ACTIVIDAD GUIADA 14

El Modelo Vista-Controlador

Arquitectura Modelo-Vista-Controlador

El **modelo vista-controlador** (MVC) es un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de su interfaz de usuario y las comunicaciones. MVC propone la construcción de tres elementos diferentes que son:

- **El modelo** es la representación de la información con la cual el sistema opera, por tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (lógica de negocio).

  Envía a la vista aquella parte de la información que en cada momento se le solicita para que sea mostrada.

- **La vista** presenta el modelo (información y lógica de negocio) en un formato adecuado para interactuar (normalmente es la interfaz de usuario).
- **El controlador** es la parte encargada de responder a eventos (normalmente a acciones de usuario e invocar peticiones al modelo cuando se hace alguna solicitud de información (por ejemplo, editar un registro de una base de datos). También puede enviar comandos a su "vista" asociada si se solicita un cambio en la forma en que se presenta el modelo (por ejemplo, si nos desplazamos entre los distintos registros de una base de datos). Por tanto, podríamos decir que el controlador hace de **intermediario entre la vista y el modelo**.

Ejemplo: Creación de un contador.

Vamos a crear una pequeña aplicación que ilustre la utilización de la arquitectura MVC.

Será un contador como el que muestra la imagen, con dos botones *Arriba* y *Abajo.*

Al pulsar el botón *Arriba*, en la etiqueta central se incrementará su valor de la cuenta, mientras que al pulsar el botón *Abajo*, en la etiqueta se decrementará dicho valor. El valor no puede ser menor que cero en ningún caso.

El modelo

Dado que nuestra aplicación es un contador, vamos a desarrollar una clase que funcione como un contador, y que albergará una variable privada con el valor que en ese momento está seleccionado. Debemos implementar métodos que realicen operaciones sobre el contador:

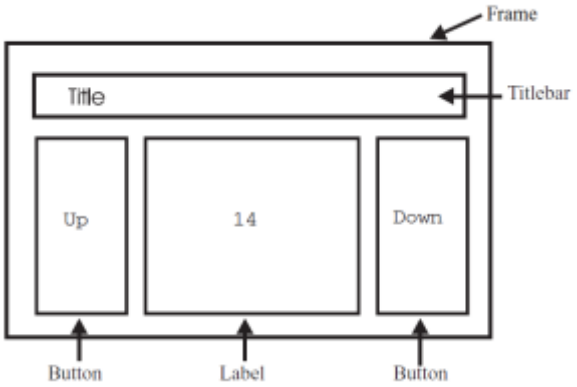
- Incrementar el contador
- Decrementar el contador
- Resetear el contador
- Obtener el valor

Un ejemplo de clase que podría modelizar nuestro contador es la siguiente (**`Contador`**):

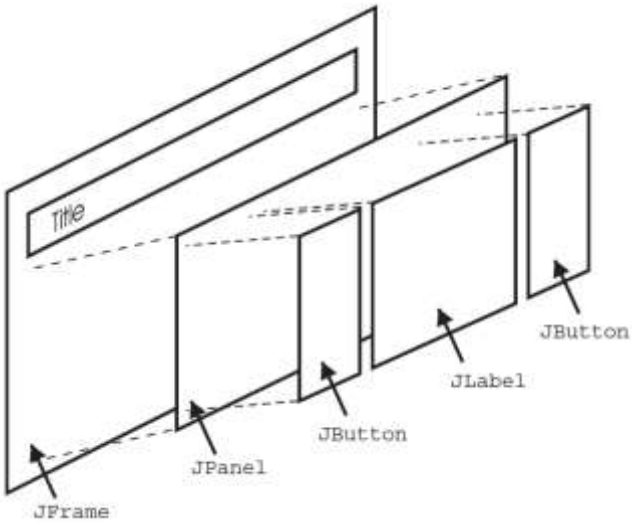
```
public class ContadorModelo {
    private int valor; //Almacena el valor del contador

    public ContadorModelo(){
        valor=0;
    }

    public void incrementar(){
        valor++;
    }

    public void decrementar(){
        if(valor>0)
            valor--;
    }

    public void reset(){
        valor=0;
    }

    public int getValor(){
        return valor;
    }
}
```

La vista del contador

Vamos a implementar una vista para nuestro contador, que tendrá la apariencia de la imagen:



En vez de añadir los elementos directamente al frame, vamos a crear un **`JPanel`** que posteriormente insertaremos en un **`JFrame`**. Para ello crearemos una clase que herede de **`JPanel`**, incluyendo en ella una etiqueta que mostrará el valor del contador y dos botones que se usarán para incrementar y decrementar el contador.



Para la disposición de pantalla, nos decantamos por un **`BorderLayout`**, colocando los botones Up y Down en los paneles Este y Oeste, y la etiqueta que muestra el contador en el panel central.

El centro del panel es la variable contador, que no es más que una instancia de **ContadorModelo** pero que es una variable **privada**, luego no es accesible desde el exterior.

Para dar acceso a las operaciones de incremento y decremento sobre el contador, creamos los métodos públicos **`incrementar()`** y **`decrementar()`**.

Para visualizar la ventana, crearemos la clase **`ContadorFrame`** colocando un **`ContadorPanel`** en su interior.

