IES de Teis
Avda. de Galicia, 101
36216 – Vigo
Tfno: 886 12 04 64
e-mail: ies.teis@edu.xunta.es
http://www.iesteis.es

UD 3.- Creación de componentes visuales.

Enlace para exportar un proyecto javafx : https://javautodidacta.es/como-exportar-proyecto-javafx-intellij/

1 Desarrollo de componentes

1.1 Concepto de componente

Un **componente** es un elemento software reutilizable, con una especificación y funcionalidad clara, usando en la ingeniería de software basada en componentes.

Por definición, un **componente** puede ser reutilizado en diferentes sistemas, así como también puede ser reemplazado por otro que sea capaz de proveer la misma funcionalidad.

El desarrollo de software basado en componentes permite reutilizar piezas de código para diversas tareas, que conlleva diversos beneficios como las mejoras a la calidad, reducción del ciclo de desarrollo y el mayor retorno sobre la inversión.

Un **componente** puede ser tan simple como un conversor Dolar-Euro o tan complejo como para manejar un sistema completo. Los componentes más complejos pueden ser divididos en partes más simples, subcomponentes, igualmente reutilizables y reemplazables.

Cuando se habla de **componentes gráficos**, se habla de un grupo de elementos gráficos que funcionan de manera conjunta. Estos pueden ser paneles completos, grupos de botones, agrupación de varios componentes o incluso vistas completas de la interfaz o ventanas personalizadas, como son los cuadros de dialogo, normalmente ya implementados en la biblioteca grafica.

1.2 Características:

Para poder catalogar un componente como tal ha de cumplir una serie de características. Un componente puede ser:

- **Independiente de la plataforma:** hardware, software o sistema operativo.
- **Identificable:** para poder clasificarlo y acceder a sus servicios.
- **Autocontenido**, es decir, no debe requerir la utilización de otros componentes para su correcto funcionamiento.
- **Reemplazable**, por nuevas versiones u otro componente que realice lo mismo.
- **Accesible solamente a través de su interfaz**. Una interfaz define las operaciones (servicios o responsabilidades) que el componente puede realizar.
- **Invariante.** Sus servicios no deben variar, su implementación puede.
- **Documentado apropiadamente**, para facilitar su utilización.
- **Genérico**, sus servicios han de ser lo más reutilizables posible.

- **Reutilizable dinámicamente**, cargado en tiempo de ejecución en una aplicación.
- **Distribuido como paquete,** que almacena todos los elementos que los componen.

1.3 Puntos fuertes y débiles:

Como todo, los componentes tienen ciertas características que lo hacen útiles, pero el desarrollo de estos implica complicación extra, a veces innecesaria.

Ventajas:

- Reutilizacion del software. Nos lleva a alcanzar un mayor nivel de reutilización de software.
- Agiliza las pruebas. Permite hacer pruebas a cada componente antes de probar el conjunto completo de componentes ensamblados.
- Simplifica el mantenimiento del sistema. Cuando existe un débil acoplamiento entre componentes, el desarrollador es libre de actualizar y/o agregar componentes según sea necesario, sin afectar otras partes del sistema.
- Mayor cálida. Dado que un componente puede ser construido y luego mejorado continuamente por un experto u organización, la calidad de una aplicación basada en componentes mejorara con el paso del tiempo.
- Funcionalidad mejorada. Para usar un que contenga una pieza de funcionalidad, solo se necesita entender su naturaleza, mas no sus detalles internos.

Desventajas

Las limitaciones en el desarrollo de componentes son las siguientes:

- Solo es habitual su aplicación en algunos campos como las interfaces graficas.
- No siempre es posible encontrar componentes adecuados para cada aplicación.
- No todas las plataformas están preparadas para ejecutar todos los componentes.
- No hay procesos de certificación para garantizar la calidad de los componentes.
- La falta de estándares hacen difícil la localización, adaptación y reusabilidad.

2 ¿Qué es la serialización y para qué sirve?

La serialización es el proceso de convertir el estado de un objeto en un formato que se pueda almacenar o transportar. El complemento de serialización es deserialización, que convierte una secuencia en un objeto. Juntos, estos procesos permiten almacenar y transferir datos.

Para que un programa java pueda convertir un objeto en un montón de bytes y pueda luego recuperarlo, el objeto necesita ser **Serializable**. Al poder convertir el objeto a bytes, ese objeto se puede enviar a través de red, guardarlo en un fichero, y después reconstruirlo al otra lado de la red, leerlo del fichero

Para que un objeto sea serializable basta con que implemente la interfaz **Serializable**. Como la interfaz Serializable no tiene métodos, es muy sencillo implementarla, basta con un implements Serializable y nada más.

Por ejemplo, la clase Datos siguiente es Serializable y java sabe perfectamente enviarla o recibirla por red, a través de socket o de rmi. También java sabe escribirla en un fichero o reconstruirla a partir del fichero.

```
public class Datos implements Serializable
{
  public int a;
  public String b;
  public char c;
}
```

Si dentro de la clase hay atributos que son otras clases, éstos a su vez también deben ser Serializable. Con los tipos de java (String, Integer, etc.) no hay problema porque lo son. Si ponemos como atributos nuestras propias clases, éstas a su vez deben implementar Serializable. Por ejemplo

```
/* Esta clase es Serializable porque implementa Serializable y todos sus
 *  campos son Serializable, incluido "Datos f;"
 */
public class DatoGordo implements Serializable
{
  public int d;
  public Integer e;
  Datos f;
}
```

Además vamos a necesitar dos clases ObjectOutputStream y ObjectInputStream, para distribuir y recuperar un objeto. (Ir a la API y echar un vistazo de estas clases y de la interfaz serializable)

Pasos: Almacenar el objeto empleado

```
class Empleado implements Serializable{
public Empleado(String n, double s, int agno, int mes, int dia){

        nombre=n;

        sueldo=s;

        GregorianCalendar calendario=new GregorianCalendar(agno, mes-1,dia);

        fechaContrato=calendario.getTime();

}


public String getNombre() {
        return nombre;
}

public double getSueldo() {
        return sueldo;
}

public Date getFechaContrato() {
        return fechaContrato;
}
```

```
        public void subirSueldo(double porcentaje){
                double aumento=sueldo*porcentaje/100;
                sueldo+=aumento;
        }
        public String toString(){
                return "El Nombre es =" + nombre + ",y su sueldo es =" + sueldo + ", fecha de contrato="
+ fechaContrato;
        }
        private String nombre;
        private double sueldo;
        private Date fechaContrato;
}
}
```

1. **Crear un flujo de salida:**

   ObjectOutputStream flujoSalida=new ObjectOutputStream(new FileOutputStream(C:\salida\objeto.dat);

2. **Escribir ese flujo de salida:**

   flujoSalida.writeObject(persona)

3. **Crear una nueva clase Recuperando**

```
public class Recuperando
 {
        public static void main(String[] args){
                ObjectInputStream flujoEntrada=new ObjectInputStream(new FileInputS-
tream(C:\salida\objeto.dat)
                Empleado[] entradaObjeto=(Empleado[])flujoEntrada.readObject();
                flujoEntrada.close();
        }
        }
```

3 Asociación de aciones a eventos

La funcionalidad de un componente viene definida por las acciones que puede realizar definidas en sus métodos y no solo eso, también 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 pro-

cesar ejecutando alguna función. Pero no solo eso, un componete puede también lanzar un evento cuando sea necesario y que su tratamiento se realice en otro objeto.

Los eventos que lanza un componete, se pueden usar para programar la realización de acciones.

Para que el componente pueda reconocer el evento y responder ante el tendremos que:

- 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.
- Definir dos operaciones, para añadir y eliminar manejadores. Si queremos tener más de un manejador para el evento tendremos que almacenar internamente estos manejadores en una estructura de datos como un ArrayList o LinkedList

4 Ejemplo

La clase *Asalariado* tiene un constructor por defecto, que asigna un valor inicial de 20 al miembro dato *sueldo*. Sueldo es una propiedad ya que tiene asociados dos métodos que empiezan por **set** y **get**.

Para notificar un cambio en dicha propiedad necesitamos llevar a cabo las siguientes tareas:

1. Crear una clase que defina un suceso (event) personalizado, denominada *XXXEvent*
2. Crear un interface denominado *XXXListener*, que declare los métodos que los objetos (listeners) a los que se le notifican el cambio en dicha propiedad, precisen implementar.
3. Crear un array (vector) que contenga la lista de objetos (listeners) interesados en el cambio en el valor de dicha propiedad.
4. Definir dos funciones denominadas *addXXXListener* y *removeXXXListener*, que añadan o eliminen objetos de dicha lista.

Vamos a estudiar detalladamente cada uno de los pasos:

La clase que define un suceso

Un suceso (event) es un objeto que indica que algo ha sucedido. Puede ser que el usuario haya movido el ratón, que un paquete de datos haya llegado a través de la red, etc. Cuando algo sucede, se ha de realizar alguna acción, por ejemplo, dibujar en la superficie del applet cuando se mueve el ratón, imprimir en la pantalla la información que ha llegado, etc.

La clase que define nuestro suceso (event) personalizado, que denominamos *SalarioEvent*, deriva de *EventObject*. Dicha clase tiene dos miembros dato, el sueldo que cobraba antes *anteSueldo*, y el sueldo que cobra ahora, *nuevoSueldo*. La clase base *EventObject* precisa conocer la fuente de los sucesos, que se le pasa en el primer parámetro del constructor luego, inicializa los dos miembros dato, que se pueden declarar **private** o **protected**, según convenga.

La clase define dos funciones miembro, *getNuevoSueldo* y *getAnteSueldo*, que permiten conocer los valores que guardan los dos miembros dato.

```
import java.util.*;


public class SalarioEvent extends EventObject {
```

```
  protected int anteSueldo, nuevoSueldo;
  public SalarioEvent(Object fuente, int anterior, int nuevo) {
    super(fuente);
    nuevoSueldo=nuevo;
    anteSueldo=anterior;
  }
  public int getNuevoSueldo(){ return nuevoSueldo;}
  public int getAnteSueldo(){ return anteSueldo;}
}
```

El interface XXXListener

Un interface es un grupo de métodos que implementan varias clases independientemente de su relación jerárquica, es decir, de que estén o no en una jerarquía.

La clase cuyos objetos (listeners) están interesados en el cambio en el valor de la propiedad ligada, ha de implementar un interface que se ha denominado *SalarioListener*. Dicho interface declara una única función *enteradoCambioSueldo* que ha de ser definida por la clase que implemente el interface.

```
import java.util.*;


public interface SalarioListener extends EventListener {
  public void enteradoCambioSueldo(EventObject e);
}
```

La fuente de los sucesos (events)

Un objeto que está interesado en recibir sucesos (events) se denomina *event listener*. Un objeto que que produce los sucesos se llama *event source*, el cual mantiene una lista *salarioListeners* (objeto de la clase *Vector*) de objetos que están interesados en recibir sucesos y proporciona dos métodos para añadir *addSalarioListener* o eliminar *removeSalarioListener* dichos objetos de la lista.

```
public class Asalariado{
  private Vector salarioListeners=new Vector();


  public synchronized void addSalarioListener(SalarioListener listener){
      salarioListeners.addElement(listener);
  }


  public synchronized void removeSalarioListener(SalarioListener listener){
```

```
        salarioListeners.removeElement(listener);
    }
//...
}
```

Cada vez que se produce un cambio en el valor de la propiedad Sueldo, se ha de notificar dicho cambio a los objetos interesados que se guardan en el vector *salarioListeners.*

La función miembro o método que cambia la propiedad se denomina *setSueldo*. La tarea de dicha función como hemos visto anteriormente es la de actualizar el miembro dato *sueldo*, pero también tiene otras tareas como son las de crear un objeto de la clase *SalarioEvent* y notificar a los objetos interesados (listeners) de dicho cambio llamando a la función miembro *notificarCambio* y pasándole en su único argumento el objeto *event* creado.

Para crear un objeto *event* de la clase *SalarioEvent*, se precisa pasar al constructor tres datos: el objeto fuente de los sucesos, **this**, el sueldo que cobraba antes, *anteSueldo* y el sueldo que cobra ahora, *nuevoSueldo*.

```
 public void setSueldo(int nuevoSueldo){
   int anteSueldo=sueldo;
   sueldo=nuevoSueldo;
   if(anteSueldo!=nuevoSueldo){
      SalarioEvent event=new SalarioEvent(this, anteSueldo, nuevoSueldo);
      notificarCambio(event);
   }
 }
```

Se define la función *notificarCambio*, para notificar el cambio en la propiedad Sueldo a los objetos (listeners) que están interesados en cambio de dicha propiedad y que se guardan en el vector *salarioListeners*,. En dicha función, se crea una copia del vector *salarioListeners* y se guarda en la variable local *lista* de la clase *Vector*. La palabra clave **synchronized** evita que varios procesos ligeros o threads puedan acceder simultáneamente a la misma lista mientras se efectúa el proceso de copia.

```
   Vector lista;
   synchronized(this){
      lista=(Vector)salarioListeners.clone();
   }
```

Finalmente, todos los objetos (listeners) interesados y que se guardan en el objeto *lista*, llaman a la función miembro *enteradoCambioSueldo*, ya que la clase que describe a dichos objetos, como veremos más adelante, implementa el interface *SalarioListener*. En el estudio de la clase *Vector* vimos como se accedía a cada uno de sus elementos.

```
   for(int i=0; i<lista.size(); i++){
      SalarioListener listener=(SalarioListener)lista.elementAt(i);
      listener.enteradoCambioSueldo(event);
   }
```

El código completo de la clase *Asalariado* es el siguiente

```
import java.beans.*;
import java.util.*;

public class Asalariado{
  private Vector salarioListeners=new Vector();
  private int sueldo;

  public Asalariado() {
    sueldo=20;
  }

  public void setSueldo(int nuevoSueldo){
    int anteSueldo=sueldo;
    sueldo=nuevoSueldo;
    if(anteSueldo!=nuevoSueldo){
        SalarioEvent event=new SalarioEvent(this, anteSueldo, nuevoSueldo);
        notificarCambio(event);
    }
  }
  public int getSalario(){
    return sueldo;
  }

  public synchronized void addSalarioListener(SalarioListener listener){
        salarioListeners.addElement(listener);
  }
  public synchronized void removeSalarioListener(SalarioListener listener){
        salarioListeners.removeElement(listener);
  }

  private void notificarCambio(SalarioEvent event){
    Vector lista;
```

```
        synchronized(this){
            lista=(Vector)salarioListeners.clone();
        }
        for(int i=0; i<lista.size(); i++){
            SalarioListener listener=(SalarioListener)lista.elementAt(i);
            listener.enteradoCambioSueldo(event);
        }
    }
}
```

Los objetos (listeners) interesados

La clase *Hacienda* que describe los objetos que están interesados en el cambio en el valor de la propiedad Sueldo, han de implementar el interface *SalarioListener* y definir la función *enteradoCambioSueldo*. Definimos una clase denominada *Hacienda* cuyos objetos (los funcionarios inspectores de hacienda) están interesados en el cambio de sueldo de los empleados.

```
public class Hacienda implements SalarioListener{


  public Hacienda() {
  }
  public void enteradoCambioSueldo(EventObject ev){
    if(ev instanceof SalarioEvent){
        SalarioEvent event=(SalarioEvent)ev;
        System.out.println("Hacienda: nuevo sueldo    "+event.getNuevoSueldo());
        System.out.println("Hacienda: sueldo anterior "+event.getAnteSueldo());
    }
  }
}
```

Como la función *enteradoCambioSueldo* recibe un objeto *ev* de la clase *SalarioEvent*, podemos extraer mediante las funciones miembro que se definen en dicha clase toda la información relativa al suceso: el sueldo que cobraba antes el empleado y el sueldo que cobra ahora. Esta información la obtenemos llamando a las funciones miembro *getNuevoSueldo* y *getAnteSueldo*. Con esta información el funcionario de Hacienda puede calcular las nuevas retenciones, impuestos, etc. En este caso, se limita, afortunadamente, a mostrar en la pantalla el sueldo anterior y el nuevo sueldo.

Vinculación entre la fuente de sucesos y los objetos (listeners) interesados

Para probar las clases *Asalariado* y *Hacienda* y comprobar como un objeto de la primera clase notifica el cambio en el valor de una de sus propiedades a un objeto de la segunda clase, creamos una aplicación.

IES de Teis
Avda. de Galicia, 101
36216 – Vigo
Tfno: 886 12 04 64
e-mail: ies.teis@edu.xunta.es
http://www.iesteis.es

En dicha aplicación, se crean dos objetos uno de cada una de las clases, llamando a su constructor por defecto o explícito según se requiera.

```
Hacienda funcionario1=new Hacienda();

Asalariado empleado=new Asalariado();
```

La vinculación entre el objeto fuente, *empleado*, y el objeto interesado en conocer el cambio en el valor de una de sus propiedades, *funcionario1* se realiza mediante la siguiente sentencia.

```
empleado.addSalarioListener(funcionario1);
```

El objeto *funcionario1* se añade a la lista (vector) de objetos interesados en conocer el nuevo sueldo del empleado. Podemos poner más sentencias similares, para que más funcionarios de Hacienda sean notificados de dicho cambio. También podemos crear otra clase que se llame por ejemplo *Familia*, que implemente el interface *SalarioListener* y defina la función *enteradoCambioSueldo*. Creamos objetos de esta clase, la mujer, los hijos, etc, y los añadimos mediante *addSalarioListener* a la lista de personas (listeners) interesados en conocer la noticia.

Cuando escribimos la sentencia

```
empleado.setSueldo(50);
```

1. El objeto *empleado* llama a la función *setSueldo* que cambia la propiedad.
2. En el cuerpo de *setSueldo*, se comprueba que hay un cambio en el valor del miembro dato *sueldo*.
3. Se llama a la función miembro *notificarCambio* para dar a conocer a los objetos interesados que se guardan en el vector *salarioListeners* el cambio en el valor de dicha propiedad.
4. En el cuerpo de *notificarCambio*, los objetos (listeners) interesados llaman a la función *enteradoCambioSueldo*.
5. Los objetos interesados que pueden ser de la misma clase o de distinta clase, siempre que implementen el inteface *SalarioListener*, realizan en el cuerpo de la función *enteradoCambioSueldo* las tareas, no siempre oportunas, con la información que se le proporciona a través del objeto *ev* de la clase *SalarioEvent*.