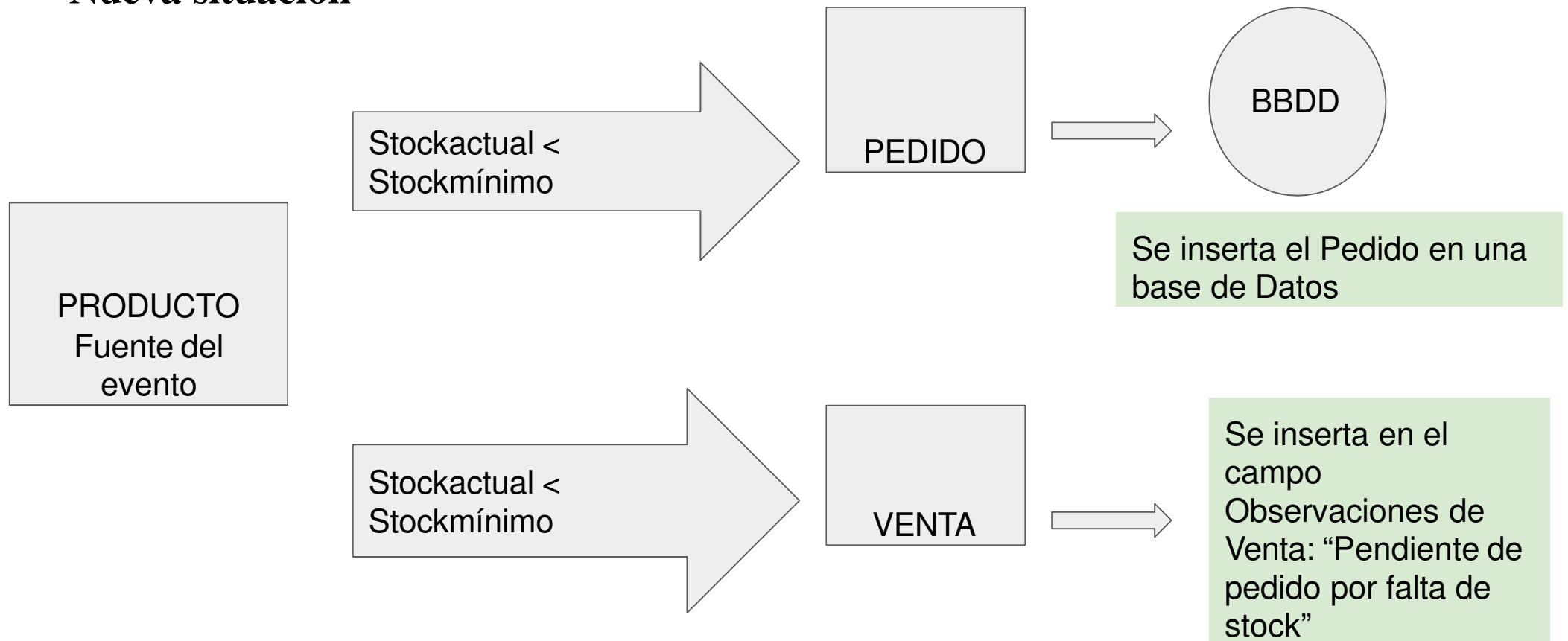
6. PRACTICA 3

Venta

- **Este objeto implementará SERIALIZABLE y PROPERTYCHANGELISTENER**
- **Tendrá las siguientes propiedades:**
 - NumeroVenta → Identificador de la venta
 - IdProducto → Identificador del producto que se vende
 - FechaVenta → Fecha de la venta
 - Cantidad → Cantidad de unidades
 - Observaciones → Observaciones de la venta
- **Tendrá un método PROPERTYCHANGE. Este método se activará cuando el stock actual del producto supere el stock mínimo. Cuando se active, cambiará el campo “*Observaciones*” igualándolo a: “Pendiente de pedido por falta de stock”
Es decir, será un oyente de Producto.**