Cuando la aplicación arranca, la ventana aparece perfectamente construida, pero sin funcionalidad, pues aún necesitamos crear el **controlador** que se encargará de atender los eventos que envíen los botones.

```
import java.awt.BorderLayout;
import javax.swing.*;

public class ContadorPanel extends JPanel {
    private ContadorModelo contador;
    private JLabel lblCuenta;
    private JButton btnArriba, btnAbajo;

    public ContadorPanel(){
        contador=new ContadorModelo();

        BorderLayout admin=new BorderLayout();
        this.setLayout(admin);

        btnArriba=new JButton("Arriba");
        btnAbajo=new JButton("Abajo");
        lblCuenta=new JLabel("",JLabel.CENTER);
        actualizaEtiquetaCuenta();

        add(btnArriba,BorderLayout.WEST);
        add(btnAbajo,BorderLayout.EAST);
        add(lblCuenta,BorderLayout.CENTER);
    }

    private void actualizaEtiquetaCuenta(){
      lblCuenta.setText("Contador: "+contador.getValor());
    }

    public void incrementar(){
        contador.incrementar();
        actualizaEtiquetaCuenta();
    }

    public void decrementar(){
        contador.decrementar();
        actualizaEtiquetaCuenta();
    }

}
```

```
import javax.swing.JFrame;
import javax.swing.UIManager;

public class ContadorFrame extends JFrame {

    public ContadorFrame(String titulo){
        super(titulo);
        setSize(200,200);
        setLocation(200,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        ContadorPanel contadorPanel=new ContadorPanel();
        add(contadorPanel);

    }

   public static void main(String[] args){
        new ContadorFrame("Contador"). setVisible(true);
    }
}
```

El controlador para el contador

Ya tenemos una aplicación que muestra un contador, y tiene dentro de sí un modelo válido para éste. Sólo necesitamos que funcionen los botones de la interfaz gráfica para que la que la aplicación esté terminada.

Para finalizar la aplicación, debemos hacer que al pulsar los botones *Up* y *Down* se lancen los métodos **`incrementar()`** o **`decrementar()`** de la vista, que actualizan la vista y el propio modelo.

En los ejemplos realizados hasta ahora, implementábamos los escuchadores como parte de la propia interfaz o como clases internas.

En este caso, para **separar el controlador correctamente**, vamos a implementar el escuchador en una nueva clase:

```
public class ContadorListener implements ActionListener {

    private ContadorPanel contadorPanel;

    public ContadorListener(ContadorPanel contadorPanel){
        this.contadorPanel=contadorPanel;
    }

    @Override
    public void actionPerformed(ActionEvent ae) {
        String comando=ae.getActionCommand();
        if(comando.equals("Arriba"))
            contadorPanel.incrementar();
        else if(comando.equals("Abajo"))
            contadorPanel.decrementar();
        else
            System.err.println("ERROR: ActionCommand inesperado");
     }
}
```

Esto implica que insertemos las siguientes líneas en la parte de la vista (**`ContadorPanel`**):

```
        ContadorListener contadorListener=new ContadorListener(this);
        btnArriba.addActionListener(contadorListener);
        btnAbajo.addActionListener(contadorListener);
```

El problema es que seguimos necesitando acceder a la vista desde el método **`actionPerformed()`**. Por ello le pasamos el panel contador en el que nos encontramos al constructor del escuchador, que así podrá hacer uso de los métodos públicos implementados en el panel.

Dentro del citado método, usamos uso del comando del botón, (**`getActionCommand()`**)para distinguir qué botón se ha pulsado. Nótese que, en caso de no definir comando en el botón, se toma el texto del mismo. También se podría haber usado el método **`getSource().`**

Valoración

Aunque la aplicación del MVC aparentemente añade complejidad a nuestra aplicación, especialmente en el caso de aplicaciones de tamaño reducido como las que estamos implementando, hay una serie de **ventajas asociadas** a la aplicación de este patrón:

- **División del trabajo**. Al desarrollar una aplicación, la división en capas permite dividir el trabajo de manera eficaz entre distintos desarrolladores. Por ejemplo, aquellos con mayor experiencia en la construcción de interfaces gráficas podrían dedicarse a construir la interfaz y aquellos que conocen más a fondo el funcionamiento interno de la aplicación construirían el modelo de datos.
- **Reducción de errores.** Al tratarse las tres capas de manera independiente, se pueden construir y probar sin necesidad de que la aplicación completa esté construida. Esto permite realizar pruebas unitarias automatizadas y de forma más rápida, las cuales se hacen más complicadas si hay un alto acoplamiento entre las capas.
- **Construcción evolutiva de la interfaz.** La construcción de GUIs puede llegar a ser muy costosa en tiempo. Separando los datos de la interfaz, se potencia el hecho de poder construir aplicaciones con una interfaz sencilla de manera rápida (que permitan ir validando el resto del código de la aplicación) y, de forma paralela, poder ir construyendo otras clases de interfaz que pueden llevar más tiempo.
- **Posibilidad de elegir entre varias interfaces**. Al ser la construcción de la interfaz independiente de los datos, es posible que se construyan varias interfaces independientes que hagan uso de los mismos datos. Por ejemplo, sobre los mismos datos podría trabajar tanto una interfaz Swing como una interfaz Web: