Tema 5. Interfaces gráficas de usuario

Vicente Benjumea García

Programación Orientada a Objetos
Departamento de Lenguajes y Ciencias de la Computación.
E.T.S.I. Informática. Univ. de Málaga.

Tema 5. Interfaces gráficas de usuario

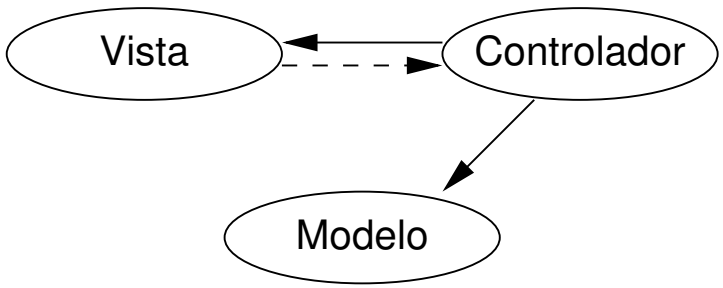
- Los paquetes `java.awt` y `javax.swing`
- Diseño de Interfaces Gráficas de Usuario (GUIs)
  - El patrón de diseño *Modelo-Vista-Controlador* (MVC)
- Conexión Vista-Controlador: el modelo de eventos
  - Interfaces para implementar controladores
- Un ejemplo Modelo-Vista-Controlador
- Construcción de Vistas
  - Componentes, Contenedores, Gestores de Esquemas
  - Un ejemplo completo
- Un ejemplo MVC

Los paquetes `java.awt` y `javax.swing`

- Permiten la construcción de **Interfaces Gráficas de Usuario** (GUIs).
- Inicialmente sólo existía **AWT** (Abstract Window Toolkit).
  - Por cada **componente** visible de AWT (botón, campo de texto, etc) existe otro en el **sistema operativo**, que es el que realmente realiza la representación.
  - Problemas con AWT:
    - AWT sólo define los **componentes comunes** que tienen todos los sistemas operativos.
    - Los componentes pueden tener **diferentes representaciones** (características y propiedades) en sistemas operativos distintos. Además la visualización es diferente.
- **SWING** se basa y extiende AWT, eliminando estos problemas.
  - Define todos los **componentes usuales** en GUIs.
  - La **representación** (características y propiedades) de cada componente es **común** en cualquier sistema operativo. Aunque la visualización puede ser diferente.
  - Necesita los paquetes: `java.awt`, `java.awt.event` y `javax.swing`
- Se verán las características más importantes para construir GUIs básicos.

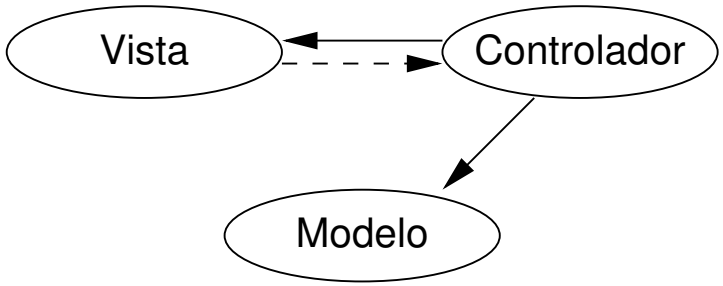
Diseño de Interfaces Gráficas de Usuario (GUIs)

- Usaremos el patrón de diseño **Modelo-Vista-Controlador** (MVC)
  - Modelo: representa la información y la lógica de la aplicación.
  - Vista: representa la interacción y la presentación de la información.
  - Controlador: representa la lógica de la interacción. Reacciona ante las acciones del usuario sobre la vista.
    - Consulta y actualiza la vista y el modelo.
- Uno de los objetivos fundamentales del patrón MVC es el de **independizar** los distintos componentes, de forma que sea posible modificar la vista, o incluso reemplazarla, sin que el controlador se vea afectado.



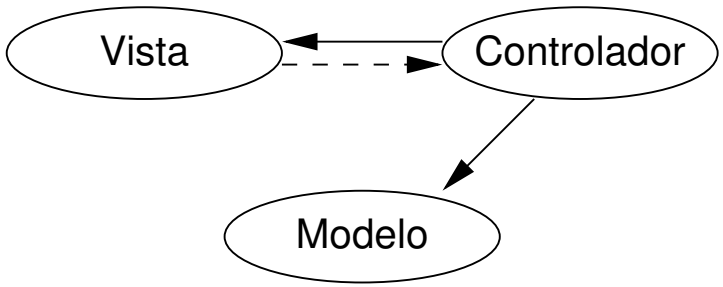
MVC: El modelo

- El **modelo** representa la **información** y la **lógica** de la aplicación para la que se realiza la interfaz gráfica.
  - El modelo es el componente principal del patrón MVC.
  - Puede ser desde una variable, hasta una gran cantidad de objetos.
  - Debe ser lo más **independiente** posible de la vista y del controlador.
  - Existe aunque no tengamos interfaz gráfica.
  - Puede ser, por ejemplo, una clase desarrollada en las prácticas que se han realizado durante el curso.



MVC: La vista

- La **vista** representa la **presentación** de la información y la **interacción** con el usuario.
  - Es un **panel** que contiene botones, áreas editables de texto, etiquetas, listas, etc.
  - **Interactúa** con el usuario a través de la interfaz gráfica (botones, texto, etc).
  - A partir de la interacción con el usuario, genera y **envía eventos** al controlador.
  - El controlador interactúa con la vista para **consultar** la información suministrada por el usuario.
  - El controlador interactúa con la vista para **actualizar** la información presentada al usuario.
  - En ciertas ocasiones, la vista también interactúa con el modelo.
- Para conseguir la independencia de los componentes, es conveniente **definir la vista como una interfaz** que especifique:
  - Constantes de **eventos**.
  - Métodos para que el controlador pueda **consultar** y **actualizar** la información.
  - Métodos para **registrar** los controladores que estarán a la escucha de eventos.



MVC: El controlador

- El **controlador** representa la **lógica** de la interacción.
  - El controlador **interactúa** tanto con la vista como con el modelo.
  - El controlador debe **registrarse** en ciertos elementos activos de la vista, que emitirán **eventos** a partir de la interacción con el usuario.
  - Cuando el usuario **actúa** sobre la vista, se envía un **evento** al controlador.
  - El controlador interactúa con la vista para **consultar** la información suministrada por el usuario.
  - El controlador interactúa con el modelo para llevar a cabo las **acciones** adecuadas para responder al evento solicitado por la vista.
  - El controlador interactúa con el modelo para **consultar** la información actualizada, e interactúa con la vista para **actualizar** la información presentada al usuario.
- En un buen diseño, varias vistas podrían disponer del mismo controlador.
- También es posible disponer de varios controladores especializados, cada uno controlando distintos eventos.
- A veces, es posible que el modelo y la vista interactúen directamente entre sí.

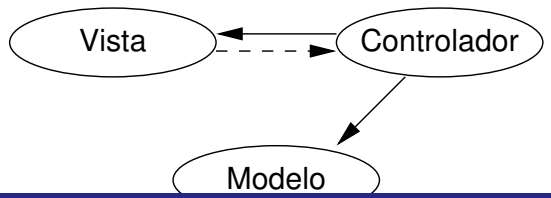
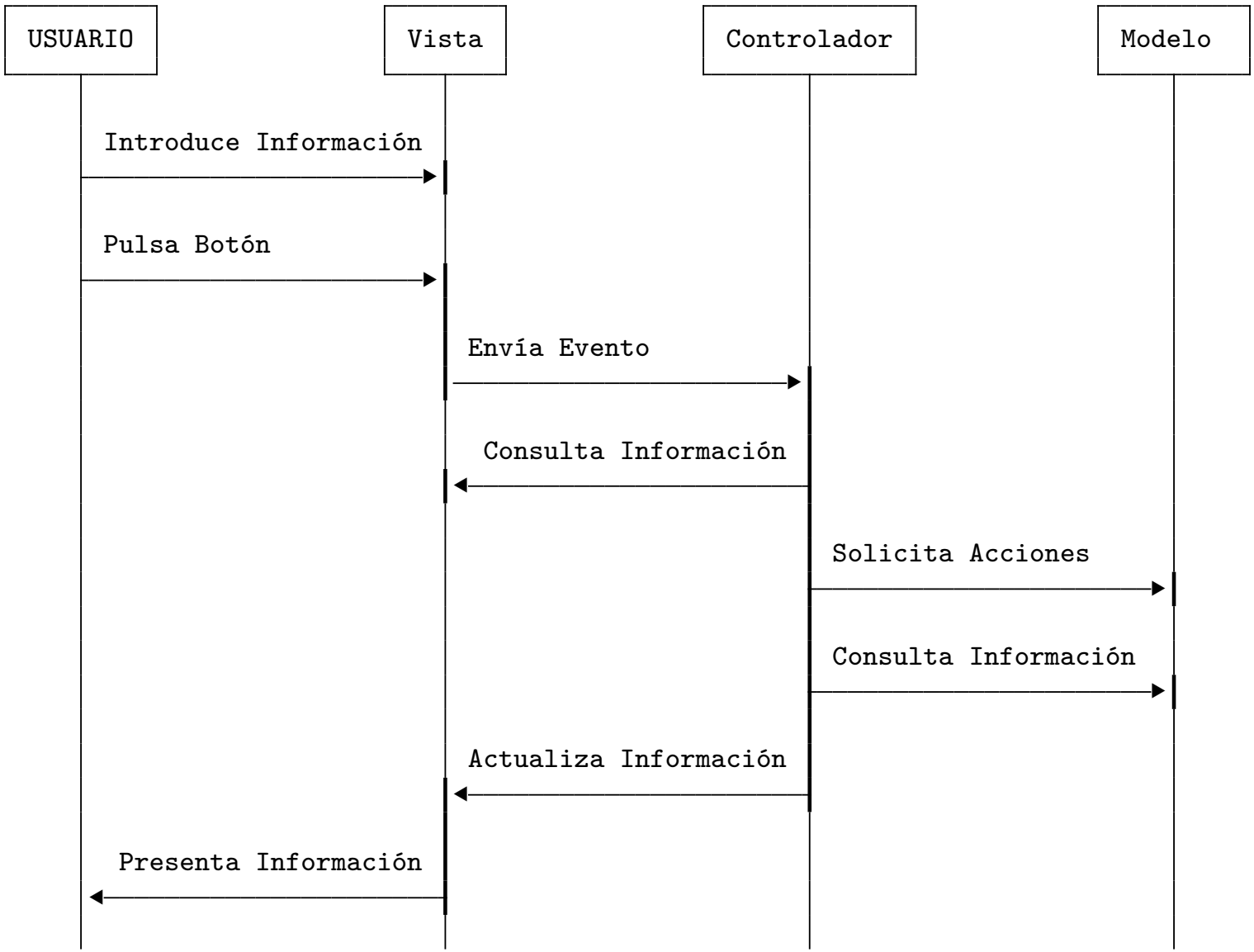


Diagrama de secuencia UML del escenario de interacción



El modelo de eventos: Eventos

- El **modelo de eventos** proporciona un mecanismo adecuado para establecer una **comunicación** reactiva entre la **vista** y el **controlador**.
- Un componente puede disparar un **evento** como resultado de una interacción con el usuario u otros componentes.
- Los eventos se definen como subclases de `java.util.EventObject`, y se encuentran en los paquetes: `java.awt.event` y `javax.swing.event`.
- El nombre de la clase de un evento tiene el formato `XxxxxEvent` (según el tipo de evento).
- Por ejemplo, el evento `ActionEvent` se dispara si:
  - Se pulsa un botón de cualquier tipo.
  - Se selecciona una opción de menú.
  - Se pulsa retorno de carro en un campo de texto.

Evento	Interfaz	Métodos
`ActionEvent`	`ActionListener`	`actionPerformed(Actio`
`ItemEvent`	`ItemListener`	`itemStateChanged(Item`
`FocusEvent`	`FocusListener`	`focusGained(FocusEven`
		`focusLost(FocusEvent)`
`MouseEvent`	`MouseListener`	`mouseClicked(MouseEve`
		Entered, Exited, etc.

El modelo de eventos: Eventos e Identificadores

- El **modelo de eventos** proporciona un mecanismo adecuado para establecer una **comunicación** reactiva entre la **vista** y el **controlador**.
- Un componente puede disparar un **evento** como resultado de una interacción con el usuario u otros componentes.
- Se puede asociar un **identificador** (**String**) a un determinado componente, de tal forma que los eventos del tipo `ActionEvent` generados por ese componente lleven asociados el identificador registrado.
- Para registrar un identificador a un determinado componente, se debe invocar al método `setActionCommand(String)` sobre el componente.

```
componente.setActionCommand("Sumar");
```

Evento	Registro de Identificadores	Interfaz
`ActionEvent`	`setActionCommand(String)`	`ActionListener`

El modelo de eventos: Registro de controladores

- Cuando un componente dispara un **evento**, se comunica con cada uno de los objetos controladores a la escucha (**listeners**) que tenga registrados el componente.
- Para **registrar** un *controlador a la escucha* de eventos disparados por un componente, se debe invocar al método adecuado (según el tipo de evento, `addXxxxxListener()`) sobre el componente que emite el evento, especificando cual es el controlador a la escucha. Por ejemplo:

```
componente.addActionListener(controlador);
```

- El *controlador a la escucha* debe **implementar la interfaz** adecuada (según el tipo de evento, `XxxxxListener`), para proporcionar el comportamiento adecuado en el tratamiento de dicho evento. Por ejemplo:

```
public class Controlador implements ActionListener {
    public void actionPerformed(ActionEvent e) { /* ... */ }
}
```

Evento	Registro de Controlador	Interfaz
`ActionEvent`	`addActionListener(ActionListener)`	`ActionListener`
`ItemEvent`	`addItemListener(ItemListener)`	`ItemListener`
`FocusEvent`	`addFocusListener(FocusListener)`	`FocusListener`
`MouseEvent`	`addMouseListener(MouseListener)`	`MouseListener`

El modelo de eventos: Tratamiento de eventos

- Cuando un componente dispara un evento, envía a cada uno de los *controladores a la escucha* registrados un mensaje que lleva como argumento el evento generado.
- El controlador *captura* dicho mensaje mediante la implementación de los métodos correspondientes especificados por la interfaz (según el tipo de evento, `XxxxxListener`).
- Las diferentes interfaces relacionadas con eventos obligan a implementar distintos métodos por parte de los controladores correspondientes. Por ejemplo:

```
public class Controlador implements ActionListener {
    public void actionPerformed(ActionEvent e) { /* ... */ }
}
```

Evento	Interfaz	Métodos
`ActionEvent`	`ActionListener`	`actionPerformed(Actio`
`ItemEvent`	`ItemListener`	`itemStateChanged(Item`
`FocusEvent`	`FocusListener`	`focusGained(FocusEven`
		`focusLost(FocusEvent)`
`MouseEvent`	`MouseListener`	`mouseClicked(MouseEve`
		Entered, Exited, etc.

El modelo de eventos: Distinción de eventos

- En el tratamiento de eventos, hay dos mecanismos diferentes para conocer cual es el origen o la causa del evento:
  - Utilizar **identificadores de acciones**. Para ello:
    - Sobre el componente, se puede invocar al método `setActionCommand(String)` para asociar identificadores de acciones con los eventos que emite:
    - Sobre el objeto evento, se puede invocar al método `getActionCommand()` para conocer su identificador asociado, que fue previamente establecido:

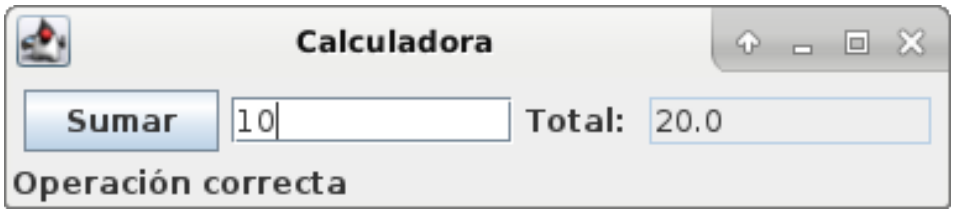
```
public void actionPerformed(ActionEvent e) {
    String c = e.getActionCommand();
    switch (c) {
    case Vista.SUMAR: sumarValor(); break;
    }
}
```

  - Sobre el objeto evento, se puede invocar al método `getSource()` para conocer que componente disparó el evento:

```
public void actionPerformed(ActionEvent e) {
    JButton boton = (JButton) e.getSource();
    String c = boton.getText();
    switch (c) {
    case Vista.SUMAR: sumarValor(); break;
    }
}
```

Ejemplo MVC: diagrama de clases UML de la aplicación

- Ejemplo: aplicación de una calculadora simple que permite sumar cantidades.



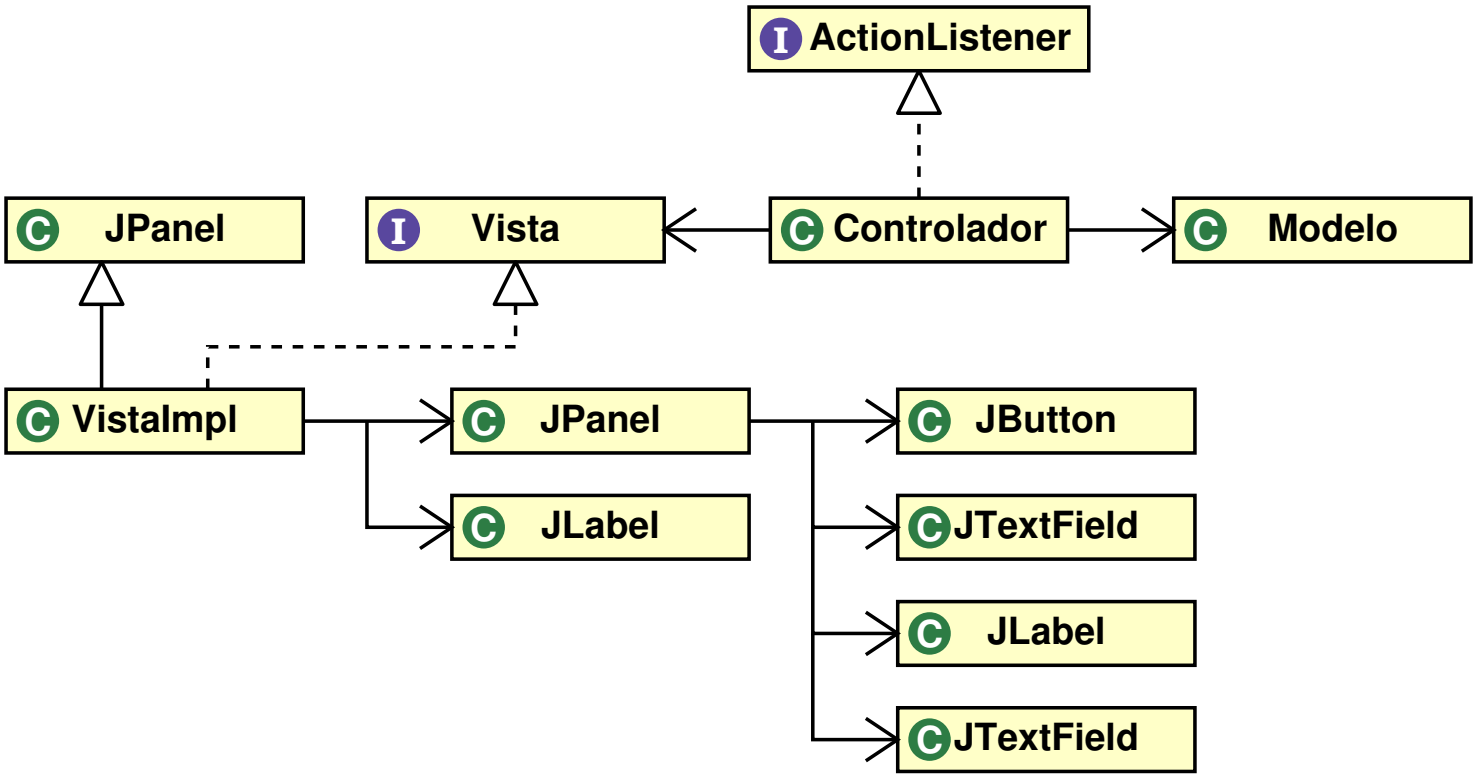
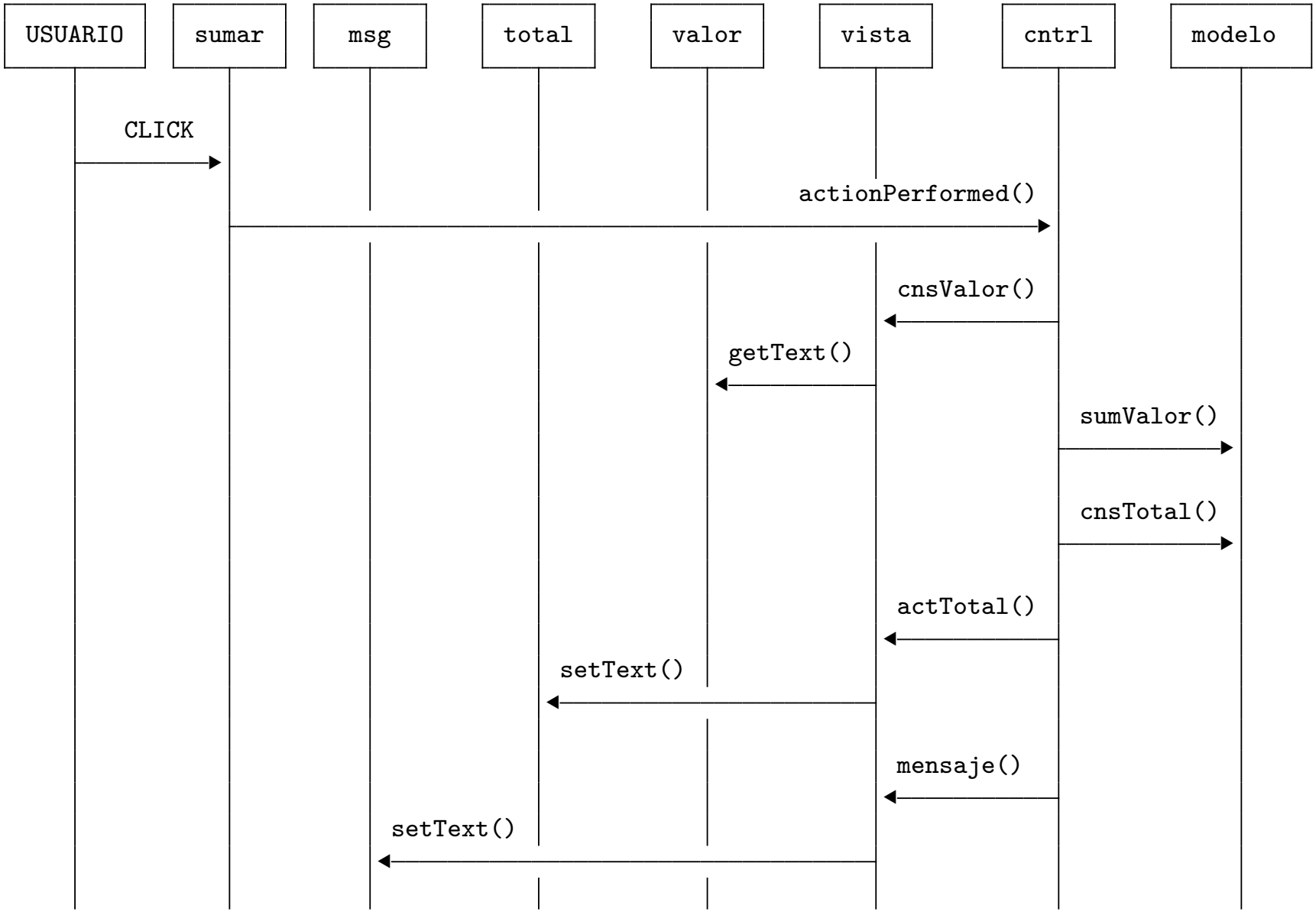


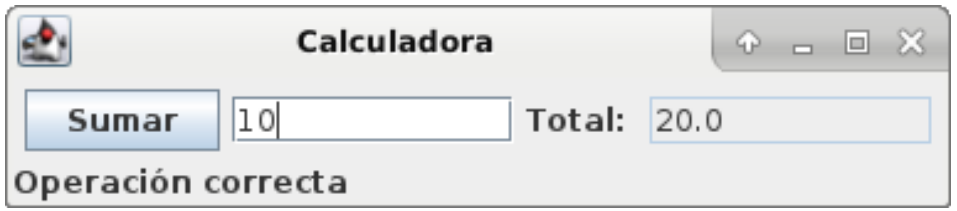
Diagrama de secuencia UML del escenario de interacción

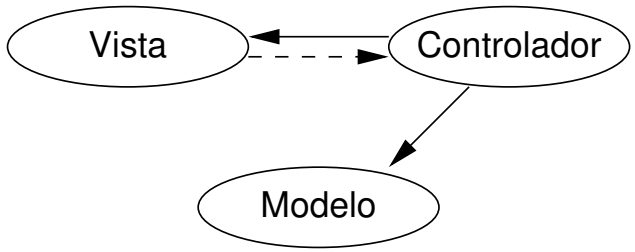


Ejemplo MVC: el modelo

- El Modelo representa la información y la lógica de la aplicación, con las siguientes operaciones:
  - `void sumarValor(double)`: añade la cantidad especificada al total acumulado
  - `double consultarTotal()`: consulta el total acumulado

```
public class Modelo {
    private double total;
    public Modelo() {
        total = 0;
    }
    public double consultarTotal() {
        return total;
    }
    public void sumarValor(double valor) {
        total += valor;
    }
}
```



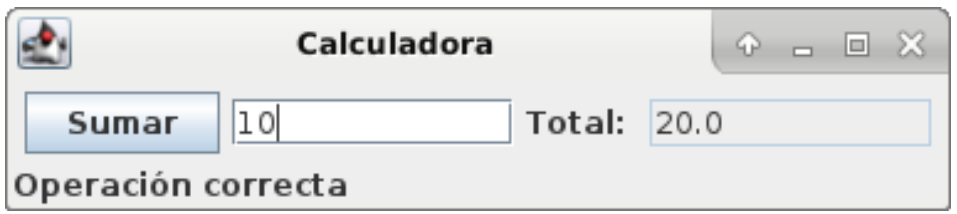


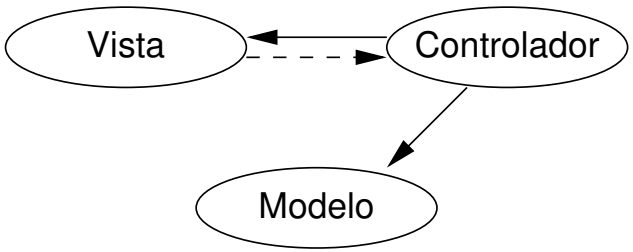
Ejemplo MVC: la interfaz de la vista

- Definimos una interfaz adecuada para las vistas:
  - Especifica las constantes y métodos para que el controlador pueda consultar y actualizar la información.
  - Especifica los métodos para registrar los controladores de eventos que estarán a la escucha.

```
import java.awt.event.ActionListener;
public interface Vista {
    public static final String SUMAR = "Sumar";
    public double consultarValor();
    public void actualizarTotal(double t);
    public void mensaje(String m);
    public void registrarControlador(ActionListener ctrl);
}
```

  - La implementación del método `registrarControlador(ActionListener)` debe registrar el controlador en los componentes adecuados de la vista.
- Crearemos distintas vistas implementando este interfaz.



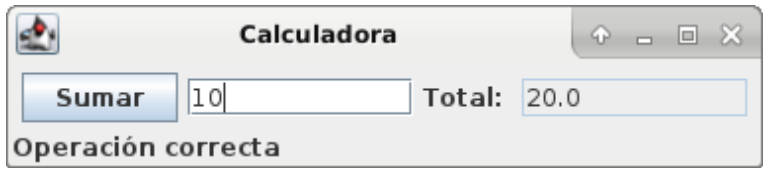


Ejemplo MVC: el controlador (I)

- El controlador **interactúa** con la vista y con el modelo.
- El controlador contiene, al menos, dos variables de instancia: el modelo, y la vista.
  - A veces, el controlador recibe la vista y el modelo desde el **exterior**.
  - A veces, el modelo se crea desde **dentro** del controlador.
- Implementa la interfaz `ActionListener`, control de eventos `ActionEvent`.
- Se debe registrar ante la vista como controlador (receptor de eventos).

Ejemplo MVC: el controlador (II)

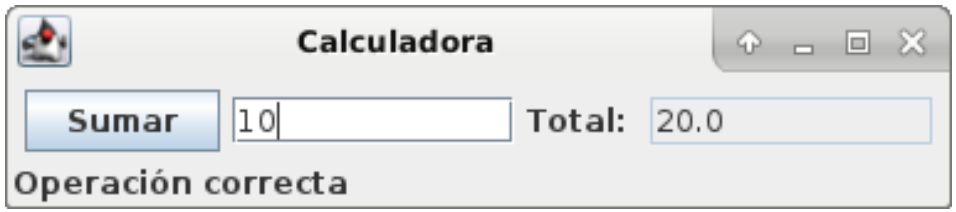
```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
public class Controlador implements ActionListener {
    private Vista vista;                                    // La vista y el modelo DEBEN ser
    private Modelo modelo;                                  // variables de INSTANCIA, no pueden
    public Controlador(Vista v, Modelo m) {                 // ser variables locales.
        vista = v;                                          // A veces, el modelo se crea en un
        modelo = m;                                         // método invocado por actionPerformed
        vista.mensaje("Inicio");                            // del controlador, como resultado al
    }                                                       // recibir un evento adecuado
    public void actionPerformed(ActionEvent e) {
        try {
            String c = e.getActionCommand();
            switch (c) {
            case Vista.SUMAR: sumarValor(); break;
            }
            vista.mensaje("Operación correcta");
        } catch (Exception ex) {
            vista.mensaje("Error: "+ex.getMessage());
        }
    }
    private void sumarValor() {
        modelo.sumarValor(vista.consultarValor());
        vista.actualizarTotal(modelo.consultarTotal());
    }
}
```



Construcción de Vistas

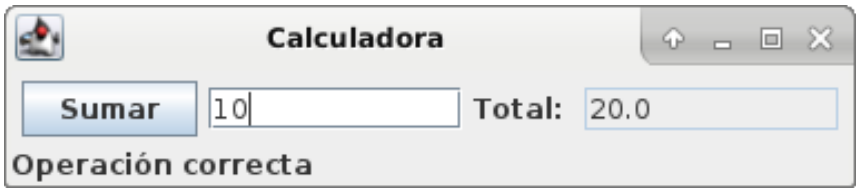
Definir la Vista, heredando de un contenedor intermedio (`JPanel`), e implementando el interfaz de la vista.