6. PRACTICA 3

Nueva situación



6. PRACTICA 3

- Implementa la clase *Venta* cumpliendo con los requisitos que impone *JavaBean*.
- Modifica la clase “*BaseDatos*” para añadir un método que permita insertar ventas. Se debe tener en cuenta:
 - El número de la venta (*NumeroVenta*) se obtiene sumándole uno al número de venta más alto de las ventas ya insertadas en la base de datos.
 - El campo *Observaciones* se puede dejar inicialmente en blanco.
 - Además, al insertar una venta en la base de datos, debemos modificar el *stockActual* del producto vendido. En el caso de que $\text{StockActual} < \text{StockMínimo}$ se producen los siguientes “eventos”:
 - Se genera un pedido del producto solicitado (esta parte ya la hicimos en la sesión anterior)
 - Se modifica el campo *Observaciones* del objeto *Venta* insertando “Pendiente de pedido por falta de stock”

6. PRACTICA 3

- Generar el programa que utilizará todos los componentes. El programa solicitará un identificador de producto y una cantidad de unidades de dicho producto al usuario. A continuación:
 - Si el producto no se encuentra en la base de datos (en la tienda) devolverá un mensaje informando
 - Si el producto sí se encuentra, tratará de almacenar la venta del mismo en la base de datos.
 - Al realizar la venta, el número de unidades disponibles de dicho producto (stockActual) se reducirá.
 - Si, al reducirse, el stockActual de un producto es menor que el stockMínimo de dicho producto debe generarse:
 - Un pedido que se almacenará en la base de datos.
 - Una inserción en el campo *Observaciones* de Venta indicando que la venta queda pendiente de realizar el pedido.
 - Se informará al usuario de la circunstancia

7. CREAR COMPONENTE

Objetivo

- Vamos a crear un componente EJB: *Enterprise JavaBeans*
- Para ello utilizaremos los programas que ya tenemos y los “reconvertiremos” en un componente EJB
- Después, utilizaremos este componente para realizar nuestras aplicaciones.

7. CREAR COMPONENTE

Proyecto EJB

1. En eclipse crearemos un nuevo proyecto según la plantilla EJB. Llamaré a este proyecto “MisBeans”
2. Observa que Eclipse ha añadido un archivo de texto llamado MANIFEST.MF
3. Traslamos nuestros componentes (los que hicimos la última sesión) al proyecto: “Pedido.java”, “Producto.java”, “BaseDatos.java”, “Venta.java”
4. Hay que modificar el archivo MANIFEST.MF. Para ello, simplemente añadimos por cada componente las siguientes líneas:

```
Name: <NombreProyecto>/<componente>.class  
Java-Bean: True
```

NOTA: El componente BaseDatos.class NO es un Bean puesto que es nuestra interfaz de acceso a datos.

7. CREAR COMPONENTE

Componente EJB

- Una vez tenemos todo listo, tenemos que exportar el componente como EJB. Esto comportará la generación de un archivo “<MisBeans>.jar”
- Este archivo (MisBeans.jar) será nuestro componente EJB que utilizaremos como prácticamente cualquiera de los paquetes .jar que venimos utilizando a lo largo del curso (por ejemplo, para acceder a NeoDatis utilizamos las librerías externas neodatis.jar)

7. CREAR COMPONENTE

Comprobando nuestro EJB

- Crea un nuevo proyecto java en eclipse.
- Agrega el componente recién creado MisBeans.jar
- Agrega también las librerías de acceso a la base de datos NeoDatis
- Copia en este proyecto el programa de la sesión anterior que llamábamos “LlenarVentas.java”
- Realiza las modificaciones pertinentes para que LlenarVentas utilice nuestro EJB (habrá que importar las clases)
- Ejecuta y comprueba que todas las funcionalidades operan correctamente.

8. PATRON DAO

- El patrón DAO nos permite acceder a datos que pueden estar localizados en distintas fuentes.
- La idea se basa en tres elementos:
 - **Objeto DAO:** es un simple POJO (*Plain Old Java Object*). Contiene la estructura de nuestros datos.
 - **Interfaz:** define todas las operaciones que se pueden realizar con un objeto DAO: insertar, modificar, eliminar, consultar, etc. En esta interfaz no habrá ninguna referencia a ninguna base de datos del tipo que sea.
 - **Clase que implemente la interfaz:** será la clase responsable de obtener los datos de un origen (base de datos, fichero ...)

8. PATRON DAO

Ejemplo de aplicación

- Vamos a desarrollar los 3 elementos: objetoDAO, interfaz y una clase que implemente la interfaz
- Utilizaremos la base de datos Neodatis como fuente de datos
- Haremos una aplicación por componentes siguiendo el patrón DAO para gestionar departamentos.
- Los tres componentes del patrón formarán un paquete que podemos llamar Dep.

