Tecnología de componentes JavaBeans

"Programación orientada a objetos :
Lenguajes, Tecnologías y Herramientas"
Master de Computación

Patricia López
Grupo de Computadores y Tiempo Real

Santander, 2009

Introducción a JavaBeans

- Primer modelo de componentes de Java
- Definición:

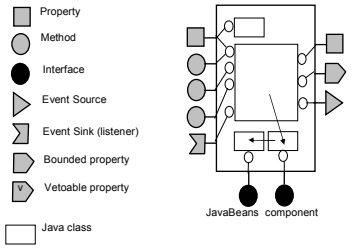
  "*A Java Bean is a reusable software component that can be manipulated visually in a builder tool*"

- Un JavaBean es una clase puramente Java desarrollada con unos patrones de diseño bien definidos, que:
  - Permiten que sea usada en posteriores aplicaciones
  - Permiten gestionar los componentes de forma automática
- Es un modelo sencillo, soportado directamente por el entorno Java => Multiplataforma (aunque no multilenguaje)
- Ejemplos de JavaBeans: Librerías gráficas AWT (Sun), y SWT (Eclipse)

Modelo de componentes JavaBeans

- JavaBeans representa una implementación del módelo Propiedad-Evento-Método

- Un componente JavaBean se define a través de :
  - Las propiedades que expone
  - Los métodos que ofrece
  - Los eventos que atiende o genera



- Para gestionar estas características, todo JavaBean debe ofrecer:
  - Soporte para "Introspection": El bean tiene que ofrecer la información necesaria para que la herramienta de diseño pueda analizar sus características de forma opaca.
  - Soporte para "Customization": La herramienta de construcción de la aplicación puede adaptar ("customizar") la apariencia o comportamiento del bean a la aplicación.
  - Soporte para Persistencia: El estado de un bean customizado puede ser almacenado para ser utilizado más tarde.
  - Soporte para Eventos: Los beans se comunican a través del envío y recepción de eventos.

Patrón de diseño básico de un JavaBean

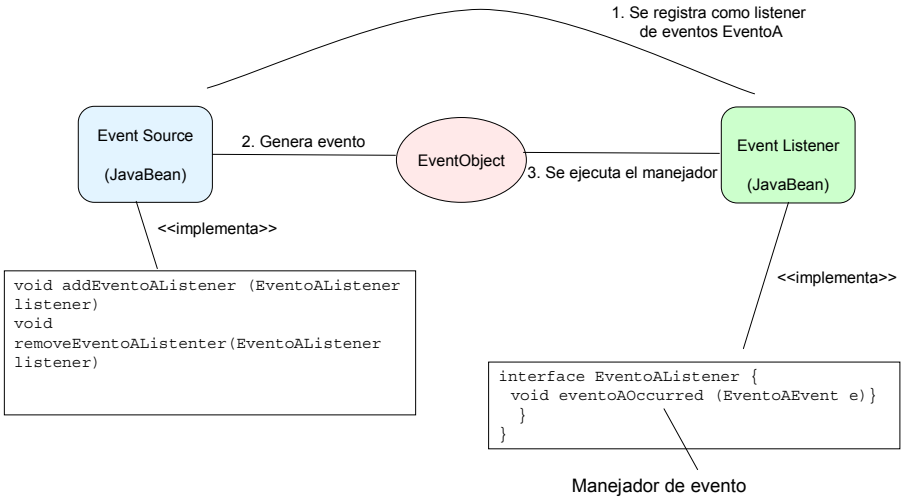
- Reglas de diseño básicas para la construcción de un JavaBean:
  - Clases públicas
  - Constructor vacío por defecto (puede tener más)
  - Implementación de la interfaz Serializable (para poder implementar persistencia)
  - Seguir las convenciones de nombres establecidas (se ven a continuación)

```
public class MiPrimerBean implements Serializable {

   String miProp;
   public MiPrimerBean(){
      miProp = "";
   }

   public void myMethod(){
      System.out.println("Ejemplo sencillo de JavaBean")
   }

   public void setMiProp(String s){
     miProp = s
   }

   public String getMiProp (){
     return miProp;
   }

}
```