1. Construir los componentes visuales e interactivos.
   1. Seleccionar un gestor de esquemas para dicho contenedor (`setLayout()`).
   2. Crear los componentes visuales.
   3. Añadirlos al contenedor intermedio (`add()`).
2. Implementar los métodos definidos en la interfaz de la vista para la interacción con el controlador.



Ejemplo MVC: la implementación de la vista (I)

```
import java.awt.event.ActionListener;
import java.awt.FlowLayout;
import javax.swing.*;
public class VistaImpl extends JPanel implements Vista {
    private JButton sumar;
    private JTextField valor, total;
    private JLabel etq, msg;
    public VistaImpl() {                          // construir los componentes visuales
        sumar = new JButton(Vista.SUMAR);
        valor = new JTextField(10);
        etq = new JLabel("Total: ");
        msg = new JLabel(" ");
        total = new JTextField(10);
        total.setEditable(false);
        total.setText("0.0");
        JPanel panelSuperior = new JPanel();
        panelSuperior.setLayout(new FlowLayout());
        panelSuperior.add(sumar);
        panelSuperior.add(valor);
        panelSuperior.add(etq);
        panelSuperior.add(total);
        this.setLayout(new BorderLayout());
        this.add(panelSuperior, BorderLayout.CENTER);
        this.add(msg, BorderLayout.SOUTH);
    }
```



Ejemplo MVC: la implementación de la vista (II)

```
    public double consultarValor() {                        // implementar la interfaz
        return Double.parseDouble(valor.getText());
    }
    public void actualizarTotal(double t) {                 // implementar la interfaz
        total.setText(String.valueOf(t));
    }
    public void mensaje(String m) {
        msg.setText(m);
    }
    public void registrarControlador(ActionListener ctrl) { // implem. la interfaz
        sumar.addActionListener(ctrl);
        sumar.setActionCommand(Vista.SUMAR);
        valor.addActionListener(ctrl);
        valor.setActionCommand(Vista.SUMAR);
    }
}
```

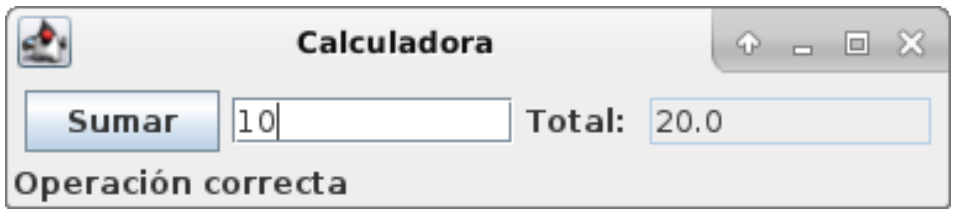
Ejemplo MVC: la aplicación principal (I)

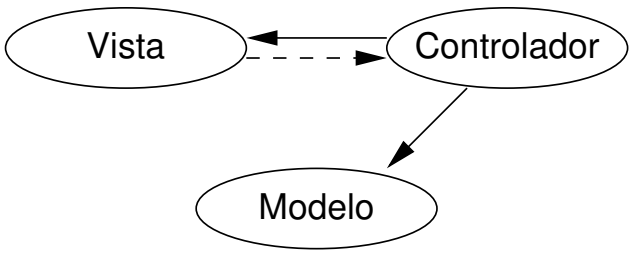
- La aplicacion principal debe crear instancias para el modelo, la vista y el controlador, así como registrar el controlador en la vista.
  - A veces, el propio controlador **crea internamente** al modelo
  - También es posible que sea el constructor del controlador el que registre al propio controlador en la vista.
- Crear un marco superior donde colocar y mostrar la vista.
  1. Crear un objeto de la clase contenedora superior (`JFrame`).
  2. Especificar la *operación de cierre* (`setDefaultCloseOperation()`).
  3. Usar un objeto de la clase Vista creada anteriormente como el panel de contenidos del objeto contenedor superior (`setContentPane()`).
  4. Dimensionar el contenedor superior (`pack()` o `setSize()`).
  5. Mostrar el contenedor superior (`setVisible()`).

Ejemplo MVC: la aplicación principal (II)

```
import javax.swing.JFrame;
public class MVC1 {
    public static void main(String[] args) {
        Modelo modelo = new Modelo();
        VistaImpl vista = new VistaImpl();
        Controlador ctrl = new Controlador(vista, modelo);
        vista.registrarControlador(ctrl);

        JFrame marco = new JFrame("Calculadora");
        marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        marco.setContentPane(vista);
        marco.pack();
        marco.setVisible(true);
    }
}
```

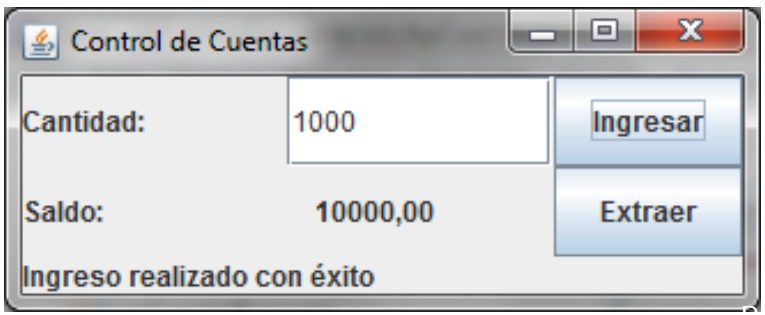




Componentes y contenedores

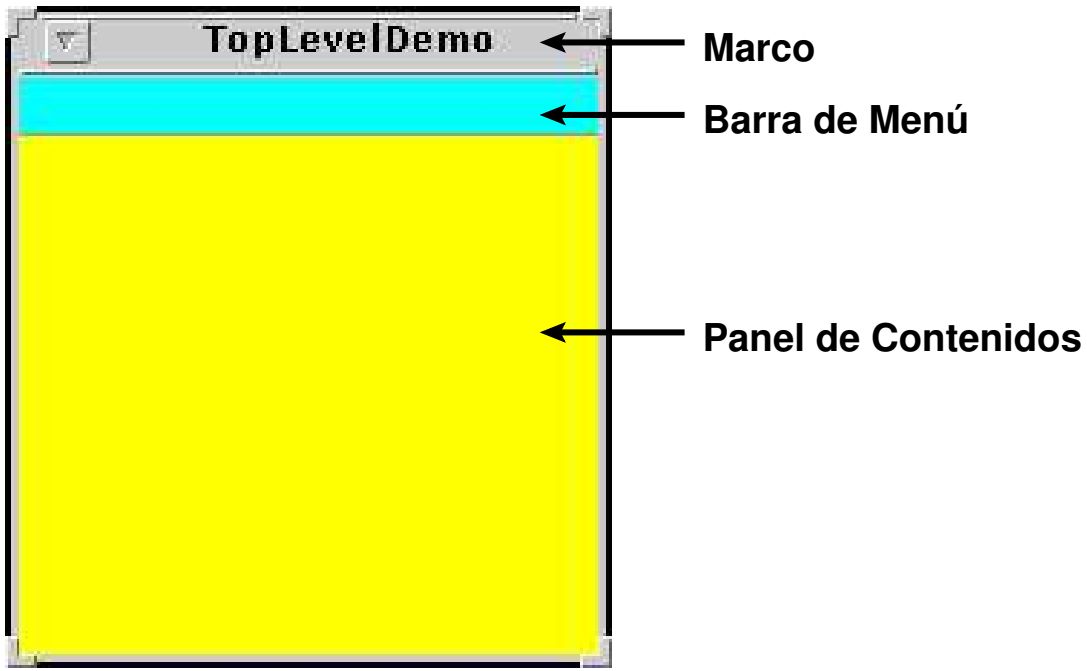
- Los **componentes** son los elementos visuales de una GUI:
  - Botones (`JButton`), etiquetas (`JLabel`), campos de texto (`JTextField`), etc.
  - Se sitúan dentro de algún contenedor (componente especializado)
  - Pagina web con ejemplos de los componentes

    `https://docs.oracle.com/javase/tutorial/uiswing/examples/components/index.html`
- Los **contenedores** contienen y organizan los componentes:
  - Pueden contener componentes y otros contenedores.
  - Los contenedores pueden ser de dos tipos:
    - Superiores: forman la base para realizar una GUI (`JApplet`, `JFrame` y `JDialog`).
    - Intermedios: almacenan componentes y contenedores intermedios (`JPanel`, `JScrollPane`, `JSplitPane`, `JTabbedPane`, etc.).

