Tarea para DI03

Juan A. García Muelas

dic.2021

Los chicos de BK están aprendiendo a hacer **componente Java** usando la plataforma NetBeans. Ahora están intentando **crear un reloj digital** que poder insertar en cualquier interfaz, el reloj debe tener al menos las siguientes **características**:

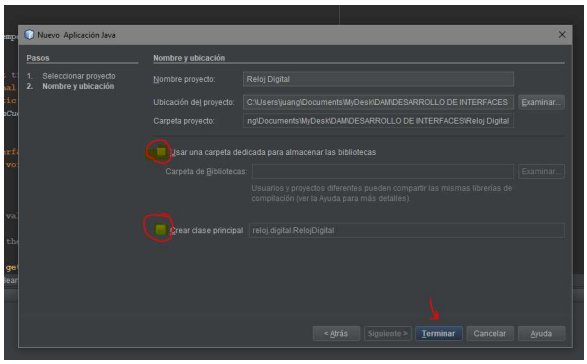
- ✓ Una **propiedad booleana** para indicar si el **formato** es de **12** o **24** horas.
- ✓ Una **propiedad booleana** para indicar si queremos **activar** una **alarma**. El funcionamiento de la alarma consistirá en que se podrá configurar el componente para que a una determinada hora nos **muestre un mensaje**.
- ✓ Dos propiedades para determinar la **hora** y **minuto** para el cual queremos programar la **alarma**. Ambas propiedades serán de **tipo entero**.
- ✓ Una **propiedad** para configurar el **mensaje de texto**, que queremos que se muestre, cuando se produzca el salto de la alarma. Esta propiedad será de tipo texto (`String`).
- ✓ **Función de alarma**, si se programa a una hora, debe **generar un evento** cuando se llegue a esa hora.

Tendrás que crear un formulario de prueba en el que añadas el reloj digital, modifiques el formato de visionado y añadas una alarma para probar que funciona.

1. Creación del componente.

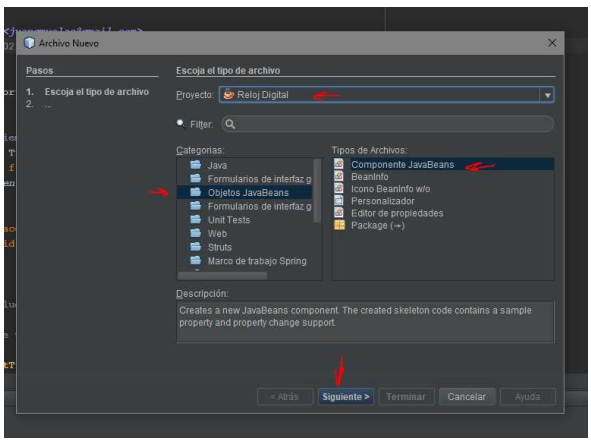
Para la resolución de esta tarea voy a tomar como ejemplo base el facilitado durante el estudio del tema y que repliqué para poder comprender y asimilar lo explicado en el mismo.

Lo primero es crear un nuevo proyecto en **NetBeans**, sin clase principal, que vamos a llamar **`Reloj Digital`**.



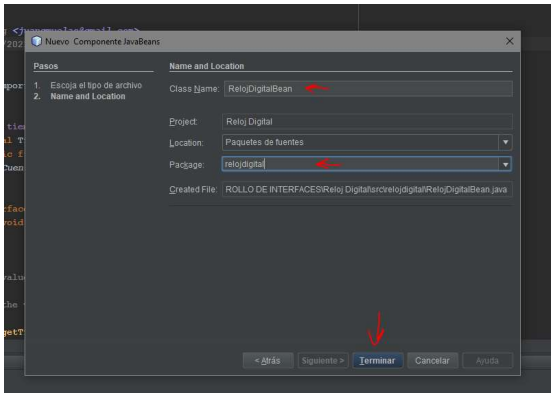
Creación de proyecto en NetBeans.

Creado el proyecto, lo siguiente es generar nuestro componente, mediante un nuevo archivo que seleccionamos mediante **`Categorías: Objetos JavaBeans`**, y **`tipo: Componente JavaBeans`**.



Creación Componente.

Se le elige un nombre (**RelojDigitalBean**) y lo creamos en un paquete generado al efecto, llamado **relojdigital**.



Creación Componente.

Nos crea un nuevo archivo (con nuestro componente ya como **Serializable**), con un código de ejemplo, que podemos borrar dejando su estructura básica (incluido el constructor vacío, necesario para nuestro componente) y personalizarlo.

```
/**
 *
 * @author juang <juangmuelas@gmail.com>
 * @since 30/12/2021
 * @version 1
 */

public class RelojDigitalBean implements Serializable {

//    private PropertyChangeSupport propertySupport;
    /**
     * Constructor vacio
     */
    public RelojDigitalBean() {
//        propertySupport = new PropertyChangeSupport(this);
    }

} //fin clase RelojDigitalBean
```

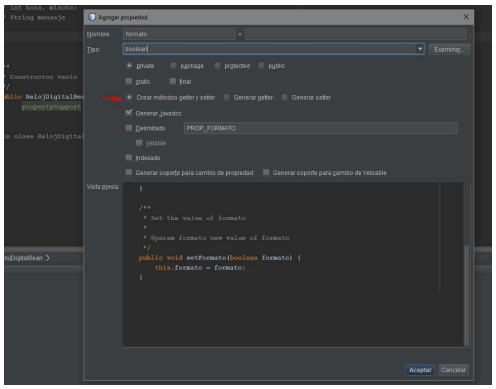
Detalle clase RelojDigitalBean y constructor vacío.

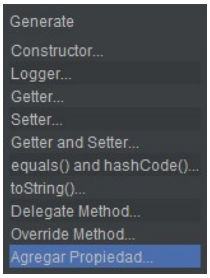
2. Añadir propiedades.

El siguiente paso es crear las propiedades necesarias, con la ayuda del menú contextual, y añadiendo los **getters** y **setters**.

Según el enunciado, creamos Un **booleano formato**, para discriminar entre 12 o 24 horas, un **booleano activar** para la alarma, dos enteros (**int**) **hora** y **minuto** y un **String** para el **mensaje**.

Nos ayudamos del menú contextual o de **Alt+Ins** para acceder a **Agregar propiedad** y generarlas:





Creación de propiedades.

Como se puede ver, se crean como privados para que no sean visibles desde fuera de la clase que implementa nuestro componente. Sin embargo, los métodos son públicos para formar parte de su interfaz.

```
    private boolean formato;
    private boolean activar;
    private int hora;
    private int minuto;
    private String mensaje;

    /**
     * getters y setters generados
     */

    /**
     * Get the value of formato
     *
     * @return the value of formato
     */
    public boolean isFormato() {
        return formato;
    }

    /**
     * Set the value of formato
     *
     * @param formato new value of formato
     */
    public void setFormato(boolean formato) {
        this.formato = formato;
    }
```

Detalle getters y setters.

La tarea nos pide mostrar un reloj, por lo que nuestro componente debe "pintarse" en la ventana (**`frame`**) donde queramos llamarlo. Por ello, deberá extender nuestra clase de **`JLabel`**.

Para manejar y mostrar esos datos, pensé primero en utilizar la clase **`LocalDate`** para las horas y minutos, perteneciente al paquete **`java.time`**, pero parece más lógico utilizar objetos de la clase **`Date`** que nos recogen la fecha de nuestra alarma y hacen más escalable nuestro proyecto, y **`SimpledateFormat`**, por lo que se crean también las propiedades **`format12`** y **`format24`** de tipo **`SimpleDateFormat`**.

Siguiendo el ejemplo del tema, también se crea una propiedad de tipo **`Timer`** llamada **`t`**.

Este lo vamos a utilizar para inicializar cada segundo (1000 milisegundos) nuestro reloj en el constructor.

```
207
208     /**
209      * Constructor vacio
210      */
211     public RelojDigitalBean() {
212         t = new Timer (1000, this); //inicializar objeto Timer.
213         t.start();
214     }
215
```

Detalle constructor.

3. Implementar alarma.

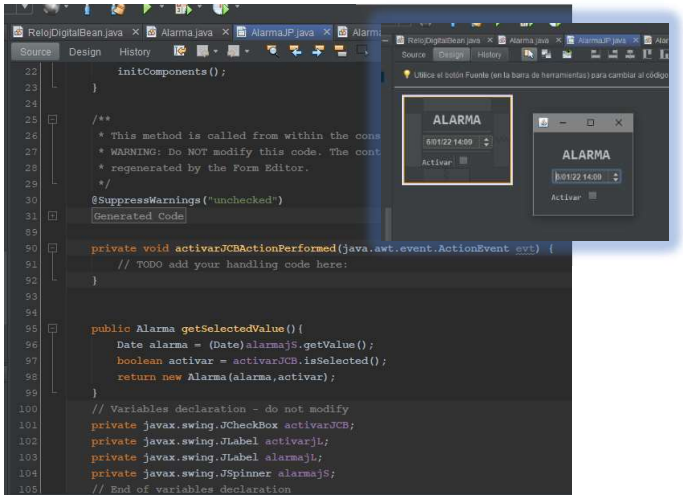
Para la alarma, pensé en recoger los datos de manera similar a como vamos a recoger el formato o el mensaje y utilizar sus métodos para disparar el evento.

Sin embargo, me ha parecido más correcto separar ese comportamiento desde una nueva clase, dentro de nuestro **`package`** que también deberá implementarse para serialización, y que tendrá las propiedades **`horaAlarma`** de tipo **`Date`** y un **`boolean`** para **`activar`**.

```
public class Alarma implements Serializable {
    private Date horaAlarma;
    private boolean activar;

    public Alarma(Date horaAlarma, boolean activar) {
        this.horaAlarma = horaAlarma;
        this.activar = activar;
    }

    public Date getHoraAlarma() {
        return horaAlarma;
    }

    public void setHoraAlarma(Date horaAlarma) {
        this.horaAlarma = horaAlarma;
    }

    public boolean isActivar() {
```

Detalle clase Alarma.

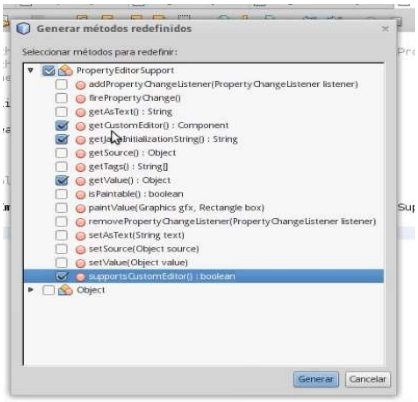
Pudiendo trabajar con objetos que dependan de nuestra nueva clase, el siguiente paso es crear una nueva clase llamada **AlarmaJP** y de tipo **JPanel** (no podemos incluir un frame dentro de otro frame), donde añadimos un **JSpinner** de tipo **Date** (de ahí el trabajar con el paquete **Util.Date** en vez de **time** para la alarma), donde el usuario podrá seleccionar el momento en el que se dispare el evento alarma y un **JCheckBox** para seleccionar su activación.



Clase AlarmaJP y detalle de vista de diseño.

4. Crear el editor de propiedades.

Tras recoger los datos, y buscando algo más de información sobre como proseguir con la creación del componente (esta parte es más confusa a mi entender dentro del temario), encontré un recurso de vídeo (Enlace: https://youtu.be/rDVAWXei8pE) donde lo abordaban y he seguido sus pasos para crear un editor personalizado.



Detalle generación métodos en video.

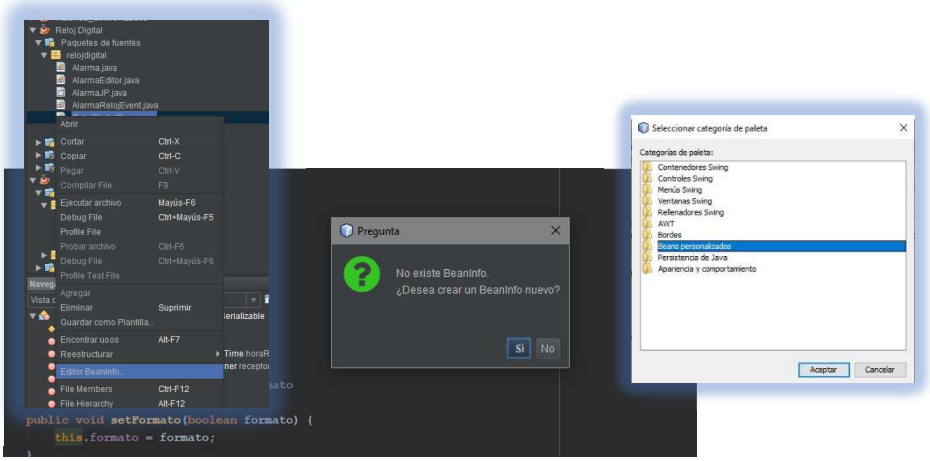
Se crea una nueva clase llamada **AlarmaEditor**, que extienda de **PropertyEditorSupport**, en la que creamos un objeto de la clase **AlarmaJP** y se añaden los métodos recomendados en el vídeo (**supportsCustomEditor()** ,para decirle si permite un editor personalizado, y para llamar al panel y sus valores están **getCustomEditor()**, **getValue()** y por último recogiendo los valores y devolviendo la instrucción de crear la alarma con ellos, **getJavaInitializationString()**).

```
public class AlarmaEditor extends PropertyEditorSupport {
    private AlarmaJP alarmaJp = new AlarmaJP();

    @Override
    public boolean supportsCustomEditor(){
    return true;//al estar personalizado devuelve true (por defecto es false)
    }

    @Override
    public Component getCustomEditor(){
        return alarmaJp;
    }

    @Override
    public Object getValue(){
        return alarmaJp.getSelectedValue();
    }

    @Override
    public String getJavaInitializationString(){
        Date horaAlarma = alarmaJp.getSelectedValue().getHoraAlarma();
        boolean activar = alarmaJp.getSelectedValue().isActivar();
        return "new relojdigital.Alarma(new java.util.Date("
            + horaAlarma.getTime() + "l)," + activar + ")";
    }
}//fin clase AlarmaEditor
```

Clase AlarmaEditor.

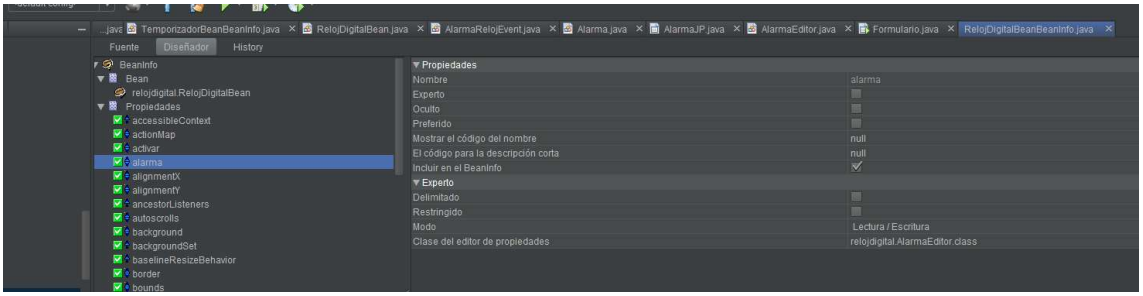
Creado el editor personalizado, ya podemos asociar las propiedades de nuestro componente a un editor de propiedades, que será el que nos muestre las opciones al utilizar el componente en el futuro.

Para ello, seleccionamos la clase **RelojDigitalBean** y con el menú contextual elegimos **Editar BeanInfo**. Al ser la primera vez, nos dirá que no existe y si deseamos crear uno. Tras aceptar, pide la categoría donde lo queremos crear, y le indicamos **Beans personalizados**. Nos crea un archivo con el mismo nombre de nuestra clase, pero acabado en "BeanInfo".



Creación BeanInfo del componente.

En esta clase, es importante buscar la propiedad por la que hemos creado nuestro editor de propiedades personalizado (en nuestro caso, **alarma**) e incluir, en el apartado **Clase del editor de propiedades**, el paquete y la clase (con **.class**) para que funcione correctamente. Sino lo indicamos nuestro componente fallará, pero no nos marcará error alguno.