Propiedades

- Son atributos con nombre que definen el estado y el comportamiento del componente.
- Patrón de diseño:
  public void set<PropertyName> (<PropertyType> value);
  public <PropertyType> get<PropertyName>();
- Si la propiedad es booleana, el metodo de acceso es:
  public boolean is<PropertyName>();
- Estos métodos constituyen el único modo de acceso a las propiedades.
- Tipos de propiedades:
  - Simples : con un valor único
  - Indexed: Representa arrays de valores. En este caso los métodos de acceso son los siguientes:
    public void set<PropertyName> (<PropertyType>[] value);
    public <PropertyType>[] get<PropertyName>();
    public void set<PropertyName> (<PropertyType> value, int index);
    public <PropertyType> get<PropertyName>(int index);

Modelo de eventos Java



Event Objects

- Es el único parámetro que reciben los manejadores de eventos.
- Encapsulan toda la información asociada a la ocurrencia de un evento. Entre ella el objeto que lanza el evento.
- Todos heredan de la clase *java.util.EventObject*.
- Por cada tipo de evento que exista en la aplicación definiremos una subclase denominada <NombreEvento>Event.
- Ejemplo : Evento que notifica un cambio de temperatura

```
public class TempChangedEvent extends java.util.EventObject
{
    // the new temperature value
    protected double theTemperature;
    // constructor
    public TempChangedEvent(Object source, double temperature)
    {
        super(source);
        // save the new temperature
        theTemperature = temperature;
     }
     // get the temperature value
     public double getTemperature()
     {
        return theTemperature;
     }
```


EventListeners

- Son los objetos que requieren notificación de la ocurrencia de eventos.
- La notificación del evento se produce por invocación de métodos manejadores en el objeto "listener".
- Los métodos manejadores se agrupan en interfaces que extienden a la interfaz *java.util.EventListener*.
- Por cada tipo de evento que se quiera manejar, se define una interfaz denominada *<NombreEvento>Listener*
- Los métodos para el manejo de eventos siguen un patrón:

  void <eventOccurenceMethodName> (<EventObjectType> evt);
- Ej: Clase que maneja eventos del tipo TempChangedEvent

```
interface TempChangedListener extends java.util.EventListener {
   // this method is called whenever the ambient temperature changes
   void tempChanged(TempChangedEvent evt);
}

// Ejemplo de objeto "listener" de eventos de tipo TempChangedEvent
class Termomether implements TempChangedListener {
   public void tempChanged(TempChangedEvent evt) {
      ...
   }
}
```

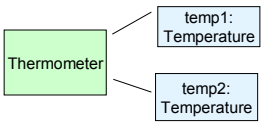
Event Sources

- Son los objetos que generan y lanzan eventos
- Proporcionan métodos para que los objetos "listener" puedan registrarse en ellos y así ser notificados de los eventos que se produzcan.
- Estos métodos siguen el siguiente patrón de diseño:
  public void add<ListenerType>(<ListenerType> listener);
  public void remove<ListenerType>(<ListenerType> listener);
- Toda clase que presente el anterior patrón se identifica como fuente de eventos del tipo correspondiente a *ListenerType*
- Cuando se produce un evento es notificado a todas los listeners que se hayan registrado (Multicast delivery)
- Existe también la posibilidad de notificar el evento en modo unicast:
  public void add<ListenerType>(<ListenerType> listener) throws java.util.TooManyListenersException;
  public void remove<ListenerType>(<ListenerType> listener);
- La invocación de los métodos de notificación en los objetos listener se realiza de forma síncrona

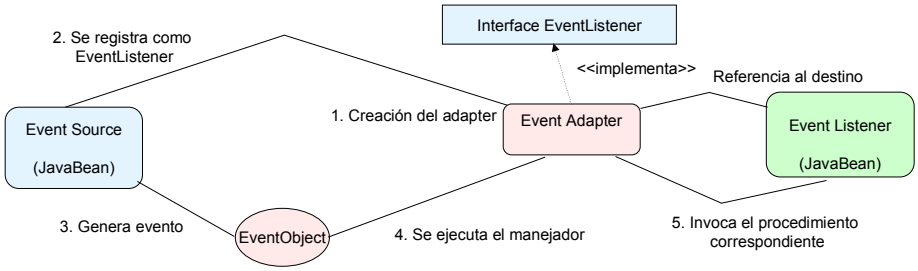
Ejemplo de notificación de evento

```
public class Temperature {
   // the new temperature value
   protected double currentTemp;
   // list of Listeners for temperature changes
   private List<TempChangedListener> listeners = new LinkedList<TempChangedListener>();

   public Temperature() {
      currentTemp = 22.2;
   }

   public synchronized void addTempChangedListener(TempChangedListener listener) {
      listeners.add(listener);
   }

   public synchronized void removeTempChangedListener(TempChangedListener listener) {
      listeners.remove(mcl);
    }
   // notify listening objects of temperature changes
   protected void notifyTempChanged() {
      List<TempChangedListener> l;
      // create the event object
      TempChangedEvent e = new TempChangedEvent(this, currentTemp);

      //Make a copy of the listener object list so that it can not be changed while we are
      // firing events
      synchronized(this) { l = (LinkedList)listeners.clone(); }
      for (int i = 0; i < l.size(); i++) { // deliver it!
         (l.elementAt(i)).TempChanged(e);
     }
```

Event adapters




```
public class Thermometer implements TempChangedListener {
    public tempChanged(TempChangedEvent e) {
        if e.getSource = temp1 {
            temp1changed();
        elsif e.getSource = temp2
            temp2changed();
        }
    }
}
```



Event adapters: Ejemplo

```
public class Thermometer
   // references to the two temperature objects that we are
      monitoring
   protected Temperature theTemperature1;
   protected Temperature theTemperature2;

   // the temperature change adapters
   protected TemperatureAdapter1 tempAdapter1;
   protected TemperatureAdapter2 tempAdapter2;

   // the first adapter class
   class TemperatureAdapter1 implements
       TempChangedListener
   {
      TemperatureAdapter1(Temperature t)
      {
         t.addTempChangedListener(this);
      }
      public void tempChanged(TempChangedEvent evt)
      {
         temperature1Changed(evt.getTemperature());
      }
   }
// the second adapter class
class TemperatureAdapter2 extends TemperatureAdapter
{
   TemperatureAdapter2(Temperature t)
   {
      t.addTempChangedListener(this);
   }
   public void tempChanged(TempChangedEvent evt)
   {
      temperature2Changed(evt.getTemperature());
   }
}

// constructor
Thermometer()
{
   // save references to the temperature objects
   theTemperature1 = new Temperature();
   theTemperature2 = new Temperature();
   // create the adapters
   tempAdapter1 = new TemperatureAdapter1(temp1);
   tempAdapter2 = new TemperatureAdapter2(temp2);
}

// handle changes to Temperature object 1
protected void temperature1Changed(double newTemp)
{
}
// handle changes to Temperature object 2
protected void temperature2Changed(double newTemp)
{
}
}
```

Propiedades Bound y Constrained

- **Bound**: Cuando el valor de la propiedad cambia, se le notifica a otros beans a través de un evento del tipo PropertyChangeEvent.
  - Un bean con propiedades de este tipo debe implementar:
    public void addPropertyChangeListener(PropertyChangeListener x);
    public void removePropertyChangeListener(PropertyChangeListener x);
  - Los listener implementan la interfaz PropertyChangeListener
  - La notificación se realiza cuando el cambio de la propiedad ya se ha realizado

- **Constrained**: Cuando el valor de la propiedad cambia, se le notifica a otros beans que pueden rechazar el cambio.
  - El método set para este tipo de propiedades tiene un patrón especial:
    public void set<PropertyName>(<PropertyType> value)
    throws java.beans.PropertyVetoException;
  - Un bean con propiedades de este tipo debe implementar:
    public void addVetoableChangeListener(VetoableChangeListener p);
    public void removeVetoableChangeListener(VetoableChangelistener p);
  - Los listener tendrán que implementar la interfaz VetoableChangeListener
  - La notificación se realiza antes de que el cambio de la propiedad se haya realizado

Introspection

- Mecanismo para conocer las características de un componente, necesarias para poder manipularlo desde la herramienta de diseño sin acceder a su código:
  - En el caso de un componente JavaBean permite conocer sus propiedades, métodos y eventos.

- Se requieren mecanismos simples, que no obliguen al desarrollador a conocer o escribir código complicado.

- El modelo JavaBeans ofrece dos tipos de introspección:
  - Implícita: A través de la interfaz *java.beans.Introspector*
  - Explícita: A través de clases que implementan la interfaz *java.beans.BeanInfo*, que se añaden y se distribuyen junto a la implementación del JavaBean.

Introspector

- Mecanismo reflector de bajo nivel ofrecido por Java: analiza el código de una clase, extrayendo los métodos que la clase ofrece. A partir de ellos se puede extraer la información acerca de propiedades y eventos
- Se basa en reconocimiento de patrones de diseño:
  - Para propiedades:
    - *public < PropertyType> get< PropertyName>();*
    - *public void set< PropertyName>(< PropertyType> a);*
    - ...
  - Para eventos:
    - *public void add< EventListenerType>(< EventListenerType> a)*
    - *public void remove< EventListenerType>(< EventListenerType> a)*
    - ...
  - Para métodos:
    - Todos los métodos públicos son expuestos
- En JavaBeans los patrones no son obligatorios, pero son necesarios si se quiere utilizar la interfaz Introspector
- Cuando se analiza una clase, se analizan todas las clases superiores en la jerarquía.
- Es el mecanismo que usa el plug-in de Eclipse para desarrollo de interfaces gráficas, VisualEditor

Bean Info

- El desarrollador del componente especifica qué métodos, propiedades y eventos quiere que su Bean exponga a las herramientas de diseño.
- Para ello acompaña a la implementación del Bean con una clase que implemente la interfaz BeanInfo. La clase se denomina siempre *<BeanName>BeanInfo*;
- BeanInfo ofrece métodos para describir los métodos, eventos y propiedades de un JavaBean. Ofrece, entre otros, estos métodos:
  - getBeanDescriptor()
  - getEventSetDescriptors()
  - getPropertyDescriptors()
  - getMethodDescriptors()
- Mejor que implementar directamente toda la interfaz, extender la clase *java.beans.SimpleBeanInfo*
- En este caso no se examinan las clases antecesoras, hay que dar toda la información a través del objeto

Customizers

- Mecanismo que permite al usuario de un componente adaptar su apariencia y comportamiento a la aplicación concreta en la que se vaya a utilizar:
  - Sin necesidad de acceder o escribir ningún código
  - Modificando sus propiedades
- Dos mecanismos:
  - Editor de propiedades incluido en el entorno de desarrollo
    - Vista Properties en el VisualEditor
  - Uso de clases "customizer" (incluidas en la clase BeanInfo)
    - Son ventanas (*wizards*) que van pidiendo los valores de configuración del componente

Persistencia

- Capacidad de almacenar el componente en el estado concreto que tenga en un momento determinado, y poder re-instanciarlo posteriormente en el mismo estado.
- Implementando la interfaz Serializable se puede almacenar el estado de los beans a través de las librerías de Streams que ofrece Java.
- Se deberán guardar aquellas características que permitan reincorporar al bean posteriormente en el mismo estado y con la misma apariencia.
  - Propiedades expuestas
  - Propiedades internas
- Cuando se carga el componente en el programa se hace a partir del fichero binario serializado.

Empaquetamiento

- Una vez desarrollado un JavaBean se empaqueta para su posterior distribución y utilización.
- El empaquetamiento y distribución de JavaBeans se realiza a través de paquetes .jar. Se pueden incluir varios beans en un mismo jar
- En el .jar correspondiente a un bean se incluirán todas las clases que lo forman:
  - El propio bean,
  - Objetos BeanInfo,
  - Objetos Customizer,
  - Clases de utilidad o recursos que requiera el bean, etc
- El .jar debe incluir siempre un manifest-file (fichero .mf) que describa su contenido. Ejemplo:

```
Manifest-Version: 1.0
Name: demo/MyBean.class
Java-Bean: True
Name: demo/MyBeanBeanInfo.class
Java-Bean: False
Name: demo/MyAuxiliaryClass.class
Java-Bean: False
Name: demo/MyImage.png
Java-Bean: False
```