Creación de componentes visuales.

Caso práctico

Juan no ha entrado con buen pie esta mañana en la oficina. Ha llegado a primera hora con mucho trabajo pendiente y está teniendo contratiempos, puesto que su ordenador no ha querido arrancar. **María**, que siempre está encantada de desmontar cajas y ver qué hay por dentro, le está echando una mano para averiguar dónde está la avería.

-Siempre me ha llamado la atención cómo todos los dispositivos encajan en la placa y con la caja para funcionar en conjunto -le cuenta **María** a Juan-, aunque cada uno de ellos sea independiente y tenga su función concreta, en realidad por sí mismos no hacen nada. Tenemos la memoria, el disco duro, la placa, e incluso la placa está compuesta de otros elementos más pequeños con su propia función, fíjate por ejemplo en el chip que contiene la BIOS... Aquí está, la fuente de alimentación ha sufrido un cortocircuito, probablemente hayamos tenido algún pico de tensión esta noche y te ha dejado la fuente literalmente "frita". No te preocupes, la cambiaremos por una nueva y en la próxima reunión del equipo propondremos instalar un sistema de alimentación ininterrumpida propio para evitar que vuelva a suceder.

Como bien ha observado **María**, un equipo informático está formado por una serie de componentes electrónicos con un función muy específica que pueden incorporarse a sistemas más grandes (a veces no tienen sentido fuera de estos sistemas más grandes). Se producen en serie lo que permite reducir costes y mejorar la calidad del producto final, al ser bien conocido por sus fabricantes.

1.- Concepto de componente. Características.

Caso práctico

María continúa explicando la tecnología de componentes hardware a **Juan**:

—Un componente es una pieza de hardware o software pequeña, que tiene un comportamiento específico y dispone de una interfaz que permite que se inserte fácilmente. Por ejemplo, en la creación de sistemas hardware es muy habitual basarse en componentes pequeños con funciones específicas, como la BIOS, las tarjetas de memoria, los microprocesadores, etc. Los fabricantes de estos componentes están muy especializados, lo que les permite ser muy competitivos porque pueden trabajar sobre sus piezas día a día sin modificar demasiado cómo se insertan en el sistema general (su interfaz), lo que redunda en el aumento de la calidad.

Un **componente software** es una clase creada para ser reutilizada y que puede ser manipulada por una herramienta de desarrollo de aplicaciones visual. Se define por su estado que se almacena en un conjunto de **propiedades**, las cuales pueden ser modificadas para adaptar el componente al programa en el que se inserte. También tiene un **comportamiento** que se define por los eventos ante los que responde y los **métodos** que ejecuta ante dichos eventos.

Citas para pensar

Un JavaBean es un componente software reusable que puede ser manipulado visualmente mediante una herramienta gráfica.)

(Especificación de JavaBeans, Sun 1997

Los componentes tienen una interfaz bien definida formada por sus propiedades y métodos, y se distribuyen mediante un paquete instalable que contiene todo lo necesario para su funcionamiento. Además, deben ser independientes de otras bibliotecas o componentes.

Para que una clase sea considerada un componente debe cumplir ciertas normas:

- Debe poder modificarse para adaptarse a la aplicación en la que se integra.
- Debe tener persistencia, es decir, debe poder guardar el estado de sus propiedades cuando han sido modificadas.
- Debe tener introspección, es decir, debe permitir a un IDE que pueda reconocer ciertos elementos de diseño como los nombres de las funciones miembros o métodos y definiciones de las clases, y devolver esa información.
- Debe poder gestionar **eventos**.

El desarrollo basado en componentes tiene, además, las siguientes ventajas:

- Es mucho más sencillo y se realiza en menos tiempo y con un coste inferior.
- Se disminuyen los errores en el software ya que los componentes se deben someter a un riguroso control de calidad antes de ser utilizados.

Reflexiona

Ya has creado algunas interfaces gráficas con diferentes herramientas como NetBeans o Designer de QT. En ambos casos el procedimiento consiste en seleccionar los controles gráficos de la interfaz y posicionarlos en un lienzo que será la interfaz. Todos estos controles **cumplen con los requisitos para ser componentes, de hecho lo son**, son elementos reutilizables que pueden ser manejados por una herramienta de desarrollo visual. Por ejemplo, se puede modificar su tamaño o color para adaptarlo a la interfaz y estos cambios permanecen después de cerrarla, además tienen una interfaz formada por un conjunto de métodos y propiedades accesibles desde la paleta de propiedades.

Autoevaluación

El uso de componentes software implica disminuir del coste de producción de software.

- ○ Verdadero.
- ○ Falso.

Es correcto. Así es, el uso de componentes implica que usamos software ya programado que ha sido probado exhaustivamente antes de su distribución y con una funcionalidad cerrada, aunque podamos adaptar el componente a nuestras necesidades.

No es correcto, la afirmación anterior es verdad.

Solución

1. Opción correcta
2. Incorrecto

2.- Propiedades y atributos.

Como en cualquier clase, un componente tendrá definido un estado a partir de un conjunto de **atributos.** Los atributos son variables definidas por su nombre y su tipo de datos que toman valores concretos. Normalmente los atributos son privados y no se ven desde fuera de la clase que implementa el componente, se usan sólo a nivel de programación.

Las propiedades son un tipo específico de atributos que representan características de un componente que afectan a su apariencia o a su comportamiento. Son accesibles desde fuera de la clase y forman parte de su interfaz. Suelen estar asociadas a un atributo interno.

Una propiedad no es exactamente un atributo público, sino que tiene asociados ciertos métodos que son la única manera de poder modificarla o devolver su valor y que pueden ser de dos tipos:

- getter: permiten leer el valor de la propiedad. Tienen la estructura:

```
public <TipoPropiedad> get<NombrePropiedad>( )
```

- si la propiedad es booleana el método getter se implementa así:

```
public boolean is<NombrePropiedad>()
```

- Setter: permiten establecer el valor de la propiedad. Tiene la estructura:

```
public void set<NombrePropiedad>(<TipoPropiedad> valor)
```

Si una propiedad no incluye el método set entonces es una propiedad de sólo lectura.