5. Gestión de los eventos.

Teniendo definido nuestro componente, vuelvo al temario, donde nos indica que los componente para implementar la gestión de eventos necesitan:

1) Una clase que implemente los eventos y herede de **java.util.EventObject**. Con esas pautas y siguiendo el ejemplo del temario, creo la clase **AlarmaRelojEvent**.

```
4  package relojdigital;
5
6  import java.util.EventObject;
7
8  /**
9   *
10  * @author juang <juangmuelas@gmail.com>
11  * @since 30/12/2021
12  * @version 1
13  */
14 public class AlarmaRelojEvent extends EventObject {
15
16     public AlarmaRelojEvent(Object source){
17         super(source);
18     }
19 }//fin clase AlarmaRelojEvent
20
```

Clase AlarmaRelojEvent para el manejo de eventos.

2) Definir una interfaz que defina los métodos a usar que implemente de **`java.util.EventListener`**. Para ello, vuelvo a la clase **`ReloDigitalBean`** y añado la interfaz siguiendo el ejemplo.

```
    private Alarma alarma;

    public interface AlarmaRelojListener extends EventListener{
        public void capturarAlarmaReloj(AlarmaRelojEvent ev);
    }
```

Definición de interfaz en la clase de nuestro componente.

3) Tener métodos que permitan añadir y eliminar oyentes. Sin movernos de archivo, añadimos los métodos siguiendo el ejemplo planteado.

```
    public void addAlarmaRelojListener(AlarmaRelojListener receptor){
        this.receptor = receptor;
    }

    public void removeAlarmaRelojListener(AlarmaRelojListener receptor){
        this.receptor=null;
    }

} //fin clase RelojDigitalBean
```

Métodos para añadir y eliminar oyentes.

4) Implementar el método que lanza el evento. En esta parte, por un lado vamos a tener la parte que se encarga de recoger los datos y crear nuestra alarma. Para ello, se crea el método **`activarAlarma`**, que recogerá la hora actual de nuestro reloj mediante la propiedad **`horaReloj`** de tipo **`LocalDateTime`**, y la facilitada por el usuario. Si los datos coinciden, devolverá verdadero para poder activar el evento.

```
public boolean activarAlarma(Date horaAlarma,LocalDateTime horaReloj){

    /**
     * Creo una instancia de Calendar para manejar el Date
     * recibido por la alarma, pues el Spinner no deja
     * castear los datos que recibe desde java.Util.Date a Time.
     * La hora actual si la recogemos con un objeto Time.
     */
    Calendar reloj = Calendar.getInstance();
    horaReloj = LocalDateTime.now();
    //Por seguimiento particular de ejecución recojo los datos
    System.out.println("hora reloj" + horaReloj);
    System.out.println("hora alarma" + horaAlarma);
    int hReloj = LocalDateTime.now().getHour();
    int mReloj = LocalDateTime.now().getMinute();
    int sReloj = LocalDateTime.now().getSecond();

    reloj.setTime(horaAlarma);
    hora = reloj.get(Calendar.HOUR_OF_DAY);
    minuto = reloj.get(Calendar.MINUTE);
    segundo = reloj.get(Calendar.SECOND);

    //Si los datos coinciden, retorna true para activar el evento
    if( hReloj == hora && mReloj == minuto && sReloj == segundo){
        return true;
    }else{
        return false;
    }
}//fin método activarAlarma
```

Método activarAlarma.

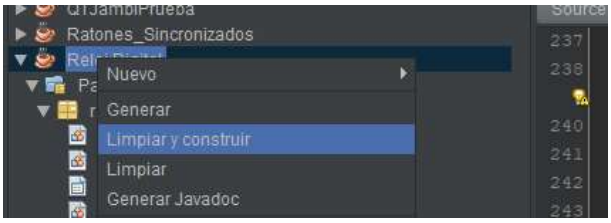
Por otro lado, tenemos nuestro **`actionPerformed`**, que primero, recoge la hora de nuestro reloj y según elija el usuario, se mostrará con un formato u otro, "repintando" el reloj tras la selección. Por último, revisará si tenemos una alarma activa y si además ha recibido un verdadero desde el método **`activarAlarma()`**. En ese caso, dispara el evento.

```
    @Override
    public void actionPerformed(ActionEvent e)
    {
        /**
         * Comprobamos hora y recogemos formato
         */
        horaReloj = LocalDateTime.now();
        if(formato){
            setText(format12.format(LocalDateTime.now()));
        }else{
            setText(format24.format(LocalDateTime.now()));
        }
        //hacemos que refresque lo mostrado
        repaint();

        //evitamos un nullpointerException con este primer if
        if(alarma != null){

            /**
             * Si la alarma está activada y recibimos datos
             * correctos desde el método, mandamos el evento
             */
            if(alarma.isActivar() && activarAlarma(alarma.getHoraAlarma(), horaReloj)){
            receptor.capturarAlarmaReloj(new AlarmaRelojEvent(this));
            }
        }
    }//fin del ActionPerformed
```

Detalle actionPerformed para el formato y alarma.

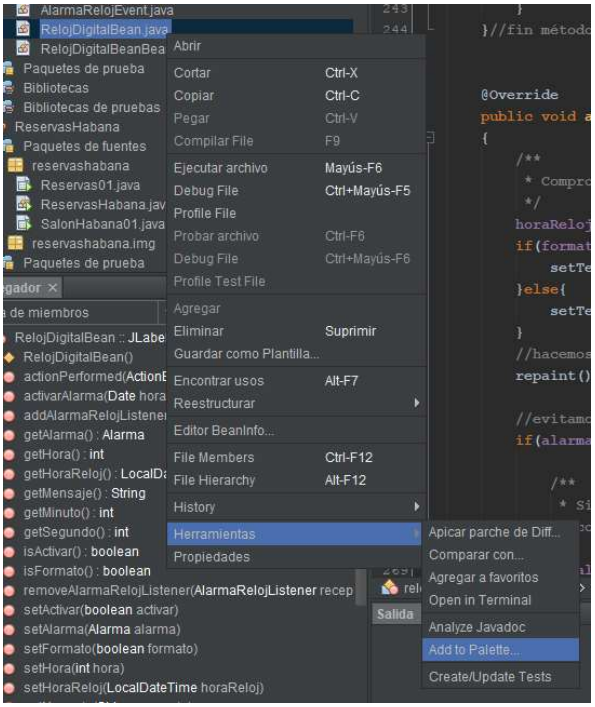
6. Uso del componente.

El siguiente paso, previo a su utilización es compilar nuestro proyecto, mediante la opción **`Limpiar y construir`**.



Compilar mediante Limpiar y construir.

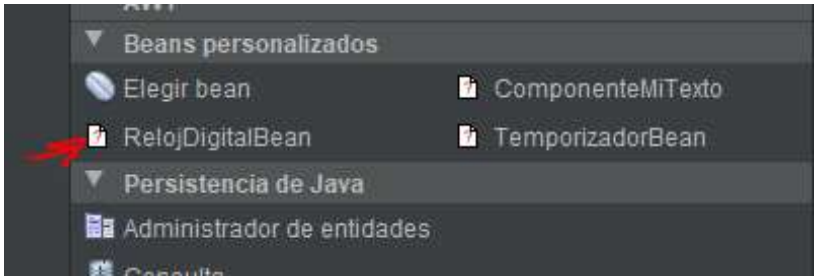
Para poder usarlo ahora en nuestros proyectos, debemos añadirlo a la **`paleta de componentes`**. Seleccionamos la clase en la que hemos definido nuestro componente y mediante el menú contextual, seleccionamos **`Herramientas>Añadir a la paleta`**.



Detalle Añadir a la paleta.