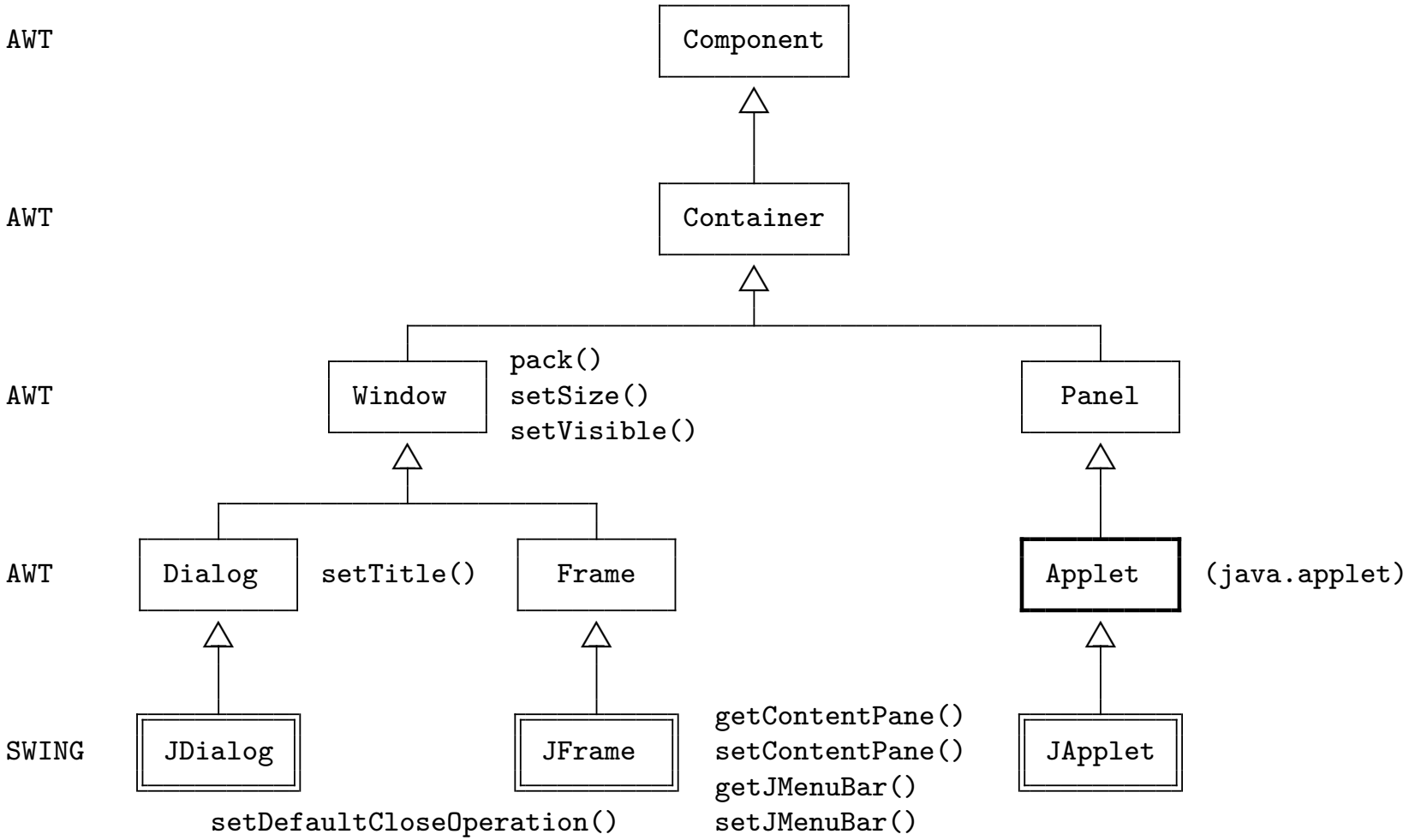


Los contenedores superiores

- Los **contenedores superiores** (`JApplet`, `JFrame` y `JDialog`) proporcionan un **marco** que contiene toda la intefaz gráfica de la aplicación.
- Contienen un panel de contenidos (`ContentPane`) donde usualmente se coloca la vista.
- Pueden contener opcionalmente una barra de menú (`MenuBar`).



La jerarquía de los contenedores superiores



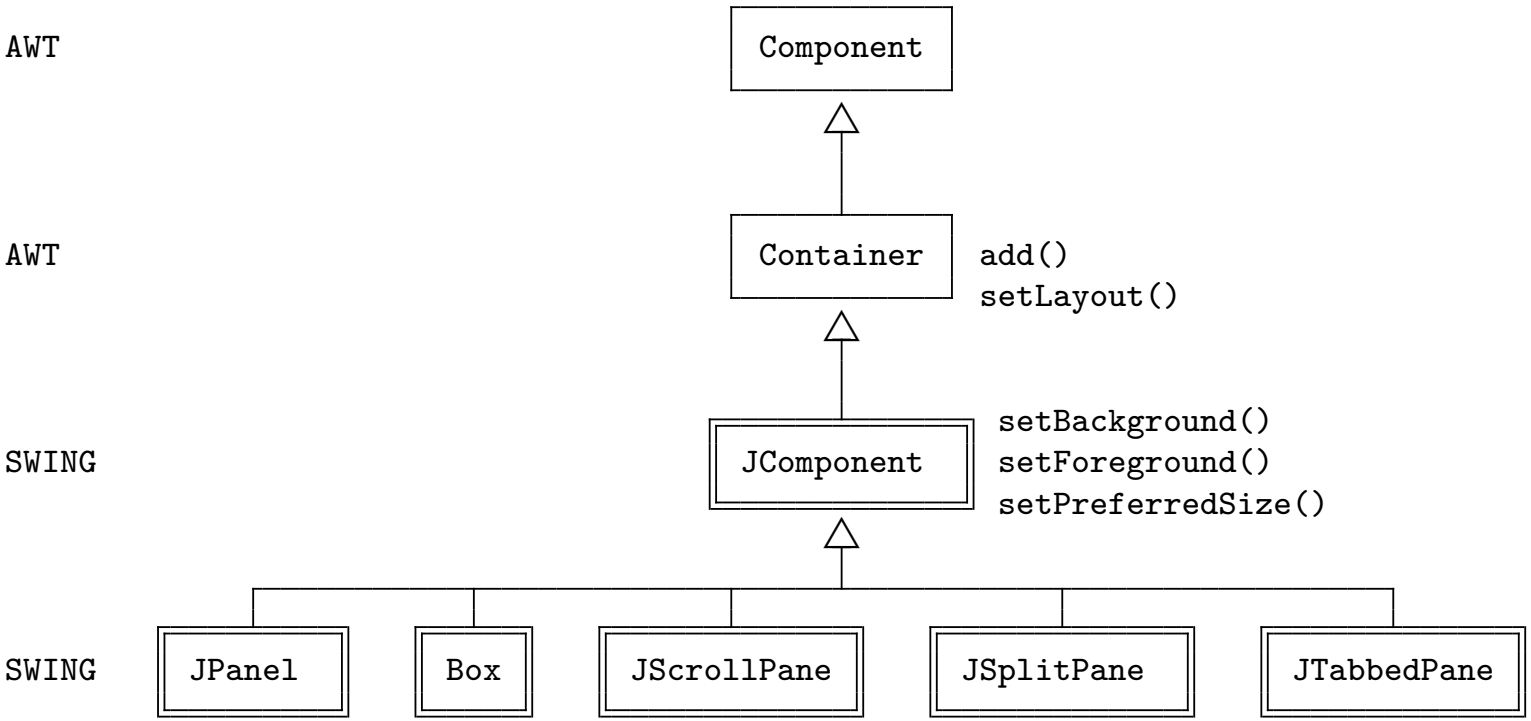
El contenedor superior `javax.swing.JFrame`

- El contenedor superior `javax.swing.JFrame` proporciona un **marco** que contiene toda la intefaz gráfica de la aplicación.
- Proporciona, entre otros, los siguientes constructores y métodos públicos:

```
JFrame();                          // crea un marco invisible
JFrame(String);                    // crea un marco invisible con título
void setTitle(String);             // actualiza el título del marco
void setDefaultCloseOperation(int); // JFrame.EXIT_ON_CLOSE
JMenuBar getJMenuBar();            // devuelve la barra de menú
void setJMenuBar(JMenuBar);        // actualiza la barra de menú
Container getContentPane();        // devuelve el panel de contenidos
void setContentPane(Container);    // actualiza el panel de contenidos
void pack();                       // redimensiona ajustando al contenido
void setSize(int, int);            // redimensiona a la anchura y altura
void setVisible(boolean);          // actualiza el estado de visibilidad
```

La jerarquía de los contenedores intermedios

- Los **contenedores intermedios** (`JPanel`, `JScrollPane`, `JSplitPane`, `JTabbedPane`, etc) permiten organizar la estructura y composición de las vistas.
- Contienen a los componentes y a otros contenedores, permitiendo construir jerarquías de componentes.

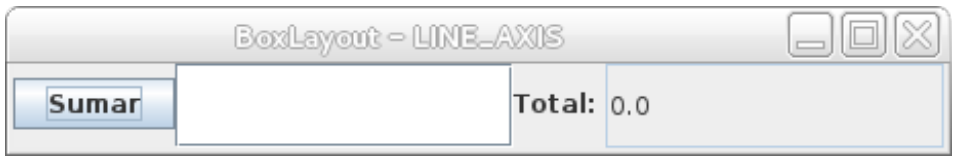


El contenedor javax.swing.JPanel

- El contenedor intermedio más utilizado es `javax.swing.JPanel`.
- El contenedor `javax.swing.JPanel` permite añadir múltiples componentes y contenedores intermedios al panel, que serán organizados según un determinado gestor de esquemas.
- El gestor de esquemas por defecto del contenedor `JPanel` es `FlowLayout`.
- Proporciona, entre otros, los siguientes constructores y métodos públicos:
  - `JPanel();` // crea un panel contenedor con FlowLayout
  - `JPanel(LayoutManager);` // crea un panel contenedor con esquema
  - `void setLayout(LayoutManager);` // actualiza el gestor de esquemas
  - `void add(Component);` // añade al final el componente
  - `void add(Component, Object);` // añade el componente según especificado

El contenedor javax.swing.Box

- El contenedor `javax.swing.Box` permite añadir múltiples componentes y contenedores intermedios al panel, que serán organizados según el gestor de esquemas `BoxLayout`.
- Proporciona, entre otros, los siguientes constructores y métodos públicos:
  - `Box(int);` // crea un panel con BoxLayout
  - `static Box createHorizontalBox();` // crea un panel horizontal
  - `static Box createVerticalBox();` // crea un panel vertical
  - `void add(Component);` // añade al final el componente
- Es posible añadir al contenedor espacios de tamaño fijo, y expandible:
  - `static Component createHorizontalStrut(int width);` // espacio horizontal fijo
  - `static Component createVerticalStrut(int height);` // espacio vertical fijo
  - `static Component createRigidArea(Dimension);` // añade un espacio de tamaño fijo
  - `static Component createGlue();` // añade un espacio expandible
  - `static Component createHorizontalGlue();` // añade un espacio expandible horizontal
  - `static Component createVerticalGlue();` // añade un espacio expandible vertical

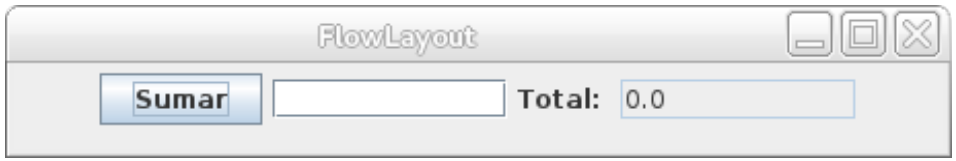


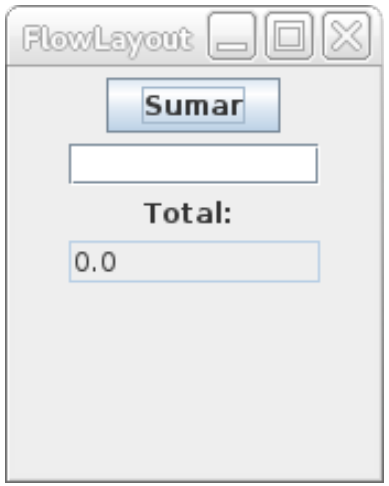
Gestores de esquemas

- Los **gestores de esquemas** son clases que determinan cómo se distribuirán los componentes dentro de un contenedor intermedio.
- La mayoría de gestores de esquemas están definidos en `java.awt`
  - `FlowLayout`: organiza los componentes según un flujo direccional.
  - `BoxLayout`: organiza los componentes horizontalmente o verticalmente.
  - `BorderLayout`: organiza los componentes alrededor de uno central.
  - `GridLayout`: organiza los componentes según una cuadrícula.
  - Etc.
- El contenedor `JPanel` dispone por defecto de un `FlowLayout`.

El gestor de esquemas `java.awt.FlowLayout`

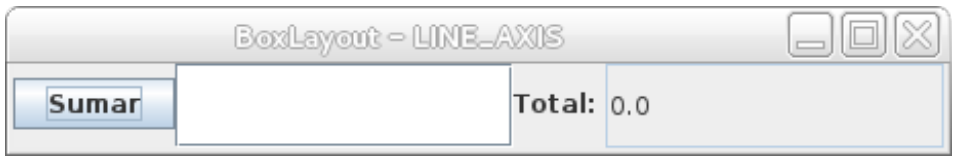
- Con el gestor de esquemas `java.awt.FlowLayout`, los componentes fluyen de izquierda a derecha y de arriba a abajo.
- Al cambiar el tamaño de la ventana, puede cambiar la disposición de los componentes.

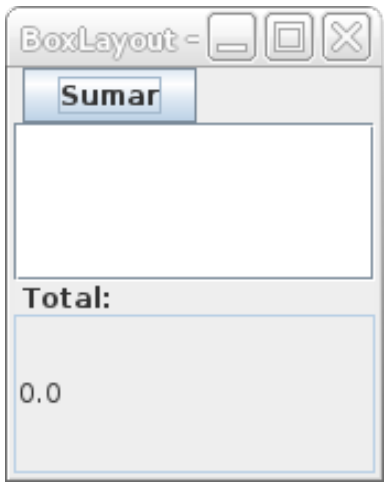




El gestor de esquemas `java.awt.BoxLayout`

- Con el gestor de esquemas `java.awt.BoxLayout`, los componentes se distribuyen horizontalmente o verticalmente.
- Al cambiar el tamaño de la ventana, se redimensionan los componentes.
- El constructor debe especificar el panel donde reside, y la orientación de los componentes (`BoxLayout.X_AXIS`, `BoxLayout.Y_AXIS`).
- Proporciona, el siguiente constructor:
  - `BoxLayout(Container, int);  // crea el gestor de esquemas`





El gestor de esquemas `java.awt.BorderLayout`

- Con el gestor de esquemas `java.awt.BorderLayout`, el contenedor se divide en 5 zonas: `NORTH`, `WEST`, `CENTER`, `EAST`, `SOUTH`.
  - Los componentes ajustan su tamaño hasta rellenar completamente cada zona.
  - Si falta algún componente, entonces su zona se ajusta con el resto.
  - Para añadir un componente al contenedor, se utiliza una versión de add que indica la zona en la que se añade (`BorderLayout.NORTH`, `BorderLayout.WEST`, `BorderLayout.CENTER`, `BorderLayout.EAST`, `BorderLayout.SOUTH`).

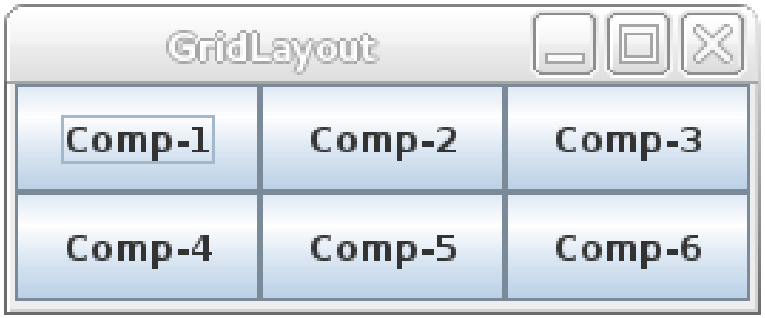
```
add(boton, BorderLayout.NORTH);
```

El gestor de esquemas `java.awt.GridLayout`

- Con el gestor de esquemas `java.awt.GridLayout`, el contenedor se divide en una cuadrícula con tantas filas y columnas como se especifique en el constructor.

```
setLayout(new GridLayout(2, 3)); // Dos filas y tres columnas
```

- Los componentes se mantienen de igual tamaño dentro de cada celda.
- El orden a la hora de agregar los componentes determina la posición que ocupan (de izquierda a derecha y de arriba a abajo).



Jerarquías de Contenedores

- Podemos utilizar un contenedor intermedio en lugar de un componente para agregarlo a otro contenedor intermedio.
- Este nuevo contenedor intermedio podrá:
  - Incorporar sus propios componentes.
  - Tener su propio gestor de esquemas.

El contenedor javax.swing.JScrollPane

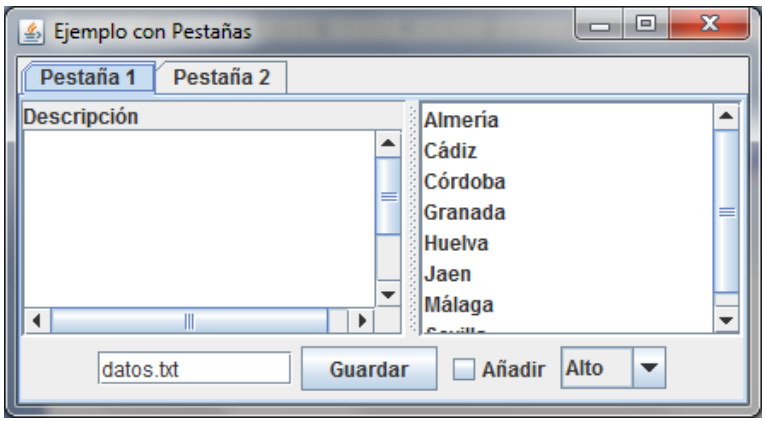
- El contenedor `javax.swing.JScrollPane` permite hacer **scroll** sobre un componente u otro contenedor intermedio.
- Proporciona, entre otros, los siguientes constructores y métodos públicos:
  - `JScrollPane();` // crea un panel de scroll
  - `JScrollPane(int,int);` // crea un panel de scroll con políticas
  - `JScrollPane(Component);` // crea un panel de scroll con componente
  - `JScrollPane(Component,int,int);` // crea un panel de scroll con componente y políticas
  - `void setViewportView(Component);` // actualiza el componente dentro del panel
- Constantes para políticas de control del scroll vertical y horizontal
  - `VERTICAL_SCROLLBAR_AS_NEEDED`
  - `VERTICAL_SCROLLBAR_ALWAYS`
  - `VERTICAL_SCROLLBAR_NEVER`
  - `HORIZONTAL_SCROLLBAR_AS_NEEDED`
  - `HORIZONTAL_SCROLLBAR_ALWAYS`
  - `HORIZONTAL_SCROLLBAR_NEVER`

El contenedor javax.swing.JSplitPane

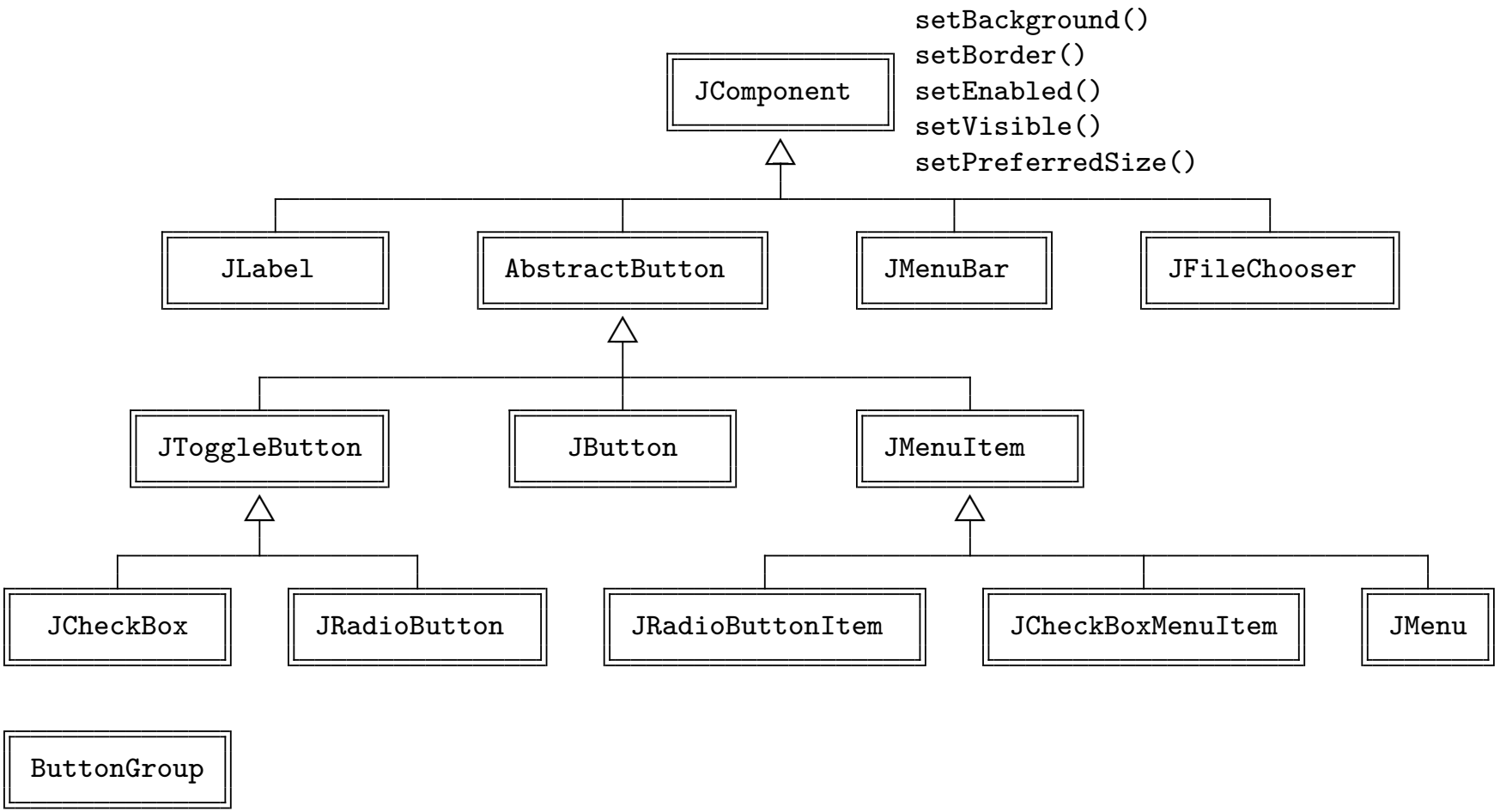
- El contenedor `javax.swing.JSplitPane` permite **dividir** un contenedor en dos mitades (vertical u horizontal).
- Proporciona, entre otros, los siguientes constructores y métodos públicos:
  - `JSplitPane();` // crea un panel de división
  - `JSplitPane(int);` // crea un panel de división con política
  - `JSplitPane(int,Component,Component);` // crea un panel de división con componentes y pol
  - `void setLeftComponent(Component);` // actualiza el componente izquierdo
  - `void setTopComponent(Component);` // actualiza el componente superior
  - `void setRightComponent(Component);` // actualiza el componente derecho
  - `void setBottomComponent(Component);` // actualiza el componente inferior
  - `void setDividerLocation(double);` // actualiza la posición de la división
- Constantes para políticas de control de la división:
  - `HORIZONTAL_SPLIT`
  - `VERTICAL_SPLIT`

El contenedor `javax.swing.JTabbedPane`

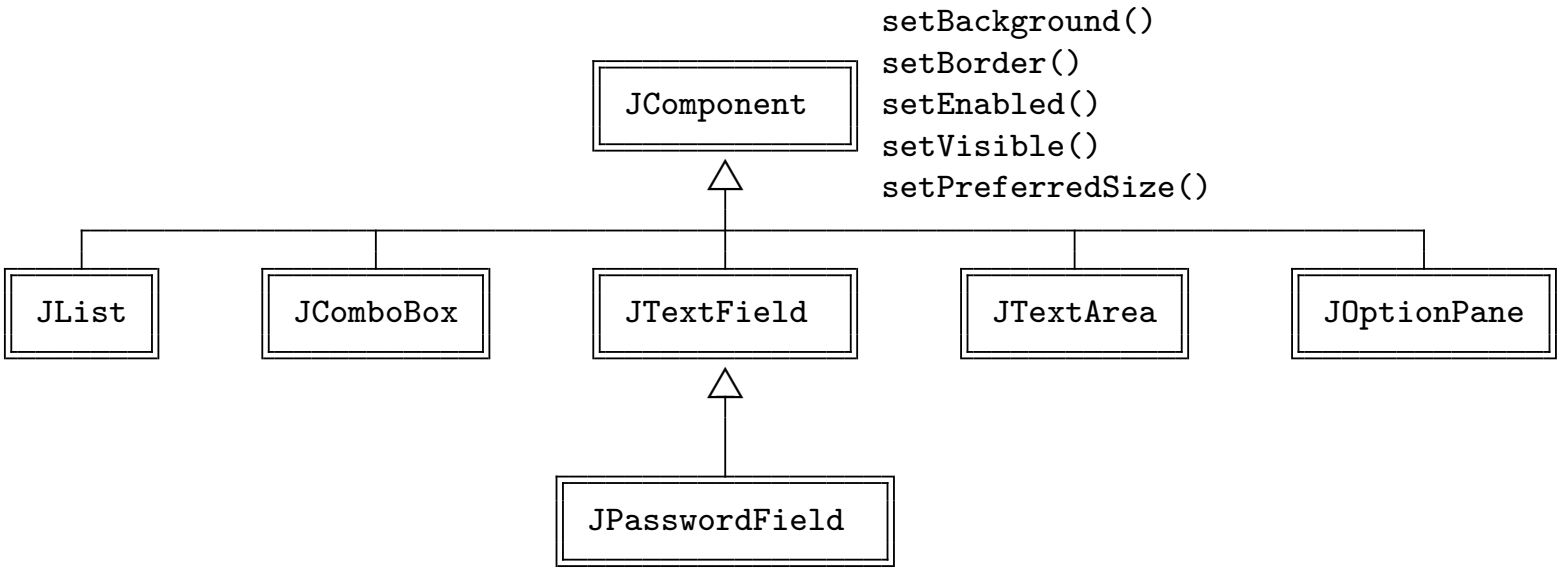
- El contenedor `javax.swing.JTabbedPane` permite crear un contenedor con varias **pestañas** seleccionables, cada una con su propio componente.
- Proporciona, entre otros, los siguientes constructores y métodos públicos:
  - `JTabbedPane();` // crea un panel de pestañas
  - `JTabbedPane(int);` // crea un panel de pestañas con política
  - `void setSelectedIndex(int);` // selecciona una pestaña
  - `void setSelectedComponent(Component);` // selecciona una pestaña
  - `void addTab(String, Component);` // añade una pestaña con tit y comp
  - `void addTab(String, Icon, Component);` // añade con tit, icono y comp
- Constantes para políticas de control de pestañas:
  - `TOP`
  - `BOTTOM`
  - `LEFT`
  - `RIGHT`



La jerarquía de los componentes (I)



La jerarquía de los componentes (II)



Iconos

- En algunos constructores y métodos de los componentes aparece un argumento `Icon` que representa un icono (`javax.swing.Icon` es una interfaz).
- `javax.swing.ImageIcon` es una clase que implementa la interfaz `Icon`.
  - Para cargar un icono desde un fichero: `new ImageIcon("Imagen.jpg")`
  - Para utilizar un icono en un botón:

    ```
    JButton btn = new JButton(new ImageIcon("Imagen.jpg"));
    ```

  - Para utilizar un icono en una etiqueta:

    ```
    JLabel etq = new JLabel(new ImageIcon("Imagen.jpg"));
    ```

  - Es posible acceder a la imagen (`java.awt.Image`) del icono:

    ```
    Image img = new ImageIcon("Imagen.jpg").getImage();
    ```

El componente javax.swing.JLabel

- La clase `JLabel` permite crear etiquetas con texto o graficos.
- Proporciona, entre otros, los siguientes constructores y métodos públicos:

```
JLabel();                   // crea una etiqueta sin texto ni icono
JLabel(Icon);               // crea una etiqueta con icono
JLabel(Icon, int);          // crea una etiqueta con icono y alineación
JLabel(String);             // crea una etiqueta con texto
JLabel(String, int);        // crea una etiqueta con texto y alineación
JLabel(String, Icon);       // crea una etiqueta con texto e icono
JLabel(String, Icon, int);  // crea una etiqueta con texto, icono y alineación
String getText();           // devuelve el texto de la etiqueta
void setText(String);       // actualiza el texto de la etiqueta
Icon getIcon();             // devuelve el icono de la etiqueta
void setIcon(Icon);         // actualiza el icono de la etiqueta
```

```
JLabel.LEFT JLabel.RIGHT JLabel.CENTER // constantes para alineación
```

El componente javax.swing.JButton

- La clase `JButton` permite crear botones que ceden ante una pulsación.
- Proporciona, entre otros, los siguientes constructores y métodos públicos:

```
JButton();                          // crea un botón sin texto ni icono
JButton(Icon);                      // crea un botón con icono
JButton(String);                    // crea un botón con texto
JButton(String, Icon);              // crea un botón con texto e icono
String getText();                   // devuelve el texto del botón
void setText(String);               // actualiza el texto del botón
Icon getIcon();                     // devuelve el icono del botón
void setIcon(Icon);                 // actualiza el icono del botón
void setActionCommand(String);      // actualiza el comando de acción
void addActionListener(ActionListener); // añade un controlador de acción
```

El componente javax.swing.JCheckBox

- La clase `JCheckBox` permite crear marcadores que pueden activarse o desactivarse con una pulsación.
- Proporciona, entre otros, los siguientes constructores y métodos públicos:

```
JCheckBox();                        // crea un marcador sin texto ni icono
JCheckBox(Icon, boolean);           // crea un marcador con icono y estado
JCheckBox(String, boolean);         // crea un marcador con texto y estado
JCheckBox(String, Icon, boolean);// crea un marcador con texto, icono y estado
String getText();                   // devuelve el texto del marcador
void setText(String);               // actualiza el texto del marcador
Icon getIcon();                     // devuelve el icono del marcador
void setIcon(Icon);                 // actualiza el icono del marcador
boolean isSelected();               // devuelve el estado del marcador
void setSelected(boolean);          // actualiza el estado del marcador
void setActionCommand(String);   // actualiza el comando de acción
void addActionListener(ActionListener); // añade un controlador de acción
void addItemListener(ItemListener);     // añade un controlador de estado
```

javax.swing.JRadioButton y ButtonGroup

- La clase `JRadioButton` permite crear marcadores de selección alternativa que pueden activarse o desactivarse con una pulsación.
- Se agrupan de manera que sólo uno esté pulsado.
- Para agruparlos, se crea una instancia de `javax.swing.ButtonGroup` y los objetos `JRadioButton` se añaden al grupo con `add()`.
- Proporciona los constructores y métodos públicos similares a `JCheckBox` mostrados anteriormente.