Por ejemplo, si estamos generando un componente para crear un botón circular con sombra, podemos tener, entre otras, una propiedad de ese botón que sea color, que tendría asociados los siguientes métodos:

```
public void setColor(String color)
public String getColor()
```

Caso práctico

Ada se da cuenta de que esta conversación le interesa de cara a la nueva tecnología de software basada en componentes que quiere implantar en la empresa, y decide intervenir...

-A ver **Juan**, ¿me podrías dar un ejemplo de componente hardware?

-Una fuente de alimentación -**Juan** no tiene dudas.

-Cierto, las más antiguas estaban preparadas para trabajar con dos diferenciales de potencial diferentes, 125 V y 220 V para poder adaptarse a las instalaciones de cualquier vivienda o empresa. Para cambiar de uno a otro sólo era necesario cambiar una clavija de posición, algo parecido a lo que ocurre con los componentes software: tienen un estado interno que podemos cambiar ejecutando del conjunto de métodos que forman su interfaz.

Debes Conocer

En el lenguaje de programación Java los componentes se crean utilizando la tecnología JavaBeans, que consiste en crear una clase con unas características especiales que puede ser reutilizada después de una manera muy sencilla. De hecho, es común que a un componente Java se le llame Bean. A continuación tienes un enlace a un ejemplo de creación de un Bean.

Creación de un JavaBean.

Realiza los pasos del tutorial y analiza cual es la función principal de un componente.

Mostrar retroalimentación

La principal función de un componente es recoger en una entidad que sea fácilmente reutilizable un segmento que código con una funcionalidad cerrada. Aporta ventajas como la reutilización, disminución de costes al utilizar código ya hecho y mejorar la robustez y tolerancia a fallos de los nuevos programas ya que suelen estar ampliamente probados.

3.- Editores de propiedades.

Caso práctico

Ada explica a **María** y **Juan** cómo modificar las propiedades:

-Puedes cambiar las propiedades de un componente para modificar su aspecto o su estado interno. Por ejemplo, si añades un botón a un formulario, sus propiedades serán el color de fondo, la fuente, y el texto que aparece en el botón.

Cuando necesitamos editar estas propiedades podemos hacerlo escribiendo los valores directamente o desplegando un diálogo que permita editar el valor en el inspector de propiedades.

Una de las principales características de un componente es que una vez instalado en un entorno de desarrollo, éste debe ser capaz de identificar sus propiedades simplemente detectando parejas de operaciones get/ set, mediante la capacidad denominada introspección.

El entorno de desarrollo podrá editar automáticamente cualquier propiedad de los tipos básicos o de las clases `Color` y `Font`. Aunque no podrá hacerlo si el tipo de datos de la propiedad es algo más complejo, por ejemplo, si usamos otra clase, como Cliente. Para poder hacerlo tendremos que crear nuestro propio editor de propiedades.

Un **editor de propiedad** es una herramienta para personalizar un tipo de propiedad en particular. Los editores de propiedades se utilizan en la ventana Propiedades, que es donde se determina el tipo de la propiedad, se busca de un editor de propiedades apropiado, y se muestra el valor actual de la propiedad de una manera apropiada.

La creación de un editor de propiedades usando tecnología Java supone programar una clase que implemente la interfaz `PropertyEditor`, que proporciona métodos para especificar cómo se debe mostrar una propiedad en la hoja de propiedades. Su nombre debe ser el nombre de la propiedad seguido de la palabra Editor:

```
public Editor implements PropertyEditor {…}
```

Por defecto la clase `PropertyEditorSupport` que implementa `PropertyEditor` proporciona los editores más comúnmente empleados, incluyendo los mencionados tipos básicos, `Color` y `Font`.

Una vez que tengamos todos los editores, tendremos que empaquetar las clases con el componente para que use el editor que hemos creado cada vez que necesite editar la propiedad. Así, conseguimos que cuando se añada un componente en un panel y lo seleccionemos, aparezca una hoja de propiedades, con la lista de las propiedades del componente y sus editores asociados para cada una de ellas. El IDE llama a los métodos getters, para mostrar en los editores los valores de las propiedades. Si se cambia el valor de una propiedad, se llama al método setter, para actualizar el valor de dicha propiedad, lo que puede o no afectar al aspecto visual del componente en el momento del diseño.

Autoevaluación

Imagina que estamos creando un componente para añadir objetos de tipo persona. Indica para qué propiedad crees más adecuado implementar un editor de propiedades:

- ○ La edad (es un entero).
- ○ El nombre (es un String).
- ○ La dirección, que es un objeto de tipo TDireccion, formado por una cadena o String para la calle, localidad y provincia y un entero para el número de la casa.

No es correcta, ya que los números enteros se editan directamente sin problema.

No es correcto, ya que las cadenas de caracteres se editan directamente sin problema.

Así es, ya que no disponemos de un editor por defecto para los atributos compuestos.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

Ejemplo de creación de un componente JavaBean con netBeans.

Para ilustrar cómo se implementa un componente con una herramienta como NetBeans, crearemos uno que permita almacenar la información de un persona, guardaremos por ejemplo, su nombre, apellidos, teléfono y dirección:

- El **nombre** se puede representar como una propiedad de tipo String.
- Los **apellidos** también se pueden representar como una propiedad de tipo String.
- El **teléfono** también puede ser una cadena de caracteres o String.
- La **dirección** será una propiedad compuesta, por la dirección propiamente dicha, la población y la provincia.

Dado que la última propiedad no es simple, será necesario implementar un editor de propiedades para modificarla desde NetBeans. Veremos como crear este ejemplo.

Para saber más...

Si quieres conocer un poco más de este apartado de la tecnología Java puedes consultar la página del tutorial que tiene Oracle alojado en su página (está en inglés) y la especificación de la interfaz `PropertyEditor`:

Personalización de un bean.

Interfaz `PropertyEditor`.

4.- Eventos. Asociación de acciones a eventos.