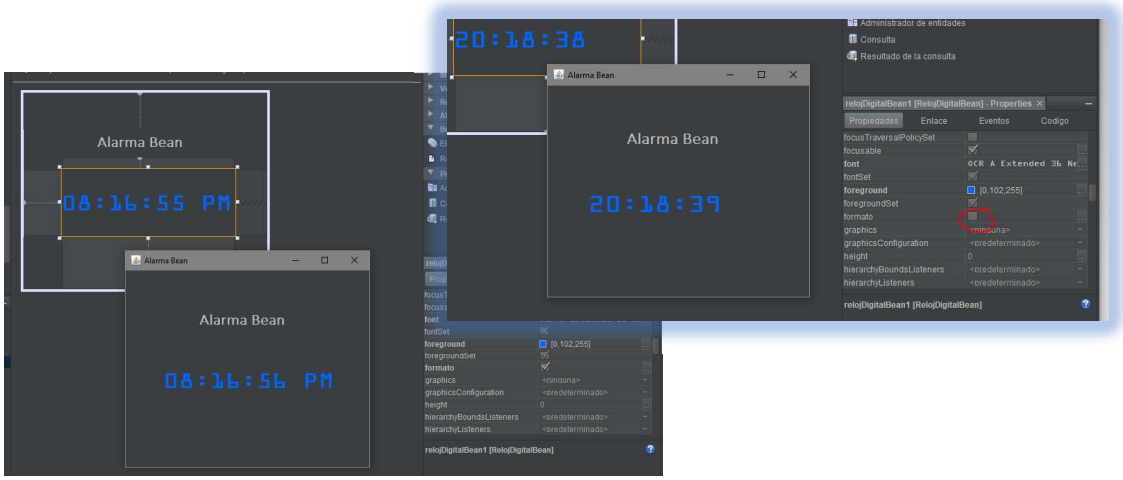
A partir de aquí, nuestro componente debería aparecer disponible.

Lo comprobamos abriendo el Proyecto creado para probar el ejemplo del temario y buscando en **`Beans personalizados`**.



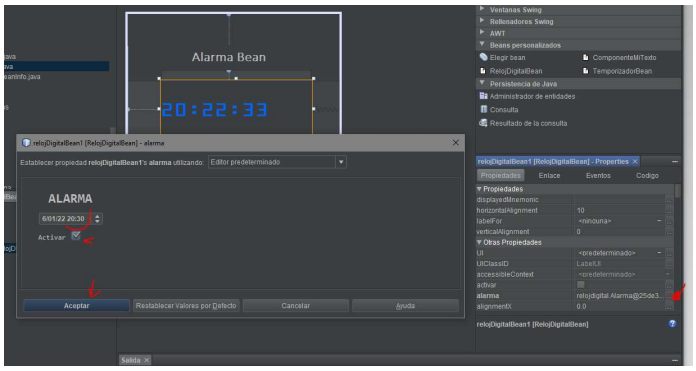
Beans personalizados con nuestro nuevo componente.

Lo arrastramos a la ventana del proyecto, y tras tenerlo ya disponible, podemos probar a personalizar un poco la apariencia para mejorar la experiencia de usuario y visualizar con **`formato`** de 12h o 24h según la selección.



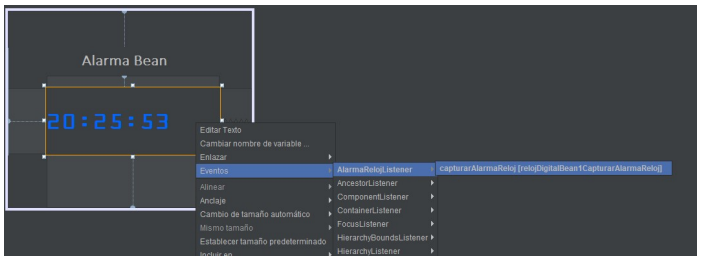
Detalle de personalizado de prueba componente.

Vamos a la propiedad **`alarma`** y hacemos click sobre los tres puntos. Se abre nuestro panel y seleccionamos una alarma.



Detalle panel alarma.

Para que tenga efecto generamos el evento:



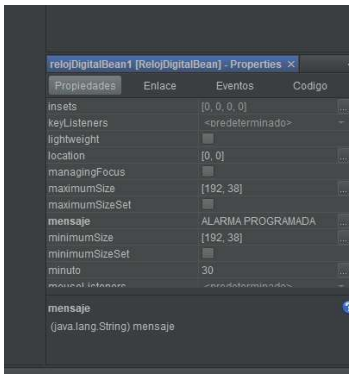
Detalle creación evento.

Como siempre, NetBeans crea de forma automática el método para nuestro evento. Añadimos como pide la tarea un cuadro de dialogo para que recoja el mensaje que queremos mostrar tras saltar la alarma, y que recibe desde la propiedad mensaje.

```
private void relojDigitalBean1CapturarAlarmaReloj(relojdigital.AlarmaRelojEvent evt) {
    // TODO add your handling code here:
    JOptionPane.showMessageDialog(null,relojDigitalBean1.getMensaje(), "Alarma", JOptionPane.INFORMATION_MESSAGE);
}

/**
 * @param args the command line arguments
```

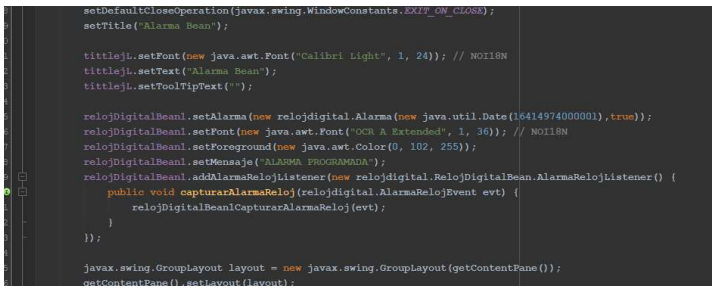
Detalle evento mensaje alarma.

Si en las propiedades accedemos al campo mensaje, podemos ahí añadir el texto.



Personalización mensaje.

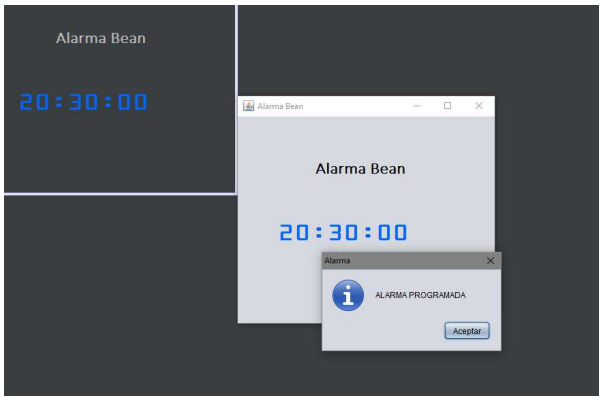
Si volvemos al código del proyecto, vemos como está generada la línea de código correctamente para la alarma, así como el resto de las acciones de personalización y mensaje creados.



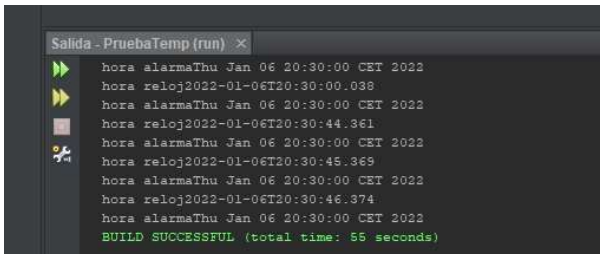
```
setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
setTitle("Alarma Bean");

tittlejL.setFont(new java.awt.Font("Calibri Light", 1, 24)); // NOI18N
tittlejL.setText("Alarma Bean");
tittlejL.setToolTipText("");

relojDigitalBean1.setAlarma(new relojdigital.Alarma(new java.util.Date(16414974000001),true));
relojDigitalBean1.setFont(new java.awt.Font("OCR A Extended", 1, 36)); // NOI18N
relojDigitalBean1.setForeground(new java.awt.Color(0, 102, 255));
relojDigitalBean1.setMensaje("ALARMA PROGRAMADA");
relojDigitalBean1.addAlarmaRelojListener(new relojdigital.RelojDigitalBean.AlarmaRelojListener() {
    public void capturarAlarmaReloj(relojdigital.AlarmaRelojEvent evt) {
        relojDigitalBean1CapturarAlarmaReloj(evt);
    }
});

javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
getContentPane().setLayout(layout);
```

Detalle generación de código de propiedades.

Tras guardar los cambios, ejecutamos y probamos el funcionamiento:



Aunque no sea visible en esta presentación, puede comprobarse en la salida de consola como refresca datos cada segundo.



Detalle actualización de hora cada segundo.