8. PATRON DAO

ObjetoDAO

- Vamos a crear una clase pública que llamaremos **Departamento** (Departamento.java)
- Implementará la interfaz **Serializable** que nos permitirá almacenar el estado de un objeto Departamento en un momento dado.
- Contendrá tres atributos:
 - int deptno → Número identificativo del departamento
 - String dnombre → Nombre del departamento
 - String loc → Nombre de la localidad donde está ubicado el departamento
- Tendrá dos constructores: Departamento() y Departamento(int deptno, String dnombre, String loc)
- Tendrá todos los “setters” y “getters” necesarios

8. PATRON DAO

Interfaz

- Añadimos a nuestra librería la interfaz DepartamentoDAO.java. La nueva interfaz tendrá las siguientes operaciones:
 - **public boolean InsertarDep (Departamento dep)** → recibe un objeto Departamento para insertarlo en la fuente de datos que sea. Devuelve *true* si la operación se ha realizado correctamente, en caso contrario devuelve *false*
 - **public boolean EliminarDep (int deptno)** → elimina el departamento de la fuente de datos. Recibe el número de departamento a eliminar. Devuelve *true* o *false* según el resultado de la operación
 - **public boolean ModificarDep (int deptno, Departamento dep)** → modifica el departamento indicado en deptno, los datos a modificar están en el objeto dep.
 - **public Departamento ConsultarDep (int deptno)** → recibe un número de departamento y devuelve el objeto departamento cuyo número coincida con deptno

8. PATRON DAO

Implementación de DepartamentoDAO

- Añadimos la librería JAR de Neodatis.
- Llamaremos a la clase **DepartamentoImpl** e implementará la interfaz DepartamentoDAO
- Tendrá una propiedad estática: *static ODB bd;*
- El constructor de la clase abrirá la base de datos (“Departamento.BD”)
- En esta clase se sobrescriben todos los métodos definidos en la interfaz anterior para la base de datos “Departamento.BD” gestionada por Neodatis. También se añade un método para obtener la conexión a la BD que previamente crea el constructor.

8. PATRON DAO

Clase que probará el DAO

- Una vez tenemos las 3 clases, generamos el JAR.
- Ahora, creamos en un nuevo proyecto (“dao_prueba”) añadimos el JAR creado y el JAR de Neodatis.
- Añadimos al nuevo proyecto una nueva clase que tendrá más o menos la siguiente estructura:

```
/* Empezará con una línea como la que sigue: */  
DepartamentoDAO depDAO = new DepartamentoImpl( );  
  
/* 1-- Inserta un nuevo departamento */  
/* 2-- Consulta el nuevo departamento */  
/* 3-- Modifica algunos valores del nuevo departamento*/  
/* 4-- Elimina el departamento creado */
```


9. USOS DE JAVABEAN 1

Introducción

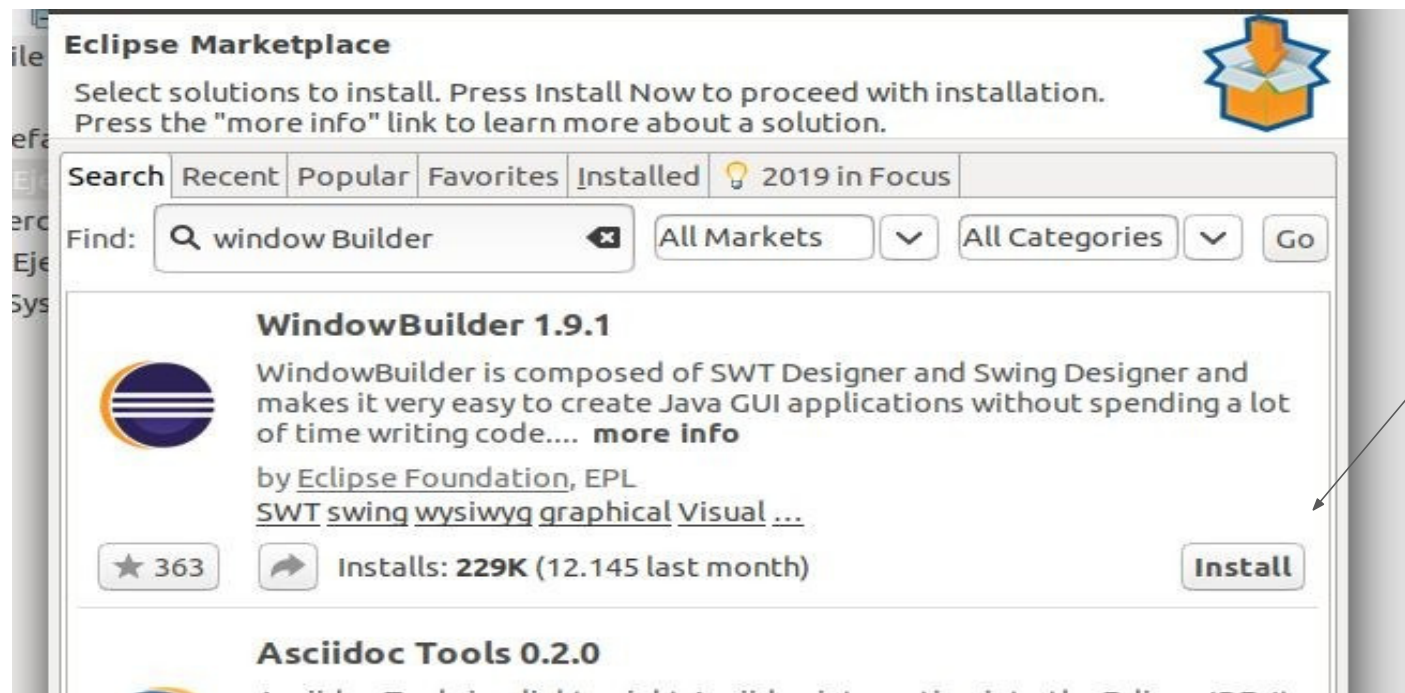
- Los javabeans son normalmente utilizados en aplicaciones de escritorio o en aplicaciones web
- En esta presentación voy a intentar explicar cómo usar los javabeans en aplicaciones de escritorio.
- Para crear aplicaciones de escritorio, necesitamos java.swing. Para trabajar con *swing* de forma “fácil” utilizaremos un plugin de Eclipse →

WindowBuilder

9. USOS DE JAVABEAN 1

Window Builder

- Window Builder es un plugin de Eclipse que nos permitirá manejar clases de java.swing de forma visual.
- Instalación → Desde MarketPlace de Eclipse. → **WindowBuilder 1.9.1**



9. USOS DE JAVABEAN 1

Proyecto Con Window Builder

- Creamos un nuevo proyecto
- Añadimos **New** → **Other** → **WindowBuilder** → **Swing Designer** → **JFrame**

```
⊕ import java.awt.BorderLayout;

public class UsoBeanCalendar extends JFrame {

    private JPanel contentPane;

    /**
     * Launch the application.
     */
    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
```

- Observad que hay dos pestañas: *Source* / *Design*

9. USOS DE JAVABEAN 1

Proyecto con Window Builder

- Si vamos a la pestaña Designer podemos añadir componentes. Por ejemplo, en la “Palette” → Components → JFrame
- Pero también podemos añadir **JavaBeans** propios. Por ejemplo, nos podemos bajar: JCalendar.jar → [JCalendar](#)
- JCalendar es un Bean diseñado por Kai Tödter. Vamos a añadirlo a nuestro proyecto.
- Lo primero será descargarlo desde [aquí](#). Descomprimos. En la carpeta *lib* tendremos el Bean que buscamos.

9. USOS DE JAVABEAN 1

Jcalendar

- Pestaña Designer → Palette → (Botón drch) → Add Category → Añadimos una categoría nueva que llamaremos “Propia”
- Botón drch sobre la nueva categoría → Import .jar → Elegimos la opción *FileSystem* → Seleccionamos todos los componentes del JCalendar_<version>.jar que nos habíamos descargado previamente.
- Ahora, en la paleta “Propia” debemos tener todos los componentes de JCalendar que podemos añadir fácilmente en nuestro proyecto.

9. USOS DE JAVABEAN 1

Conclusión

- En aplicaciones de escritorio, los JavaBeans nos permiten programar por componentes agregándolos fácilmente a nuestro proyecto.
- Existen más utilidades o “usos” de los JavaBeans → MVC en aplicaciones web
- En el MVC la parte de “Modelo” se suele implementar con JavaBeans.

10. USOS DE JAVABEAN 2

Introducción

- Vamos a ver una nueva utilidad de los JavaBean: aplicaciones web siguiendo el MVC
- ¿Qué necesitamos saber?
 - JSP
 - Servlets
 - MVC
- Todo lo iremos viendo poco a poco.
- Primero → preparar nuestro entorno para trabajar con este tipo de aplicaciones.

10. USOS DE JAVABEAN 2

Preparando Eclipse

- Help → Available Software → Del desplegable optamos por “--All available Sites --”
- Seleccionamos:



- Tardará un buen rato en calcular dependencias... → Aceptamos licencias y “Finish”
- Habrá que reiniciar Eclipse.

10. USOS DE JAVABEAN 2

Instalando Tomcat

- Acudimos a <http://tomcat.apache.org/>
- Nos descargamos la versión “Core” 8.5.40
- Activamos la perspectiva “Servers”.
-

10. USOS DE JAVABEAN 2

Primeros pasos

- Creamos proyecto del tipo “Dynamic Web Project”. Llamaremos al proyecto “WebApp”. Pediremos que Eclipse genere el “descriptor” del proyecto: **web.xml**
- Añadimos una página html en **New** → **HTML**.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"><html><head><META  
http-equiv="Content-Type" content="text/html; charset=utf-8"></head>  
<body>  
<h1 align="center">Gestión</h1>  
<p align="center">  
    <a href="http://WebApp/Controlador?accion=listado" target="_blank"> Listado </a>  
</p>  
</body></html>
```

- Ejecuta y observa el resultado.

10. USOS DE JAVABEAN 2

Añadimos Servlet

- Creamos un nuevo paquete en “Source” que llamaremos “control”
- Añadimos **New** → **Servlet**. Reescribimos el método **doGet**:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    String op = request.getParameter("accion");
    if (op.equals("alta")) {
        System.out.println("depurando");
        response.sendRedirect("alta.jsp");
    }
}
```

- Estamos reenviando a “alta.jsp”

10. USOS DE JAVABEAN 2

Web.xml

- Añadimos las siguientes líneas al descriptor del proyecto:

```
<servlet>
  <servlet-name>Controlador</servlet-name>
  <servlet-class>Controlador</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Controlador</servlet>
  <url-pattern>/Controlador</servlet>
```

10. USOS DE JAVABEAN 2

JSP

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%><!DOCTYPE html>
<html><head><meta charset="UTF-8">
<title>VayaTela</title>
</head>
<jsp:useBean id="depart" scope="request" class="miWeb.Departamentos"></jsp:useBean>
<jsp:setProperty property="*" name="depart"/>
<%
if(request.getParameter("deptno")!=null){%>
<jsp:forward page="/Controlador?accion=insertar"></jsp:forward>
<% } %>
<body>
<center><h2>Entrada </h2>
<form method="post">
<p> Num departamento:
<input name="deptno" required type="number" min="1" max="99" /> </p>
<p> Nombre:
<input name="dnombre" required type="text" size="15" maxlength="15" /> </p>
<p> localidad:
<input name="loc" required type="text" size="15" maxlength="15"/> </p>
<input type="submit" name="insertar" value="insertar" />
</form>
<a href="index.html"> Inicio</a>
</center>
</body>
</html>
```

10. USOS DE JAVABEAN 2

Modelo

- Falta añadir nuestro modelo de datos. En concreto, necesitamos un Bean para representar a la clase “Departamento”.

```
public class Departamentos {  
    private int deptno;  
    private String dnombre;  
    private String loc;  
  
    String modificar;  
    String eliminar;  
    String insertar;  
    .... Constructures y getters/setters ....  
}
```

10. USOS DE JAVABEAN 2

Acceso a datos

- Utilizaremos la base de datos NeoDatis. Hay que añadir los drivers a las rutas del proyecto. Debe poder acceder también el servidor Tomcat
- Falta añadir en el controlador el acceso a la bbdd para guardar datos. Habrá que modificar el doGet del Controlador. La modificación empezará de la siguiente manera:

```
if (op.equals("insertar")) {  
    control.Departamentos dep = (control.Departamentos) request.getAttribute("depart");  
    .....
```