UD6: PROGRAMACIÓN BASADA EN COMPONENTES

Conceptos clave

1. Índice

1 Componentes (.jar)

Una de las posibles definiciones de un componente es: una unidad de software que posee un conjunto de interfaces y funcionalidades. Puede ser integrado en otras aplicaciones favoreciendo la reutilización y la separación de responsabilidades.

1.1 Características

- Identificable: Su nombre debe indicar su funcionalidad
- Autocontenido: No puede requerir otros componentes para llevar a cabo su funcionalidad
- Reemplazable por otro componente
- Acceso solamente a través de su interfaz: No se puede acceder al código fuente
- Sus servicios no varían
- Bien documentado
- Genérico: Debería ser útil para varias aplicaciones
- Reutilizado dinámicamente: Cargado en tiempo de ejecución
- Se distribuye como un paquete

1.2 Ventajas e inconvenientes

✔ Reutilización de software

✔ Disminución de la complejidad de software

✔ Los errores son más fáciles de detectar => Incrementa la calidad del software

✗ Falta de estándares de desarrollo y de certificaciones que garanticen su calidad

2 JAVABEANS (JB)

Un JavaBean es un componente SW reutilizable (una clase Java) que debe seguir un estándar:

- El constructor sin argumentos
- Los atributos de clase deben ser privados
- Cada propiedad debe tener los métodos getter setter respetando la nomenclatura
- Debe implementar Serializable.

2.1 Características de un JavaBean (JB):

- **Introspección**: Mecanismo mediante el que el JB proporciona información sobre sus:propiedades, métodos y eventos. Existen dos formas: patrones de nombrado y la clase BeanInformation (proporciona características)
- **Manejo de eventos:** Método de comunicación entre JB
- **Propiedades**: Determinan la apariencia y comportamiento de un Bean
- **Persistencia** (Serializable)

Los JavaBenas de diseñaron para ser manipulados por herramientas visuales, pero nosotros veremos un ejemplo de uso como componentes de acceso a BD

2.2 Propiedades y atributos

- **Propiedades simples:** Representan un único valor de tipo primitivo o referenciado (objeto)
- **Propiedades indexadas:** Representa una colección, debe poseer getters y setters (ej: array)

2.2.1 Propiedades ligadas/compartidas

Propiedades **asociadas a eventos**, pertenecen a la clase *PropertyChangeSupport* que permite generar eventos no visuales. El BeanFuente cuando sufre cambios en propiedades, notifica a otro BeanOyente. El fuente debe guardar una lista de receptores y alertarlos cuando cambie una propiedad. Para ello, debe proporcionar una propiedad de tipo *PropertyChangeSupport*.
Los receptores de añaden/eliminan con los métodos:

- *addPropertyChangeListener(PropertyChangeListener listener)*
- *removePropertyChangeListener(PropertyChangeListener listener)*

Cuando la propiedad cambia, debe llamarse al método *firePropertyChange()* de *PropertyChangeSupport*. Este método toma 3 parámetros: El valor de la propiedad que se ha cambiado, el valor antiguo y el nuevo valor. Esto mismo se puede ver en la Figura 1

```java
public class BeanFuente implements Serializable {
    /**
     *
     */
    private static final long serialVersionUID = 1L;
    //Esta propiedad permite notificar a otros oyentes de que este componente (this) es la fuente del evento de cambio
    //Se puede hacer aquí como propiedad o en el constructor del bean (sin el modificador final)
    private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);

    public void addPropertyChangeListener(PropertyChangeListener listener) {
        this.pcs.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener listener) {
        this.pcs.removePropertyChangeListener(listener);
    }

    private String value;

    public String getValue() {
        return this.value;
    }

    public void setValue(String newValue) {
        String oldValue = this.value;
        this.value = newValue;
        //Se notifica el cambio de la propiedad
        // 3 parámetros: El valor de la propiedad que se ha cambiado, el valor antiguo y
        // el nuevo valor
        this.pcs.firePropertyChange("value", oldValue, newValue);
    }
```

Figura 1

El Bean receptor (que será notificado de un cambio) debe implementar la interfaz *PropertyChangeListener* y su método *propertyChange(PropertyChangeEvent evt):*

```java
public class BeanReceptor implements Serializable, PropertyChangeListener{

    /**
     *
     */
    private static final long serialVersionUID = 1L;

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

    }


}
```

Figura 2

2.2.2 Propiedades restringidas

Similares a las ligadas, pero en este caso los objetos el BeanOyente tienen derecho de veto sobre los eventos que reciben (los pueden ignorar). Para ello, el BeanFuente debe usar una propiedad de tipo `VetoableChangeSupport` y permite añadir/eliminar oyentes mediante los métodos:

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

Cuando una propiedad cambia, debe notificar a los oyentes con `fireVetoableChange` con los 3 mismos parámetros: El valor de la propiedad que se ha cambiado, el valor antiguo y el nuevo valor.

```
 8 public class VetoableBeanFuente implements Serializable {
 9
10     /**
11     *
12     */
13     private static final long serialVersionUID = 1L;
14     // Esta propiedad permite notificar a otros oyentes de que este componente
15     // (this) es la fuente del evento de cambio
16     // Se puede hacer aquí como propiedad o en el constructor del bean (sin el
17     // modificador final)
18     private final VetoableChangeSupport vcs = new VetoableChangeSupport(this);
19
20     public void addVetoableChangeListener(VetoableChangeListener listener) {
21         this.vcs.addVetoableChangeListener(listener);
22     }
23
24     public void removeVetoableChangeListener(VetoableChangeListener listener) {
25         this.vcs.removeVetoableChangeListener(listener);
26     }
27
28     private String value;
29
30     public String getValue() {
31         return this.value;
32     }
33
34     public void setValue(String newValue) {
35         String oldValue = this.value;
36
37         try {
38             this.vcs.fireVetoableChange("value", oldValue, newValue);
39             this.value = newValue;
40         } catch (PropertyVetoException pve) {
41             pve.getStackTrace();
42         }
43     }
```

Figura 3

```java
public class VetoableBeanReceptor implements Serializable, VetoableChangeListener {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Override
    public void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException {


        // se lanza excepción si no se aprueba el cambio
        if ((evt.getOldValue()!=null) && evt.getNewValue().equals("ABORT") ){
            //En función de unas condiciones, se aprueba el cambio o no. En este caso es si el nuevo valor es igual a la caden
            //ABORT
            System.out.println("Abortando notificación cambio...");
            throw new PropertyVetoException("Mensaje", evt);
        } else {
            System.out.println("Antiguo valor: " + evt.getOldValue() + "=> Nuevo valor: " + evt.getNewValue());
        }

    }
```

Figura 4

En el receptor, dependiendo de las condiciones, se puede vetar el cambio de propiedad lanzando una excepción `PropertyVetoException` impidiendo el cambio y que se notifique al resto de oyentes.

3 Empaquetado

Una vez creado el componente, es necesario empaquetarlo para poder distribuirlo y utilizarlo después. Para ello se crea el paquete jar que contiene todas las clases que forman el componente:

- El propio componente.
- Clases de utilidad o recursos que requiera el componente, etc.

Es posible incluir varios componentes en un mismo jar.

3.1 Empaquetado con Maven

Maven crea por defecto un paquete jar con las fuentes del proyecto. Para ello, botón derecho sobre el proyecto > **Run as > Maven install**

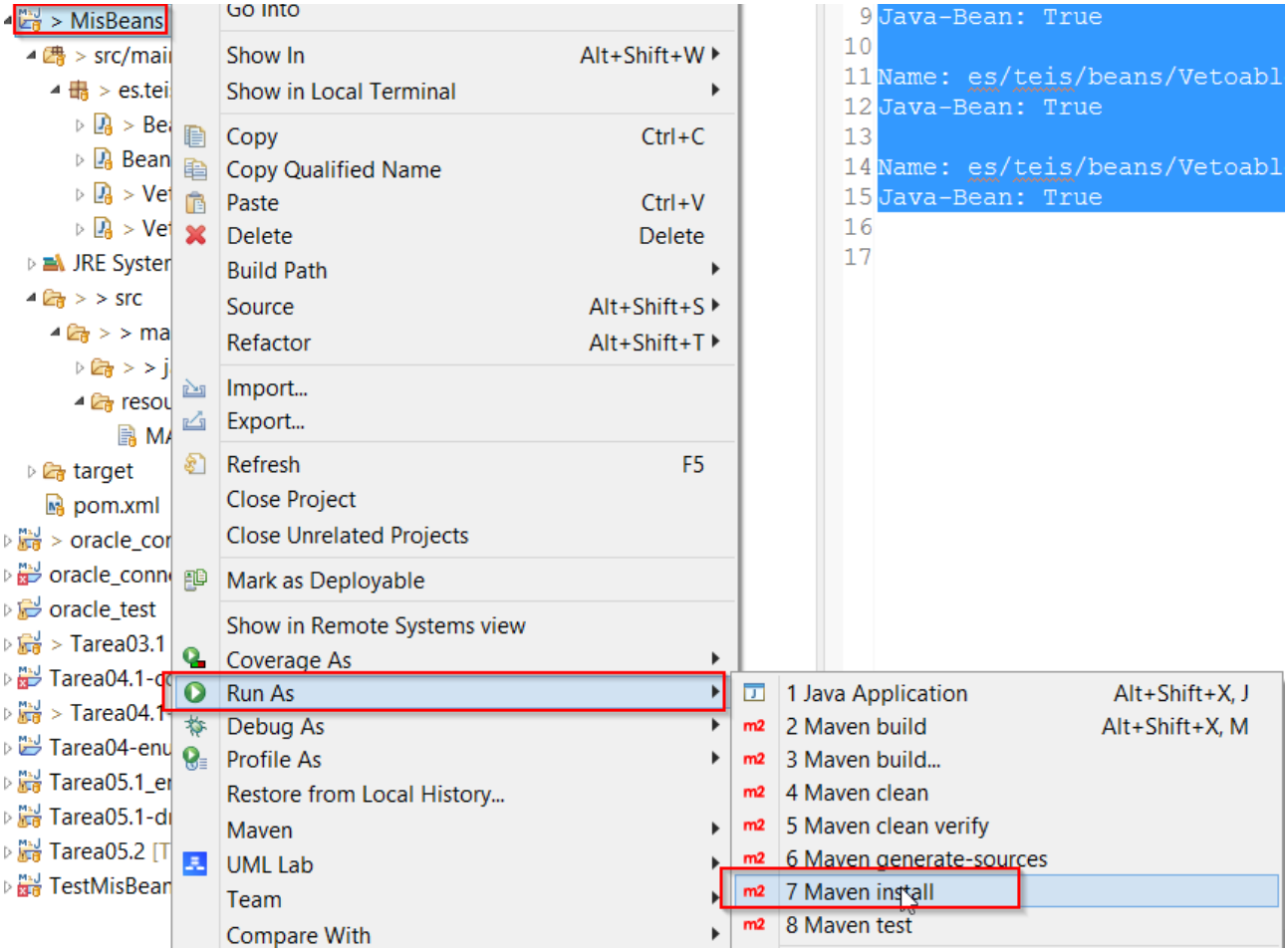

Figura 5

En función de los datos introducidos en el pom.xml, creará el jar y lo moverá al repositorio local (en Windows típicamente en C:\usuarios\nombre_usuario\.m2\...

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.c
    <modelVersion>4.0.0</modelVersion>
    <groupId>es.teis.beans</groupId>
    <artifactId>MisBeans</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>

    </properties>


</project>
```

Figura 6: pom.xml del proyecto MisBeans

```
[INFO] --- maven-install-plugin:2.4:install (default-install) @ MisBeans ---
[INFO] Installing C:\Users\maria\Documents\Curso Drive\mariaferroteis\Curso 22-23\workspace-correcciones\MisBeans\target\MisBeans-1.0-SNAPSHOT.jar to C:\Users\maria\.m2\repository\es
\teis\beans\MisBeans\1.0-SNAPSHOT\MisBeans-1.0-SNAPSHOT.jar
[INFO] Installing C:\Users\maria\Documents\Curso Drive\mariaferroteis\Curso 22-23\workspace-correcciones\MisBeans\pom.xml to C:\Users\maria\.m2\repository\es\teis\beans\MisBeans\1.0-
SNAPSHOT\MisBeans-1.0-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.682 s
[INFO] Finished at: 2023-05-17T09:30:46+02:00
[INFO] ------------------------------------------------------------------------
```

Figura 7: Se habrá creado el jar con el nombre y ruta especificados en la consola

Para reutilizarlo en otro proyecto, en TestMisBeans, se podría añadir de una de estas dos formas:

a) Con Build path > Configure Build path > Add External Jar apuntando al jar generado en el repositorio local

b) la dependencia en el pom.xml y Maven la resolverá contra el repositorio local. Nosotros vamos a hacerlo de esta forma:

```
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.o
3      <modelVersion>4.0.0</modelVersion>
4      <groupId>beans</groupId>
5      <artifactId>TestMisBeans</artifactId>
6      <version>1.0-SNAPSHOT</version>
7      <packaging>jar</packaging>
8      <properties>
9          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
10         <maven.compiler.source>1.8</maven.compiler.source>
11         <maven.compiler.target>1.8</maven.compiler.target>
12     </properties>
13     <dependencies>
14         <dependency>
15             <groupId>es.teis.beans</groupId>
16             <artifactId>MisBeans</artifactId>
17             <version>1.0-SNAPSHOT</version>
18         </dependency>
19     </dependencies>
20 </project>
```

Figura 8

De este modo ya se podrán usar las clases del paquete en el proyecto TestMisBeans:

```java
import es.teis.beans.BeanFuente;
import es.teis.beans.BeanReceptor;
import es.teis.beans.VetoableBeanFuente;
import es.teis.beans.VetoableBeanReceptor;

/**
 *
 * @author mfernandez
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws ClassNotFoundException {
//        AlumnoBean ab = new AlumnoBean();
//        ab.addAlumno();
        testPropertiesVeto();

    }

    private static void testPropertiesLigadas() {
        BeanFuente bf = new BeanFuente();
        BeanReceptor br = new BeanReceptor();

        bf.addPropertyChangeListener(br);

        bf.setValue("new value");

    }
```

Figura 9

3.2 Empaquetado manual

El paquete `jar` debe incluir un fichero de manifiesto (con extensión `.mf`) que describa su contenido, por ejemplo:

```
Manifest-Version: 1.0
Built-By: maria
Build-Jdk: 17.0.4.1
Created-By: Maven Integration for Eclipse
Name: es/teis/beans/BeanFuente.class
Java-Bean: True

Name: es/teis/beans/BeanReceptor.class
Java-Bean: True

Name: es/teis/beans/VetoableBeanReceptor.class
Java-Bean: True

Name: es/teis/beans/VetoableBeanFuente.class
Java-Bean: True
Java-Bean: False
```

Observa, en el fichero de manifiesto como la clase del componente va acompañada de `Java-Bean: True`, indicando que es un `JavaBean`.

Hay que asegurarse de que exista una línea vacía entre las entradas y que la última línea termina con un salto de línea. No debe haber espacios en blanco al final de las líneas.

El comando básico para la creación del paquete jar (una vez compilados los ficheros fuente) tiene este formato:

```
jar cfm jar-file manifest-addition input-files
```

- La opción c indica que desea crear un archivo JAR.
- La opción m indica que desea fusionar información de un archivo existente en el archivo de manifiesto del archivo JAR que está creando.
- La opción f indica que desea que la salida vaya a un archivo (el archivo JAR que está creando) en lugar de a la salida estándar.
- manifest-addition es el nombre (o la ruta y el nombre) del archivo de texto existente cuyo contenido desea agregar al contenido del manifiesto del archivo JAR.
- jar-file es el nombre que desea que tenga el archivo JAR resultante.
- El argumento de los archivos de entrada es una lista separada por espacios de uno o más archivos que desea colocar en su archivo JAR.
- Las opciones m y f deben estar en el mismo orden que los argumentos correspondientes.
- Nota: El contenido del manifiesto debe estar codificado en UTF-8.

Os he dejado un archivo MANIFEST.MF en src/main/resources del proyecto MisBeans.

1. Abrir un símbolo del sistema
2. Situarse dentro de \MisBeans\target\classes
3. Ejecutar el comando

```
jar cfm misbeans.jar ../../src/main/resources/MANIFEST.MF es/teis/beans/*.class
```

Esto generará el fichero misbeans.jar en \MisBeans\target\classes y luego tendrá que añadirse como external JAR al build path

4 Recursos

https://github.com/add-code-2223/MisBeans

https://github.com/add-code-2223/TestMisBeans.git

https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html