Caso práctico

-Creo que lo he entendido todo, un componente tiene una serie de propiedades que lo definen y métodos que modifican que pueden operar sobre ellas. Es algo parecido a los objetos y clases de UML, con los que trabajamos el año pasado -comenta **Juan.**

-Es verdad -continúa **Ada**-, pero un componente puede hacer algo más, también pueden desencadenar eventos con los que enviar información de un objeto a otro y también entre el usuario y el componente. Si haces clic en el botón Eventos, del panel de propiedades de cualquier componente, verás una lista de todos los eventos que el botón es capaz de disparar.

Se puede programar un componente para que reaccione ante determinadas acciones del usuario, como un clic del ratón o la pulsación de una tecla del teclado. Cuando estas acciones se producen, se genera un **evento** que el componente puede capturar y procesar ejecutando alguna función. Pero no solo eso, un componente puede también lanzar un evento **cuando sea necesario y que su tratamiento se realice en otro objeto.**

Los eventos que lanza un componente, se reconocen en las herramientas de desarrollo y se pueden usar para programar la realización de acciones.

Para que el componente pueda reconocer el evento y responder ante el tendrás que hacer lo siguiente:

- Crear una clase para los eventos que se lancen.
- Definir una interfaz que represente el oyente (`listener`) asociado al evento. Debe incluir una operación para el procesamiento del evento.efinir dos operaciones, para añadir y eliminar oyentes. Si queremos tener más de un oyente para el evento tendremos que almacenar internamente estos oyentes en una estructura de datos como `ArrayList` o `LinkedList`:

```
public void add<Nombre>Listener(<Nombre>Listener l)

public void remove<Nombre>Listener(<Nombre>Listener l)
```

Finalmente, recorrer la estructura de datos interna llamando a la operación de procesamiento del evento de todos los oyentes registrados. En resumen:

Resumen textual alternativo

Autoevaluación

Tenemos un componente que implementa un termostato que regula un ventilador. Mide constantemente la temperatura, de modo que cuando se alcanza una temperatura límite se enciende. El concepto de alcanzar la temperatura límite tiene que ver con...

- ○ ... las propiedades.
- ○ ... los métodos.
- ○ ... con los eventos.

No es correcta, una propiedad sería la temperatura actual por ejemplo.

Incorrecto, alcanzar la temperatura no es un procedimiento que el termostato pueda realizar, como sería por ejemplo iniciar el ventilador.

Correcto. Así es, al detectar que la temperatura actual coincide con la temperatura límite se debe lanzar un evento que recoja el ventilador para iniciarse.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

5.- Introspección. Reflexión.

Caso práctico

Ada está muy satisfecha con la conversación.

-Habéis comentado que los controles que insertamos en un formulario son también componentes, y es así porque coinciden todas sus características. ¿No te llama la atención la capacidad que tienen las herramientas que usamos de detectar el nombre y el tipo de las propiedades?

-Es verdad -contesta **María**-, también, cuando escribo código directamente, las ayudas a la edición te dan pistas acerca de los métodos que puedes usar o qué argumentos debes colocar en una función. ¿Cómo puede la herramienta conocer eso?

Un componente, como cualquier otra clase dispone de una interfaz, que es el conjunto de métodos y propiedades accesibles desde el entorno de programación. Normalmente, la interfaz la forman los atributos y métodos públicos.

La **introspección** es una característica que permite a las herramientas de programación visual arrastrar y soltar un componente en la zona de diseño de una aplicación y determinar dinámicamente qué métodos de interfaz, propiedades y eventos del componente están disponibles.

Esto se puede conseguir de diferentes formas, pero en el nivel más bajo se encuentra una característica denominada reflexión, que busca aquellos métodos definidos como públicos que empiezan por get o set, es decir, se basa en el uso de patrones de diseño, es decir, en establecer reglas en la construcción de la clase de forma que mediante el uso de una nomenclatura específica se permita a la herramienta encontrar la interfaz de un componente.

En **JavaBeans** la introspección se puede conseguir de varias maneras:

- Reflexión de bajo nivel, que utiliza patrones de diseño para descubrir las características del componente por medio de las posibilidades de reflexión del paquete `java.lang.reflect`.
- Examinando una clase asociada de información del componente (`BeanInfo`) que describe explícitamente sus características para que puedan ser reconocidas.

Para saber más

En el siguiente enlace podrás acceder a una página web de Introducción a los JavaBeans donde podrás ampliar los conceptos de introspección, persistencia, reflexión, etc. de componentes en Java.

Características de los JavaBeans.

Autoevaluación

La reflexión es una forma de implementar la introspección a bajo nivel, sin depender de la aplicación.

- ○ Verdadero.
- ○ Falso.

Correcto. Así es, es la forma más sencilla de implementar la introspección, ya que solo depende del código que se añada a la clase.

No es correcto, ciertamente la reflexión es el modo más sencillo de introspección.

Solución

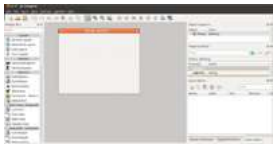
1. Opción correcta
2. Incorrecto

6.- Persistencia del componente.

Caso práctico

Ada continúa hablando sobre los componentes...

-Fijaos en editor de texto que utilizáis a diario, o en el IDE con el que desarrolláis vuestros programas, cada uno lo utiliza de una forma, a **María** le gusta dejar el máximo espacio para poder escribir sus programas, sin embargo, **Juan** necesita tener cuantos más paneles y barras de herramientas mejor, y cada mañana, cuando encendéis el equipo, la configuración que habéis establecido continúa ahí, y podéis continuar vuestro trabajo en lugar de tener que empezar cambiar, añadir y quitar cosas, con los componentes pasa algo parecido.



A veces, necesitamos almacenar el estado de una clase para que perdure a través del tiempo. A esta característica se le llama **persistencia**. Para implementar esto, es necesario que pueda ser almacenada en un archivo y recuperado posteriormente.

El mecanismo que implementa la persistencia se llama serialización.

Al proceso de almacenar el estado de una clase en un archivo se le llama serializar.

Al de recuperarlo después deserializar.

Todos los componentes deben persistir. Para ello, siempre desde el punto de vista Java, deben implementar los interfaces `java.io.Serializable` o `java.io.Externalizable` que te ofrecen la posibilidad de serialización automática o de programarla según necesidad:

Serialización Automática: el componente implementa la interfaz Serializable que proporciona serialización automática mediante la utilización de las herramientas de Java Object Serialization. Para poder usar la interfaz Serializable debemos tener en cuenta lo siguiente:

- Las clases que implementan `Serializable` deben tener un constructor sin argumentos que será llamado cuando un objeto sea "reconstituido" desde un fichero .ser.
- Todos los campos excepto `static` y `transient` son serializados. Utilizaremos el modificador `transient` para especificar los campos que no queremos serializar, y para especificar las clases que no son serializables (por ejemplo `Image` no lo es).
- Se puede programar una **serialización propia** si es necesario implementando los siguientes métodos (las firmas deben ser exactas):

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException;
private void readObject(java.io.ObjectInputStream in) throws IOExcept;
```

Serialización programada: el componente implementa la interfaz `Externalizable`, y sus dos métodos para guardar el componente con un formato específico. Características:

- Precisa de la implementación de los métodos `readExternal ()` y `writeExternal ()`.
- Las clases `Externalizable` también deben tener un constructor sin argumentos.

Los componentes que implementarás en esta unidad emplearán la serialización por defecto por lo que debes tener en cuenta lo siguiente:

- La clase debe implementar la interfaz `Serializable`.

- Es obligatorio que exista un constructor **sin argumentos.**

7.- Propiedades simples e indexadas.

Caso práctico

María ya había investigado algo del concepto de componente y ha hecho alguna prueba...

-El otro día -comenta **María**- estaba programando un componente JavaBeans para gestionar los discos que tengo, sin embargo, tuve un problema, porque al no saber a priori cuantos discos va a ser no puede crear un número fijo de propiedades...,

-Tu problema tiene fácil solución, como haría en cualquier otro programa si necesitas almacenar un número indeterminado de elementos lo más sencillo es usar un vector que vayas rellenando dinámicamente según te vaya haciendo falta.

Como siempre **Ada** está en todo.

Una propiedad simple representa un único valor, un número, verdadero o falso o un texto por ejemplo.

Tiene asociados los métodos **`getter`** y **`setter`** para establecer y rescatar ese valor. Por ejemplo, si un componente de software tiene una propiedad llamada peso de tipo real susceptible de ser leída o escrita, deberá tener los siguientes métodos de acceso:

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException;

private void readObject(java.io.ObjectInputStream in) throws IOExcept;
```

Una propiedad simple es de sólo lectura o sólo escritura si falta uno de los mencionados métodos de acceso.

Una **propiedades indexadas** representa un conjunto de elementos, que suelen representarse mediante un vector y se identifica mediante los siguientes patrones de operaciones para leer o escribir elementos individuales del vector o el vector entero (fíjate en los corchetes del vector):

```
public <TipoProp>[] get<NombreProp>()
public void set<NombreProp> (<TipoProp>[] p)
public <TipoProp> get<NombreProp>(int posicion)
public void set<NombreProp> (int posicion, <TipoProp> p)
```

Aquí tienes un ejemplo de propiedad indexada. El componente almacena un conjunto de miembros, a los que puedes acceder en conjunto o de uno en uno.

```
private String[] miembros = new String[0];

public String getMiembros(int pos){
return miembros[pos];
}
public String[] getMiembros(){
    return miembros;
}

public void setMiembros(int pos, String miembro){
    miembros[pos] = miembro;
}
public void setMiembros(String[] miembros){
    if(miembros == null){
        miembros = new String[0];
    }
    this.miembros = miembros;
}
```

A continuación puedes descargar el código asociado para que lo estudies:

Código para una propiedad indexada. (1.58 KB)

Autoevaluación

¿En qué caso crees que es más adecuado usar una propiedad indexada?

- ○ Estamos creando un componente para representar un panel de estados en el que tenemos tres posibles estados.
- ○ Para representar los teléfonos fijo y móvil y de trabajo de una persona.

- ○ Para representar los hijos de una persona.

Incorrecto, para representar tres estados sería más adecuado usar tres propiedades de tipo booleano.

No es correcto, en este caso usaríamos tres propiedades de tipo vector de enteros o String.

Correcto. Así es, al no tener un número de hijos concreto es más útil una propiedad indexada en la que a priori no se sabe cuantos elementos vamos a tener.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

8.- Propiedades compartidas y restringidas.

Caso práctico

María continúa con sus problemas de propiedades...

-Y si tengo una propiedad que no debería cambiar en según qué circunstancias, ¿cómo lo gestiono?

Los objetos de una clase que tiene una **propiedad compartida o ligada** notifican a otros objetos oyentes interesados, cuando el valor de dicha propiedad cambia, permitiendo a estos objetos realizar alguna acción. Cuando la propiedad cambia, se crea un objeto (de una clase que hereda de `ObjetEvent`) que contiene información acerca de la propiedad (su nombre, el valor previo y el nuevo valor), y lo pasa a los otros objetos oyentes interesados en el cambio.

La notificación del cambio se realiza a través de la generación de un `PropertyChangeEvent`PropertyChangeEvent. Los objetos que deseen ser notificados del cambio de una propiedad limitada deberán registrarse como auditores. Así, el componente de software que esté implementando la propiedad limitada suministrará métodos de esta forma:

```
public void addPropertyChangeListener (PropertyChangeListener l)
public void removePropertyChangeListener (PropertyChangeListener l)
```

Los métodos precedentes del registro de auditores no identifican propiedades limitadas específicas. Para registrar auditores en el `PropertyChangeEvent` de una propiedad específica, se deben proporcionar los métodos siguientes:

```
public void addPropertyNameListener (PropertyChangeListener l)
public void removePropertyNameListener (PropertyChangeListener l)
```

En los métodos precedentes, `PropertyName` se sustituye por le nombre de la propiedad limitada. Los objetos que implementan la interfaz `PropertyChangeListener` deben implementar el método `propertyChange()`. Este método lo invoca el componente de software para todos sus auditores registrados, con el fin de informarles de un cambio de una propiedad.

Una **propiedad restringida** es similar a una propiedad ligada salvo que los objetos oyentes que se les notifica el cambio del valor de la propiedad tienen la opción de vetar cualquier cambio en el valor de dicha propiedad.

Los métodos que se utilizan con propiedades simples e indexadas que veíamos anteriormente se aplican también a las propiedades restringidas. Además, se ofrecen los siguientes métodos de registro de eventos:

```
public void addPropertyVetoableListener (VetoableChangeListener l)
public void removePropertyVetoableListener (VetoableChangeListener l)
public void addPropertyNameListener (VetoableChangeListener l)
public void removePropertyNameListener (VetoableChangeListener l)
```

Los objetos que implementa la interfaz `VetoableChangeListener` deben implementar el método `vetoableChange()`. Este método lo invoca el componente de software para todos sus auditores registrados con el fin de informarles del cambio en una propiedad.

Todo objeto que no apruebe el cambio en una propiedad puede arrojar una `PropertyVetoException` dentro del método `vetoableChange()`, para informar al componente cuya propiedad restringida hubiera cambiado de que el cambio no se ha aprobado.

Autoevaluación

Supongamos que tenemos un componente para gestionar las notas de los estudiantes de una clase con una propiedad que almacena el número de estudiantes de la clase, ¿de qué tipo sería esta propiedad?

- ○ Simple.
- ○ Indexada.

Correcto, representa un número por lo que debemos declarar la propiedad como un entero y nada más.

No es correcto, hace referencia al conjunto de alumnos, pero no deja de ser un número que se representa como un valor de tipo entero.

Solución

1. Opción correcta
2. Incorrecto

9.- Herramientas para el desarrollo de componentes visuales.

Caso práctico

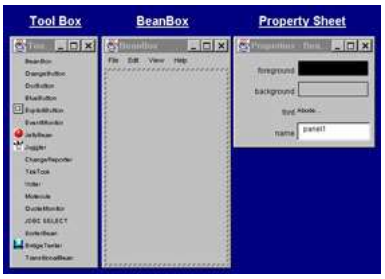
-Llevamos toda la mañana hablando y aún no hemos hecho nada real, creo que lo ideal es empezar por conocer que herramientas podemos usar para programar un componente, sin duda necesitaremos que cumpla con algunas características como las ayudas al código o generación de clases auxiliares -comenta **Juan**.

BeanBox

Es la primera herramienta que se creó para trabajar con componentes de `JavaBeans` y se podría definir como un contenedor de Beans. Se proporcionaba desde la página web de Sun, y la documentación sobre `JavaBeans` se basaba en esta herramienta.

En `BeanBox` se pueden escribir los Beans, para luego arrastrarlos dentro de `BeanBox` y comprobar si funcionan como se esperaba.

El `BeanBox` también puede verse como una demostración de cómo deben comportarse las herramientas de desarrollo compatibles con los `JavaBeans`. El `BeanBox` es una buena herramienta para aprender a trabajar con Beans, aunque hoy en día ha sido sustituida por `NetBeans`.



`Bean Builder`

En una versión mejorada del `BeanBox`. Con esta aplicación se comenzó a dar un enfoque al proyecto que se desarrollaba, puesto que al igual que el `BeanBox` permitía el enlazado de componentes de manera gráfica y ver sus propiedades en tiempo que diseño, pero a diferencia de él y lo que más interesante resultaba era que no creaba archivos .ser para la serialización de los objetos sino que utiliza archivos XML que podían ser entendidos por cualquier aplicación.

`NetBeans`

Es la apuesta actual de Oracle para crear y utilizar `Beans`. Proporciona una base completa para todo el ciclo de vida de creación del `Bean`, y ayudas para la escritura de código para añadir propiedades, métodos getter y setter en propiedades simples, complejas, compartidas o ligadas, o para la generación de eventos. Además desde el administrador de la paleta es extremadamente sencillo incorporar nuevos beans desde sus archivos .jar, e incorporarlos a una aplicación solo precisa de un par de clics.

Una de las características que aporta NetBeans es la creación de la clase **BeanInfo** para el componente. Una clase `BeanInfo` puede gestionar el modo en que el componente aparece en la herramienta de desarrollo de aplicaciones, indicando que propiedades deben aparecer, en primer lugar, cual es deben estar ocultas, si existe un editor de propiedades asociado, etc, siendo además una de las bases de la **introspección** en Java.

Puedes generar un `BeanInfo` automáticamente desplegando el menú contextual de la clase que implementa el componente en el inspector de proyectos de `NetBeans` y seleccionar **Editor** `BeanInfo`, si no existe te preguntará si quieres crearlo, cuando lo hagas se creará la clase `BeanInfo` y se mostrará el código, si cambias al diseño visual verás un panel que te permite gestionar gráficamente las propiedades de tu componente.

La principal ventaja de utilizar este método es que cuando nuestro componente hereda de una clase del tipo `JPanel` o `JFrame`, que son bastante complejas, el manejo de las propiedades heredadas es automático en el `BeanInfo`.

Autoevaluación

¿Qué herramienta usarías en la actualidad para crear un componente?

- ○ BeanBox.

- ○ BeanBuilder.
- ○ NetBeans.

Incorrecto, ésta herramienta está obsoleta y ya no se usa.

No es correcto, ésta herramienta está obsoleta y ya no se usa.

Correcto, es la mejor para crear un componente visual de una forma rápida y cómoda.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

10.- Empaquetado de componentes.

Caso práctico

-Vale, ya tenemos el componente creado y probado, estamos en condiciones de poder usarlo, pero como lo hago, ¿tengo que ir llevando mi clase de un proyecto a otro? -Dice **María**.

-No es así -contesta **Ada**-, lo que debes hacer es reunir todos los archivos en uno solo que podrás distribuir a cualquier aplicación que lo necesite.

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

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

Puedes incluir varios componentes en un mismo jar.

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

```
Manifest-Version: 1.0
Name: demo/Componente.class
Java-Bean: True
Name: demo/ComponenteBeanInfo.class
Java-Bean: False
Name: demo/ClaseAuxiliar.class
Java-Bean: False
Name: demo/Imagen.png
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`. La forma más sencilla de generar el archivo `jar` es utilizar la herramienta **Limpiar y construir** del proyecto en `NetBeans`, que deja el fichero `.jar` perfectamente en el directorio `/dist` del proyecto, aunque siempre puedes recurrir a la orden `jar` y crearlo tu directamente:

```
jar cfm Componente.jar manifest.mf Componente.class ComponenteBEanInfo.class ClaseAuxiliar.class Imagen.png proyecto.jar
```

Autoevaluación

La distribución de los componentes se realiza a través de...

- ○ La distribución de la clase que implementa la interfaz Serializable.
- ○ La copia de las clases que forman el componente en el proyecto que las usa.
- ○ El empaquetado de las clases que forman el componente (incluidas las clases que programan los objetos eventos y las clases con la información del componente y distribuyendo este paquete.

Incorrecta puesto que podemos tener más clases implicadas.

No es correcta. Podría hacerse pero no es la opción más cómoda ni la más segura.

Correcta. Así es, es necesario crear un paquete con todas las clases que forman el componente, incluidas las que se ocupan de los eventos relacionados o aquellas que contienen la información del componente.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

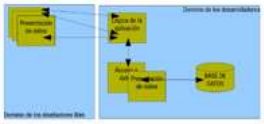
11.- Elaboración de un componente de ejemplo.

Caso práctico

-Por fin hemos revisado todos los aspectos relevantes de un componente, creo que ya estamos listos para empezar a teclear. -**Juan** está encantado, ya que la programación es su aspecto favorito de la informática.

-Pues no lo creas, aunque no lo parezca, con el uso de herramientas de desarrollo gráficas prácticamente se hace en un par de clics. Se elimina mucha inserción de código a mano -dice **Ada**, práctica como siempre.

En el ámbito del acceso a datos que estudiamos en este módulo formativo el uso de componentes está muy relacionado con el mundo de la creación de aplicaciones web. La tecnología JSP utiliza una arquitectura denominado MVC (Modelo-Vista-Controlador), cuya misión fundamental es separar el desarrollo de la base de datos (Modelo), creada por ejemplo en MySQL o cualquier otro gestor de bases de datos, de la interfaz (Vista), que puede hacerse usando HTML en combinación con JSP, de la lógica de negocio (Controlador) cuya implementación más sencilla y fácil de gestionar por JSP es un JavaBean que acceda a la base de datos y permita recuperar fácilmente la información que contiene.



En el esquema el modelo sería la base de datos, la vista el apartado de presentación de datos y el controlador está formado por la lógica de la aplicación y acceso a datos.

Para saber más

En el siguiente enlace podrás acceder a una página web con un ejemplo muy sencillo de este modelo. Para que sea un poco más sencillo no almacena la información en una base de datos sino en una lista.

Ejemplo del modelo MVC.

Para ejemplificar el uso de componentes JavaBean de acceso a datos, supondremos que queremos crear una aplicación web que muestre información de los alumnos del ciclo que mantendremos almacenados en una base de datos. De momento, nos encargaremos de los apartados Modelo y Controlador, dejando la vista para otros módulos del ciclo.

11.1.- Modelo o base de datos.

Nos basaremos en una base de datos muy sencilla que tendremos almacenada en un servidor MySql. El código SQL para crear la base de datos sería:

```
Base de datos: 'alumnos'
Estructura de tabla para la tabla 'alumnos'
CREATE TABLE IF NOT EXISTS `alumnos' (
     DNI varchar2(9) NOT NULL,
     Nombre varchar2(50) NOT NULL,
     Apellidos varchar2(70) NOT NULL,
     Direccion varchar2(100) NOT NULL,
     FechaNac date NOT NULL,
PRIMARY KEY (DNI)) ENGINE=MyISAM DEFAULT CHARSET=latin1;
Volcar la base de datos para la tabla 'alumnos'
INSERT INTO 'alumnos' VALUES
('12345678A', 'José Alberto', 'González Pérez', 'C/Albahaca, nº14, 1ºD', '1986-07-15'),
('23456789B', 'Almudena', 'Cantero Verdemar', 'Avd/ Profesor Alvarado, n27, 8ºA', '1988-11-04'),
('14785236d', 'Martín', 'Díaz Jiménez', 'C/Luis de Gongora, nº2.', '1987-03-09'),
('96385274f', 'Lucas', 'Buendia Portes', 'C/Pintor Sorolla, nº 16, 4ºB', '1988-07-10');
```

La base de datos se compone de una sola tabla de alumnos. Para cada alumno almacenamos su DNI, nombre, apellidos, dirección y fecha de nacimiento.

11.2.- Controlador o lógica del modelo.

En el entrono de la arquitectura MVC el controlador representa la lógica de negocio de una aplicación. Encapsula las reglas de negocio en componentes que son fáciles de probar, permiten mejorar la calidad del software y promueven la reutilización.

El estado del define el conjunto actual de valores del modelo e incluye métodos para cambiar estos valores. Estos métodos recogen parte de la lógica de negocio. Deberían de ser independientes del protocolo que se utilizara para acceder a ellos. Si resumimos la encapsulación, calidad del software, reutilización e independencia del protocolo obtenemos que los JavaBeans son la elección lógica para implementar estos componentes.

El componente se encarga de conectar con la base de datos y ejecutar la sentencia deseada. A través de los **métodos** puede posicionarse en cada fila de la consulta y almacena en las propiedades cada uno de los campos.

Las acciones definen los cambios permitidos para los estados en respuesta a los **eventos**.

11.3.- Estructura del JavaBean.

El componente a crear dispondrá de un constructor con las siguientes características:

- **Propiedades** que se corresponden con los campos de la tabla alumnos.
- **Clase auxiliar** llamada **Alumno** que usaremos para crear un vector de alumnos donde cargaremos el contenido de la tabla.
- **Vector** de alumnos.
- **Constructor** (sin argumentos):
  - Realiza la conexión a la base de datos.
  - Realiza la consulta.
  - Guarda los resultados en una estructura local al componente.
  - Establece los valores de las propiedades.
  - Cierra la conexión.
- **Métodos** para recorrer los resultados, recargando las propiedades cada vez que cambie. Usaremos los siguientes métodos:
  - `obtenerFilas()`: hace la consulta sobre la base de datos y almacena los resultados sobre un vector interno actualizando los valores de las propiedades.
  - `SeleccionarFilas/(i)`: se posiciona en el elemento i del vector y actualiza los valores de las propiedades.
  - `seleccionarDNI(String DNI)`: busca en el vector interno el registro que coincide con el valor del DNI y actualiza los valores de las propiedades.

Para la **elaboración de un componente**, debes tener muy claros los **pasos** que debes dar.

1. Creación del componente.
2. Adición de propiedades.
3. Implementación de su comportamiento.
4. Gestión de los eventos.
5. Uso de componentes ya creados en NetBeans.

En las siguientes secciones verás cómo se realizan estos pasos con el ejemplo que se acaba de exponer.

11.4.- Creación del componente.

Comenzamos creando un proyecto **NetBeans** nuevo de tipo Java Application. Nos aseguramos de desmarcar las opciones Crear clase principal y Configurar como proyecto principal y ponemos de nombre al proyecto **AlumnoBean**, por ejemplo.

Una vez creado el proyecto, le añadimos un archivo nuevo de tipo Componente **JavaBeans** (si no lo encuentras en la lista que sale en el menú contextual, haz clic en la opción Otros... para acceder al resto de tipos de archivos). Puedes llamar a la clase **AlumnoBean** y al paquete en el que se ubicará Alumno.

Para que una clase se pueda considerar un componente debe implementar la interfaz **Serializable** y, además, tener un **constructor sin argumentos** que vimos eran requisitos para la creación de componentes.
El proyecto cuenta con una propiedad de ejemplo que puedes eliminar (su declaración y los métodos **get** y **set** de la propiedad), así como un gestor de escuchadores de cambio de propiedades que no necesitaremos, quedando el siguiente código para empezar:

```
public class AlumnoBean implements Serializable {
     private PropertyChangeSupport propertySupport;
     public TemporizadorBean() {
          propertySupport = new PropertyChangeSupport(this);
     }
}
```

Autoevaluación

Cuando queremos crear un componente usamos la interfaz Serializable para...

- ○ La implementación de la clase.
- ○ Implementar la persistencia.
- ○ El uso de una propiedad de tipo **Serializable**.
- ○ Crear un componente con características de otro.

No es correcta, implementamos clases abstractas.

Correcta. Así es, implementar Serializable nos permite gestionar la persistencia del componente.

No es así, puesto que **Serializable** es una interfaz, no una clase.

Falso. Para eso usaremos la herencia con la palabra reservada **extends**.

Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto
4. Incorrecto

11.5.- Añadir propiedades.

El componente tendrá como propiedades las mismas que los campos de la base de datos: DNI, nombre, apellidos, dirección y fecha de nacimiento.

La adición de propiedades en una clase Java se realiza simplemente escribiendo el código de declaración del atributo privado (o protegido) y los métodos `getter` y `setter` que son la base de la **instrospección.**

Si bien `NetBeans` proporciona una ayuda especial para realizar esta tarea. Con el cursor posicionado dentro de la clase (puedes reservar una zona para el código de propiedades), haz clic con el botón secundario y selecciona la opción **Insertar Código**..., en la lista que te saldrá elige **Agregar propiedad.** Se desplegará el siguiente cuadro de diálogo en el que puedes escribir el nombre de la propiedad y su tipo, e insertar los métodos `get` y `set` de manera automática:

Fíjate que si quisieras crear una propiedad indexada o restringida es muy sencillo de hacer, sólo hay que marcar la opción correspondiente y la herramienta genera el código base de los métodos necesarios.

Autoevaluación

Para que una clase pueda ser considerada un componente debe implementar la interfaz `Serializable` y tener un constructor sin argumentos. ¿Verdadero o falso?

- ○ Verdadero.
- ○ Falso.

Correcto. Muy bien, esos son los requisitos.

No es correcto, debes repasar este punto.

Solución

1. Opción correcta
2. Incorrecto

11.6.- Implementar el comportamiento.

Con las propiedades listas, tenemos que programar el comportamiento del componente. En principio la idea es que al crear el componente se cargue el contenido de la tabla alumno de la base de datos en un vector de uso interno que nos va a servir para no tener que estar conectándonos constantemente, recuerda que estos componentes se usan en entornos cliente servidor con gran cantidad de accesos por parte de múltiples usuarios que pueden llegar a saturar la base de datos. Por este motivo programaremos el constructor para que realice la carga de datos. En un primer momento los valores de las propiedades serán los del primer alumno recuperado.

También generaremos un par de métodos para recuperar información de un registro según su posición o según su clave.

A continuación tienes el enlace al código de la clase con lo visto hasta el momento:

Añadiremos el siguiente código:

- Código de la clase AlumnoBean. (8 KB)

11.7.- Gestión de eventos.

Los componentes Java utilizan el modelo de delegación de eventos para gestionar la comunicación entre objetos como hemos visto. Para ilustrar el uso de los componentes usan los **eventos** para generar **acciones** en respuesta a los cambios del estado permitidos.

los eventos en un componente JavaBean añadiremos le añadiremos un comportamiento extra que añadirá un nuevo alumno a la base de datos. Tendrá como especial característica que cada vez que se inserte un alumno nuevo que generará un evento que alerte de esta modificación. Esto que en principio puede parecer redundante no lo es en un entorno multiusuario en el que varios clientes se conecten simultáneamente a la base de datos. Para poder implementar esto necesitarás varias cosas:

- Una clase que implemente los eventos. Esta clase hereda de `java.util.EventObject`. En el ejemplo se llamará `BDModificadaEvent`.
- Una interfaz que defina los métodos a usar cuando se genere el evento. Implementa `java.util.EventListener`. En este caso la gestión del evento se hará a través del método `capturarBDModificada` de la interfaz `BDModificadaListener`. También tenemos un objeto de tipo `DBModificadaListener` llamado receptor que representa aquellos programas que contienen al componente `AlumnosBean` susceptibles de recibir el evento.
- Dos métodos, `addEventoListener` y `removeEventoListener` que permitan al componente añadir oyentes y eliminarlos. En principio se deben encargar de que pueda haber varios oyentes. En nuestro caso sólo vamos a tener un oyente, pero se suele implementar para admitir a varios.
- Implementar el método que lanza el evento, asegurándonos de todos los oyentes reciban el aviso. En este caso lo que se ha hecho es lanzar el método que se creó en la interfaz que describe al oyente.

Finalmente el código de la clase componente queda como aparece en el siguiente enlace:

Código del componente AlumnoBean finalizado. (10 KB)

Autoevaluación

¿Qué elementos interviene en la implementación de un evento?

- ○ Una clase que define el evento
- ○ Una interfaz que define los métodos a implementar.
- ○ Una clase que defina el evento y una interfaz que defina los métodos a implementar.

Incorrecto. No es solo eso.

No es correcto. Además necesitas otra cosa...

Correcto, ambas cosas son necesarias en el modelo de eventos de Java.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

11.8.- Uso de componentes previamente elaborados en NetBeans.

Una vez construido el componente es sencillo incorporarlo a la paleta de NetBeans. Podemos hacerlo de diferentes formas:

- Si lo hemos desarrollado nosotros mismos (con NetBeans) basta con utilizar Limpiar y Construir para generar el fichero **.jar** con el componente.
- Si es un componente generado por otras personas, es preciso disponer de la distribución en un fichero **.jar**.

Para probar el componente tendrás que importar el fichero .jar desde las propiedades del proyecto con la nueva aplicación.

La gestión de eventos pasa por crear una clase que implemente la interfaz **BDModificadaListener**.

```
public class AccedeBD implements BDModificadaListener{
```

en el constructor indicamos que vamos a estar escuchando si se produce el evento:

```
AccedeBD()
     {
          alumnos = new AlumnoBean();
          alumnos.addBDModificadaListener( (BDModificadaListener)this );
     }
```

y sobreescribimos el método **capturarBDModificada(BDModificadaEvent ev)** para escribir un mensaje cuando se produzca el evento.

```
public void capturarBDModificada(BDModificadaEvent ev)
    {
        System.out.println("Se ha añadido un elemento a la base de datos");
    }
```

Para saber más

A continuación tienes el código de la clase con el proyecto que programa componente y el proyecto en el que se usa.

Para ejemplificar el uso del componente hemos creado una clase que implementa un listado de alumnos y que muestra un mensaje en caso de producirse el evento de base de datos modificada.

Para que puedas estudiar y probar el código tendrás que descomprimir la carpeta en alguna carpeta de tu disco duro. Puedes hacerlo en el directorio donde se almacenan los proyectos de **NetBeans**. Abre los proyectos desde **Archivo >> Abrir Proyecto**. Dado que se presentan completos puedes ver el código, compilarlos y ejecutarlos sin problemas, no obstante recuerda que para usar el componente en otra aplicación debes añadirlo como archivo **.jar** a sus bibliotecas.

Código de ejemplo. (3.13 MB)

Anexo.- Licencias de recursos.

Licencias de recursos utilizados en la Unidad de Trabajo.

Recurso (1)	Datos del recurso (1)	Recurso (2)	Datos del recurso (2)
	Autoría: pgbrandolin. Licencia: CC BY-CN-SA 3.0. Procedencia: http://www.openclipart.org/image/800px/svg_to_png/pgb-chip-packetproc.png		Autoría: warszawianka. Licencia: Dominio público. Procedencia: http://openclipart.org/detail/36085/tango-font-generic-by-warszawianka
	Autoría: warszawianka. Licencia: Dominio público. Procedencia: http://openclipart.org/detail/35383/tango-applications-graphics-by-warszawianka		Autoría: warszawianka. Licencia: Dominio público. Procedencia: http://openclipart.org/detail/34177/tango-ma reply-all-by-warszawianka
	Autoría: warszawianka. Licencia: Dominio público. Procedencia: http://www.openclipart.org/detail/32407/tango-edit-find-by-warszawianka		Autoría: Nokia. Licencia: GNU GPL. Procedencia: Captura de pantalla de la herramienta Designer .
	Autoría: warszawianka. Licencia: Dominio público. Procedencia: http://www.openclipart.org/detail/32245/tango-documnet-save-by-warszawianka		Autoría: warszawianka. Licencia: Dominio público. Procedencia: http://www.openclipart.org/image/800px/svg_to_png/docume open.png
	Autoría: Silveira Neto. Licencia: CC by-sa. Procedencia: http://www.flickr.com/photos/silveiraneto/2579658422/		Autoría: java.sun.com. Licencia: Copyright (cita), se autoriza el uso sin restricciones Procedencia: http://www.javaworld.com/javaworld/jw-0 1997/jw-09-beanbox.html
	Autoría: Oracle. Licencia: Copyright (cita). Procedencia: http://netbeans.org		Autoría: warszawianka. Licencia: Dominio público. Procedencia: http://openclipart.org/detail/35347/tango-syste users-by-warszawianka
	Autoría: María José Navascués González. Licencia: GNU GPL. Procedencia: Captura de pantalla de la aplicación NetBeans de Oracle.		Autoría: María José Navascués González. Licencia: GNU GPL. Procedencia: Captura de pantalla de la aplicación NetBea de Oracle.
	Autoría: Warszawiankak. Licencia: Dominio público. Procedencia: http://openclipart.org/detail/35533/tango-emblem-system-by-warszawianka		Autoría: warszawianka. Licencia: Dominio público. Procedencia: http://openclipart.org/detail/36145/tang package-x-generic-by-warszawianka