El componente javax.swing.JTextField

- La clase `JTextField` permite crear campos de texto editables.
- Proporciona, entre otros, los siguientes constructores y métodos públicos:
  - `JTextField();` // crea un campo de texto
  - `JTextField(int);` // crea un campo de texto con anchura
  - `JTextField(String);` // crea un campo de texto con texto
  - `JTextField(String, int);` // crea un campo de texto con texto y anchura
  - `String getText();` // devuelve el texto del campo de texto
  - `void setText(String);` // actualiza el texto del campo de texto
  - `boolean isEditable();` // devuelve el estado del campo de texto
  - `void setEditable(boolean);` // actualiza el estado del campo de texto
  - `void setActionCommand(String);` // actualiza el comando de acción
  - `void addActionListener(ActionListener);` // añade un controlador de acción
- La subclase `JPasswordField` enmascara el eco, y añade el método:
  - `char[] getPassword();`

El componente javax.swing.JTextArea

- La clase `JTextArea` permite crear áreas de texto multi-línea editables.
- Proporciona, entre otros, los siguientes constructores y métodos públicos:

```
JTextArea();                        // crea un área de texto
JTextArea(int, int);                // crea un área de texto con filas y columnas
JTextArea(String);                  // crea un área de texto con texto
JTextArea(String, int, int);        // crea un área de texto con texto, filas y columnas
String getText();                   // devuelve el texto del área de texto
void setText(String);               // actualiza el texto del área de texto
void append(String);                // añade texto al final área de texto
void insert(String, int);           // inserta texto en la posición del área de texto
void replaceRange(String,int,int); // reemplaza texto
boolean isEditable();               // devuelve el estado del área de texto
void setEditable(boolean);          // actualiza el estado del área de texto
```

```
((DefaultCaret)textArea.getCaret()).setUpdatePolicy(DefaultCaret.ALWAYS_UPDATE);
```

El componente javax.swing.JList<E>

- La clase `JList<E>` permite mostrar una lista desplegada de elementos para su selección.
- Proporciona, entre otros, los siguientes constructores y métodos públicos:
  - `JList();` // crea una lista seleccionable
  - `JList(E[]);` // crea una lista seleccionable con elementos
  - `int getSelectedIndex();` // devuelve el índice del elemento seleccionado
  - `int[] getSelectedIndices();` // devuelve los índices de los elems selecc
  - `E getSelectedValue();` // devuelve el valor seleccionado
  - `List<E> getSelectedValuesList();` // devuelve los valores seleccionados
  - `boolean isSelectedIndex(int);` // comprueba si el indice está seleccionado
  - `boolean isSelectionEmpty();` // comprueba si la selección esta vacía
  - `void setListData(E[]);` // actualiza los elementos de la lista
  - `void setSelectionMode(int);` // actualiza el modo de selección
  - `int getSelectionMode();` // devuelve el modo de selección actual
  - `void clearSelection();` // reinicia la selección
  - `void addListSelectionListener(ListSelectionListener);`

  - `ListSelectionModel.SINGLE_SELECTION`
  - `ListSelectionModel.SINGLE_INTERVAL_SELECTION`
  - `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION`

El componente javax.swing.JComboBox<E>

- La clase `JComboBox<E>` permite seleccionar un elemento de una lista desplegable.
- Proporciona, entre otros, los siguientes constructores y métodos públicos:

```
JComboBox();                        // crea una lista desplegable
JComboBox(E[]);                     // crea una lista desplegable con elementos
void addItem(E);                    // añade un elemento a la lista
E getItemAt(int);                   // devuelve un elemento a la lista
int getItemCount();                 // devuelve el número de elementos de la lista
int getSelectedIndex();             // devuelve el índice del elemento seleccionado
Object getSelectedItem();           // devuelve el valor seleccionado
boolean isEditable();               // devuelve el estado de edición
void setEditable(boolean);          // actualiza el estado de edición
void setActionCommand(String);      // actualiza el comando de acción
void addActionListener(ActionListener); // añade un controlador de acción
void addItemListener(ItemListener);     // añade un controlador de estado
```

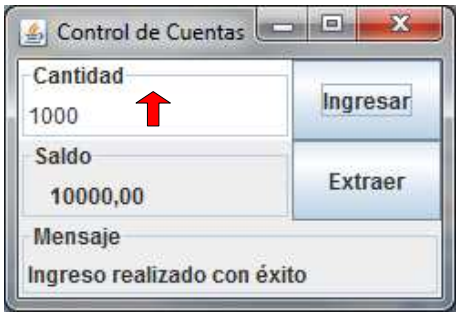
La clase javax.swing.BorderFactory

- En el paquete `javax.swing.border` existen clases que permiten definir un borde a un componente.
- Se puede invocar al método `setBorder()` de un componente para cambiar el borde:
  - `public void setBorder(Border);`
    - Ejemplo:

```
JTextField cant = new JTextField(15);
cant.setBorder(new TitledBorder("Cantidad"));
```

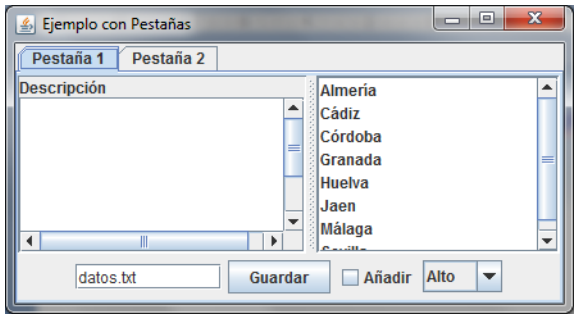
- La clase `javax.swing.BorderFactory` dispone de métodos de clase (métodos factoría) para crear bordes:

```
JTextField cant = new JTextField(15);
cant.setBorder(BorderFactory.createTitledBorder("Cantidad"));
```



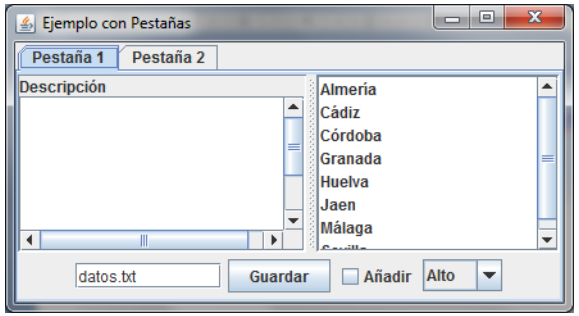
Un ejemplo completo (I)

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
static class VistaImpl extends JPanel implements Vista {
    private static final String[] valLista = {
        "Almería", "Cadiz", "Córdoba", "Granada",
        "Huelva", "Jaén", "Málaga", "Sevilla" };
    private static final String[] valNivel = {
        "Alto", "Medio", "Bajo" };
    private JTextArea desc;
    private JList<String> lista;
    private JButton guardar;
    private JTextField nombre;
    private JCheckBox anyadir;
    private JComboBox<String> nivel;
    public VistaImpl() {
        //- Componentes ----------
        JLabel etq = new JLabel("Descripción");
        desc = new JTextArea(10, 20);
        lista = new JList<String>(valLista);
        nombre = new JTextField("datos.txt", 12);
        guardar = new JButton("Guardar");
        anyadir = new JCheckBox("Añadir", false);
        nivel = new JComboBox<String>(valNivel);
        //- Scrolls --------------
        JScrollPane descScroll = new JScrollPane(desc);
        JScrollPane listaScroll = new JScrollPane(lista);
```



Un ejemplo completo (II)

```
        //- Paneles --------------
        JPanel izq = new JPanel(new BorderLayout());
        izq.add(etq, BorderLayout.NORTH);
        izq.add(descScroll, BorderLayout.CENTER);
        JSplitPane sup = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, izq, listaScroll);
        //------------------------
        JPanel inf = new JPanel(new FlowLayout());
        inf.add(nombre);
        inf.add(guardar);
        inf.add(anyadir);
        inf.add(nivel);
        //------------------------
        JPanel tab1 = new JPanel(new BorderLayout());
        tab1.add(sup, BorderLayout.CENTER);
        tab1.add(inf, BorderLayout.SOUTH);
        //------------------------
        Icon imagen = new ImageIcon("imagen.jpg");
        JLabel tab2 = new JLabel(imagen);
        //------------------------
        JTabbedPane tabs = new JTabbedPane(JTabbedPane.TOP);
        tabs.addTab("Pestaña 1", tab1);
        tabs.addTab("Pestaña 2", tab2);
        //------------------------
        setLayout(new BorderLayout());
        add(tabs, BorderLayout.CENTER);
    }
    // ...
}
```



Ejemplo MVC: gestión de cuentas bancarias (I)

- Aplicación para la gestión de cuentas bancarias.
  - Permite manipular una cuenta bancaria. Operaciones:
    - `void ingresa(double)`: ingresa en la cuenta
    - `double extrae(double)`: extrae de la cuenta
    - `double saldo()`: consulta el saldo
- El modelo: la clase `Cuenta`:

```
public class Cuenta {
    private double saldo;
    public Cuenta(double saldoInicial) {
        saldo = saldoInicial;
    }
    public double saldo() {
        return saldo;
    }
    public void ingresa(double cantidad) {
        saldo += cantidad;
    }
    public double extrae(double cantidad) {
        double realExtrae = cantidad;
        if (realExtrae > saldo) {
            realExtrae = saldo;
        }
        saldo -= realExtrae;
        return realExtrae;
    }
}
```

Ejemplo MVC: gestión de cuentas bancarias (II)

- Posibles vistas para la aplicación de gestión de cuentas bancarias: