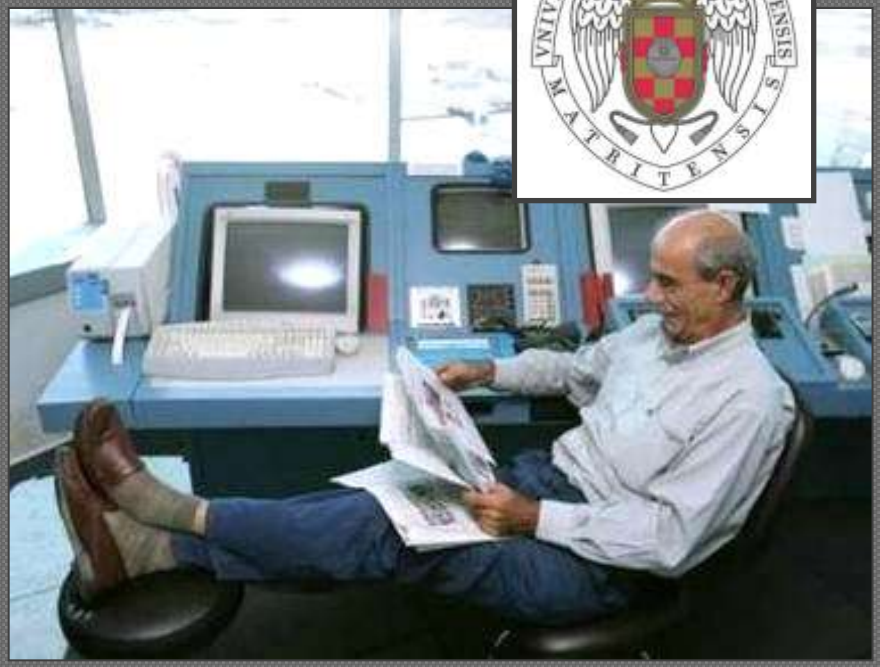
LPS: Patrón Modelo-Vista-Controlador



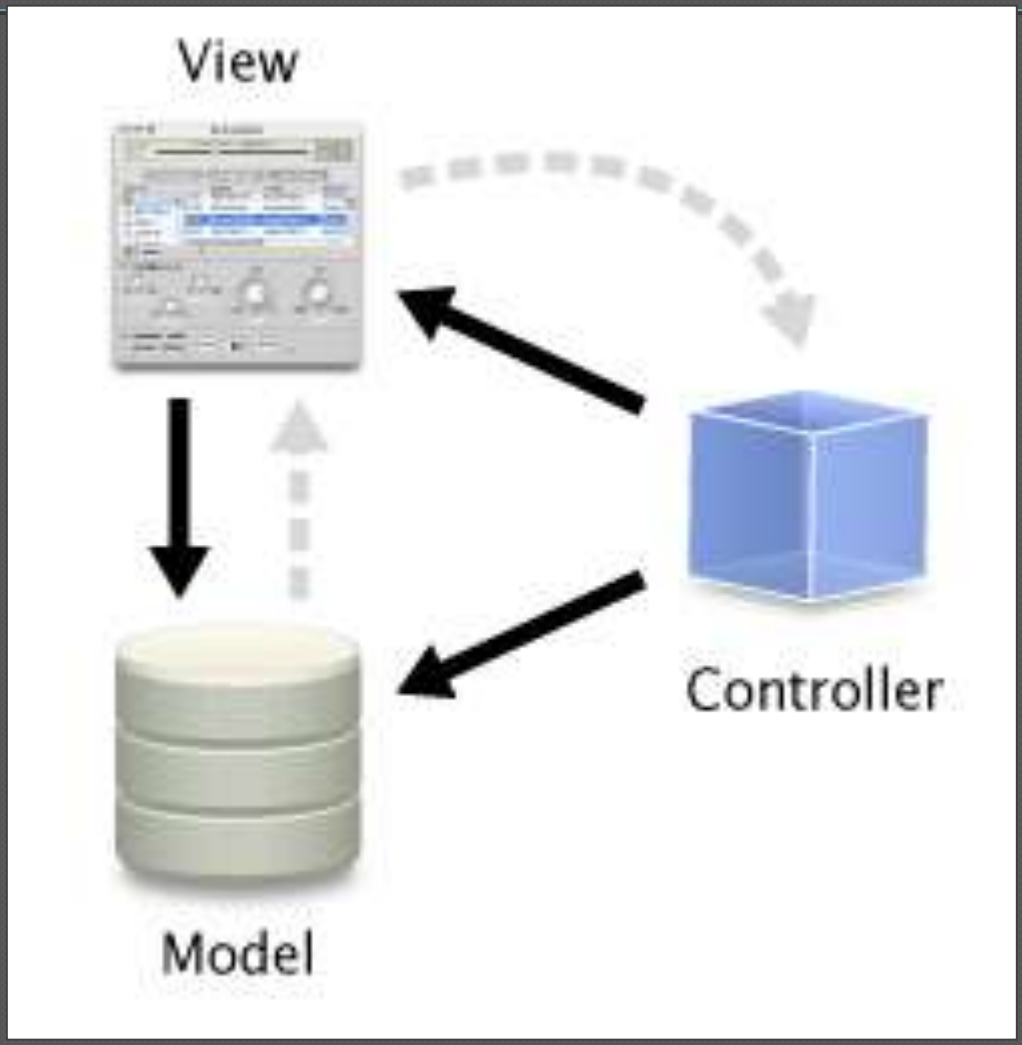
Federico Peinado
www.federicopeinado.es

Depto. de Ingeniería del Software e Inteligencia Artificial
disia.fdi.ucm.es

Facultad de Informática
www.fdi.ucm.es

Universidad Complutense de Madrid
www.ucm.es

Modelo-Vista-Controlador



Arquitectura

- En MVC cada elemento tiene tres partes:
  - Un modelo que contiene los datos y la funcionalidad de la aplicación
    - Juego ajedrez: estado del tablero, reglas del ajedrez, etc.
  - Una vista que gestiona como se muestran esos datos
    - Juego ajedrez: ventana que dibuja el tablero, oyentes de eventos, etc.
  - Un controlador que determina que modificaciones hay que hacer en el modelo cuando se interacciona con la vista. También puede contener algoritmos
    - Juego ajedrez: control de eventos, algoritmo para pensar las jugadas, etc.

Ventajas

- Es posible tener diferentes vistas para un mismo modelo
- Es posible construir nuevas vistas sin necesidad de modificar el modelo subyacente
- Proporciona un *mecanismo de configuración para componentes complejos* mucho más tratable que el puramente basado en eventos (el modelo puede verse como una representación estructurada del estado de la interacción)

Requisitos de reusabilidad

- El modelo no debe ver a ninguna clase de los otros grupos:
  - Se podría cambiar de vista y controlador sin tocar el modelo
- El controlador debe ver las clases del modelo, pero no de la vista → el cambio de vista no afecta al controlador
  - En algunas variantes de la arquitectura el controlador puede ver a la vista por si alguna acción del controlador afecta a la vista pero no al modelo (e.g. mensaje de error)
- La vista no debe ver las clases del modelo → el cambio de modelo no afecta a la vista
  - En algunas variantes la vista ve al modelo para consultarle información, pero nunca para realizar cambios en él

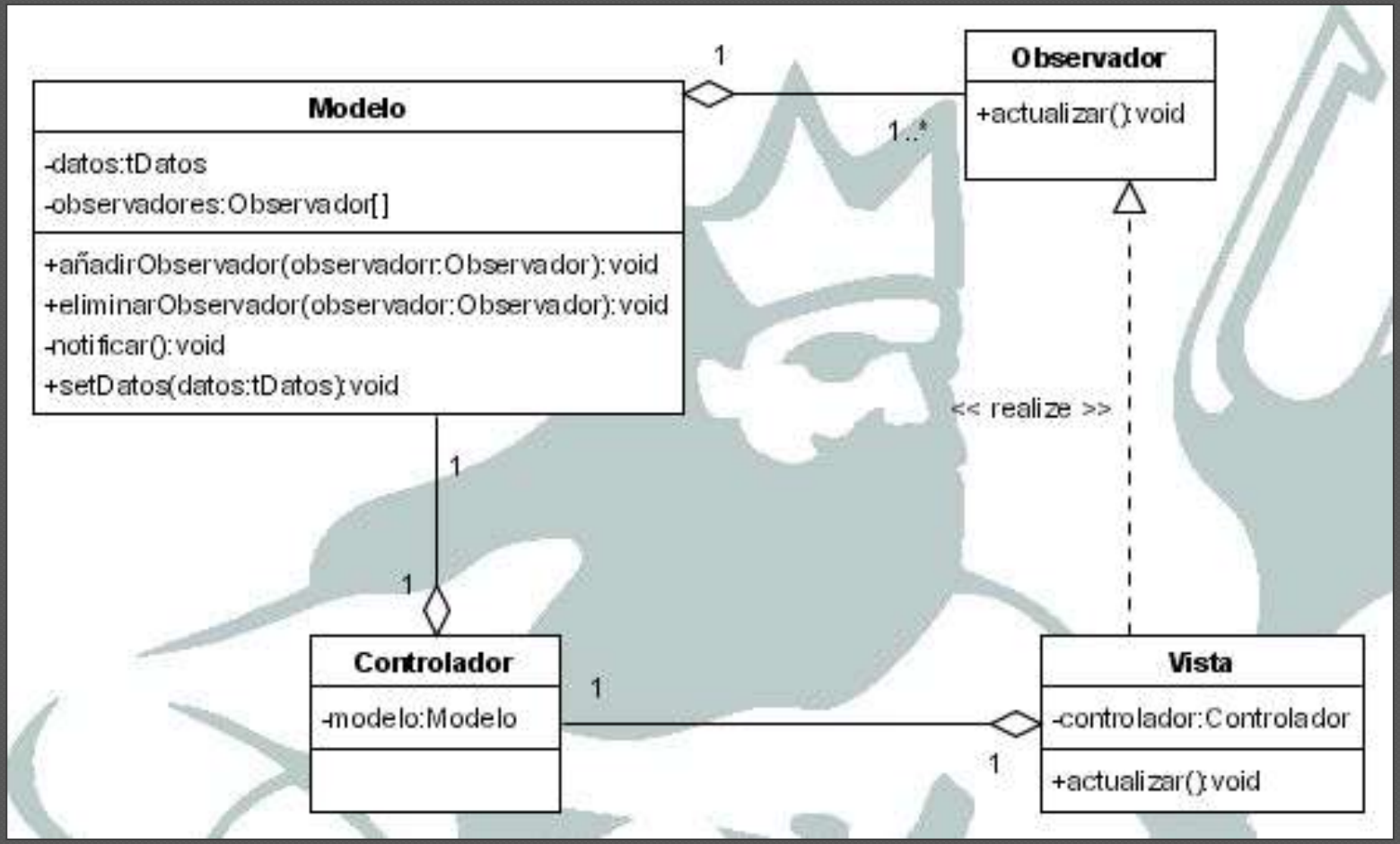
Más sobre arquitectura

- Para que la vista se entere de los cambios producidos en el modelo, se utiliza el patrón Observer
  - La vista se registra como oyente/observador del modelo
- Cuando se produce un cambio en el modelo (setDatos(datos:TDatos)):
  - Se llama al método `notificarXXXX()`
    - Normalmente los métidos `notificarXXXX()` son protegidos o privados.
  - `notificar()` llama al método `actualizarXXX()` de todos los observadores registrados
    - A veces se utilizan otros nombres para `actualizarXXXX()` como `XXXXPerformed()`
  - Los métodos `actualizar()` se encargan de actualizar las respectivas vistas

Más sobre arquitectura

- Los oyentes se pueden seguir implementando como clases internas de la vista
  - Dentro de los manejadores se llamarán a métodos del controlador
- Los métodos notificar y actualizar se pueden desdoblar para distintos tipos de actualizaciones en la vista
- Conviene separar en 3 paquetes las clases correspondientes a cada parte de la arquitectura

Diagrama


Modelo
-datos:tDatos
-observadores:Observador[]
+añadirObservador(observadorr:Observador):void
+eliminarObservador(observador:Observador):void
-notificar():void
+setDatos(datos:tDatos):void
Observador
+actualizar():void
1
1..*
<< realize >>
1
1
Controlador
-modelo:Modelo
1
1
Vista
-controlador:Controlador
+actualizar():void

Uso concreto

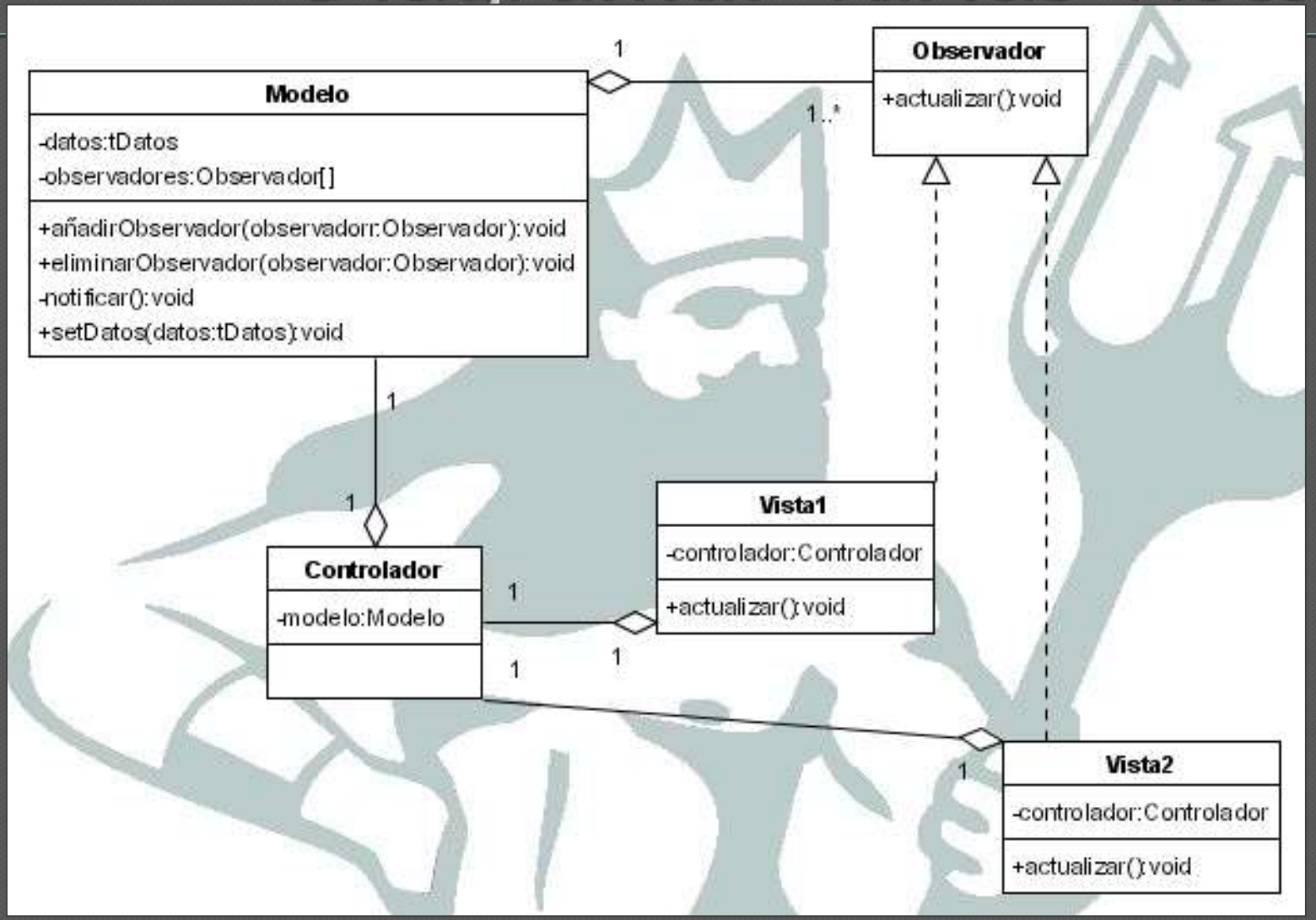
- Para arrancarlo todo en el método main:

```
Modelo modelo = new Modelo();
Controlador controlador = new
  Controlador(modelo);
Vista vista = new Vista(controlador);
modelo.añadirObservador(vista);
```

Posibilidad: Varias vistas

- Esta arquitectura tiene aún más justificación si tenemos varias vistas
  - A través de los observadores registrados en el modelo se actualizan las distintas vistas cuando se producen cambios en el modelo.
    - modelo.añadirObservador(vista1);
    - modelo.añadirObservador(vista2);

Diagrama: Varias vistas



Código: Varias vistas

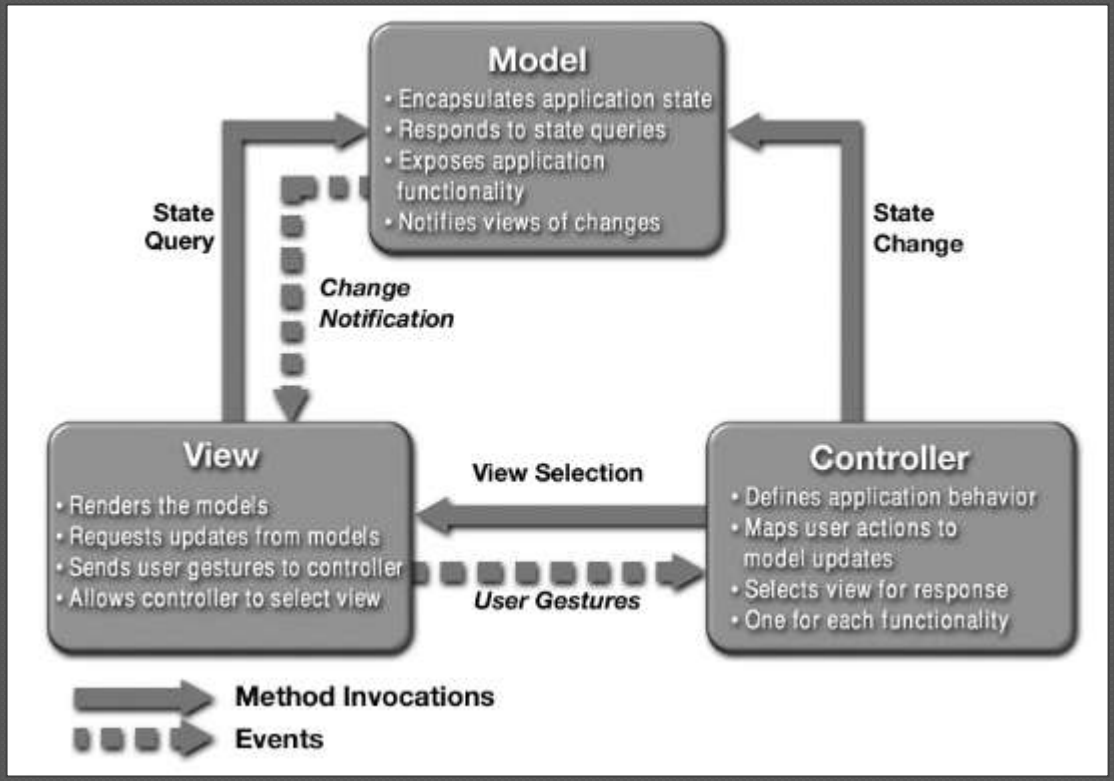
```
public class Aplicación {
  public static void main(String args[]) {
      Modelo modelo = new Modelo();
      Controlador controlador = new
  Controlador(modelo);

      Vista1 vista1 = new Vista1(controlador);
      Vista2 vista2 = new Vista2(controlador);

      Vista1 otraVista1 = new Vista1(controlador);
      Vista2 otraVista2 = new Vista2(controlador);

      modelo.añadirObservador(vista1);
      modelo.añadirObservador(vista2);
      modelo.añadirObservador(otraVista1);
      modelo.añadirObservador(otraVista2);
  }
}
```

Implementación MVC en Java



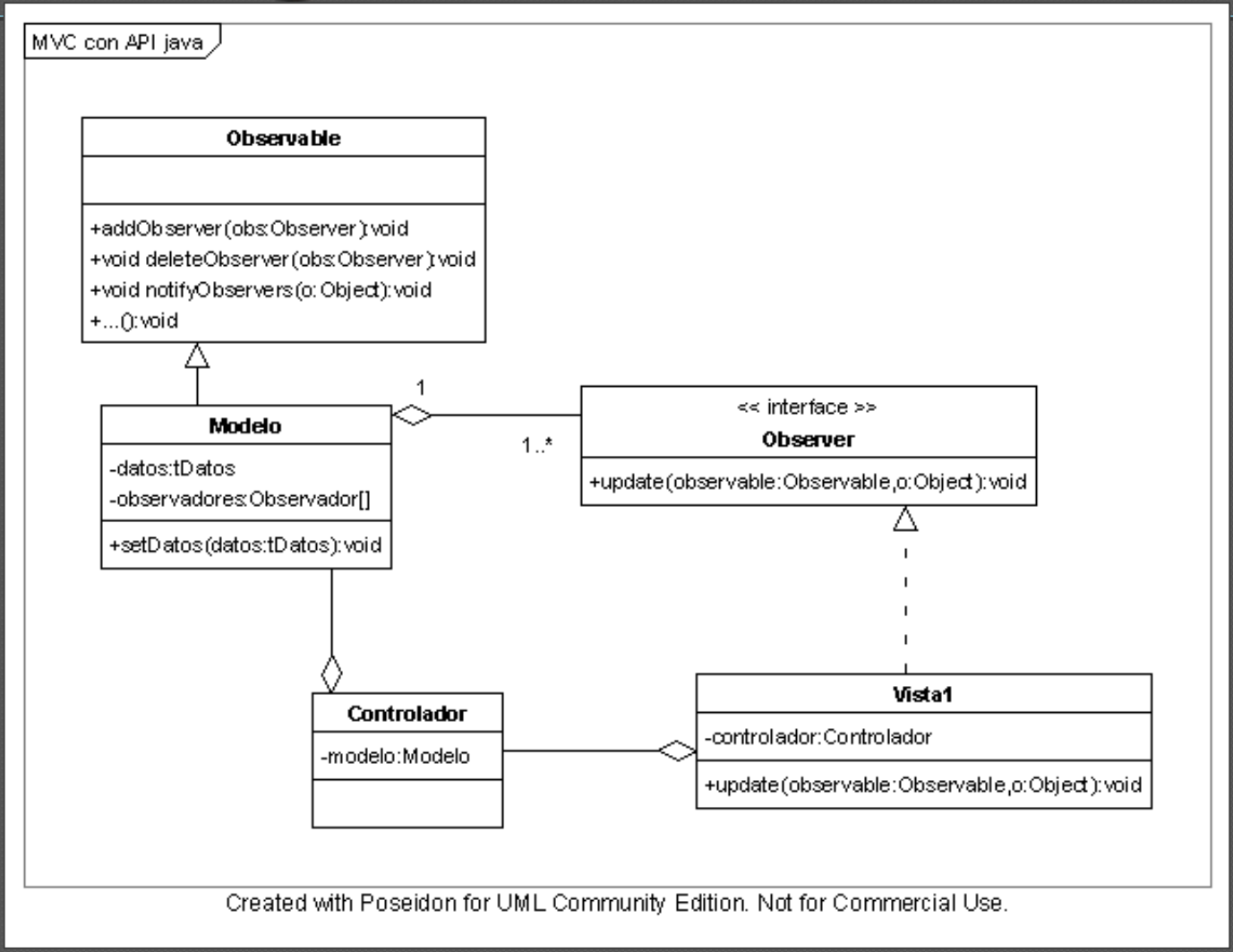
API de Java

- Interface Observer (java.util)
  - `void update (Observable observable, Object o)`
- Class Observable (java.util)
  - Tiene un flag interno que indica si el objeto observable ha cambiado o no
  - Métodos:
    - `void addObserver(Observer obs)`: Añade un observador a la lista
    - `void deleteObserver(Observer obs)`: Quita un observador de la lista
    - `void setChanged()`: Marca el objeto observable como cambiado
    - `void notifyObservers(Object o)`: Si el objeto observable ha cambiado, llama al método `update` de todos los observadores. Vuelve a dejar el objeto observable como no cambiado

MVC con API de Java

- Las vistas deben implementar el interfaz Observer
  - Por tanto, deben implementar el método update
- El modelo debe heredar de Observable
  - No necesita un atributo con la lista de observadores
  - No necesita implementar los métodos para añadir y eliminar observadores
  - No necesita implementar ningún método notificar
  - Cuando se produzca un cambio en el modelo (setDatos(datos:TDatos)) hay que:
    - Actualizar el modelo
    - Indicar que el modelo ha cambiado llamando a setChanged()
    - Notificar a todos los observadores registrados el cambio producido en el modelo llamando a notifyObservers(Object o)

Diagrama con API de Java



Ejemplo: Método main

```
public class Aplicación {
  public static void main(String args[]) {

      Modelo modelo = new Modelo();

      Controlador controlador =
                        new Controlador(modelo);

      Vista1 vista1 = new Vista1(controlador);
      Vista2 vista2 = new Vista2(controlador);


      modelo.addObserver(vista1);
      modelo.addObserver(vista2);
  }
}
```

Ejemplo: Modelo

```
public class Modelo extends Observable {

  private int valor; // Datos

  public void setValor(int nuevoValor) {

      valor = nuevoValor;

      // indica que el modelo ha cambiado
      this.setChanged();

      this.notifyObservers(new Integer(valor));

  }

}
```

Ejemplo: Controlador

```
public class Controlador {

  private Modelo modelo;

  public Controlador (Modelo unModelo){
      modelo = unModelo;
  }

  public void fijarValor(int valor)
                          throws ExcepcionRango {

      if ((valor > 16) || (valor < 0))
            throw new ExcepcionRango();

      modelo.setValor(valor);
  }
}
```

Ejemplo: Vista

```
public class Vista1 extends Jframe implements
  Observer {

  private Controlador controlador;

  (...)

  public Vista1(Controlador unControlador){
      controlador = unControlador;
      configurarComponentes();
      configurarManejadoresEventos();
  }

  public void configurarManejadoresEventos() {
      //Crea Oyentes locales de la interfaz
      //Los oyentes invocan métodos del controlador
  }
```

Ejemplo: Vista

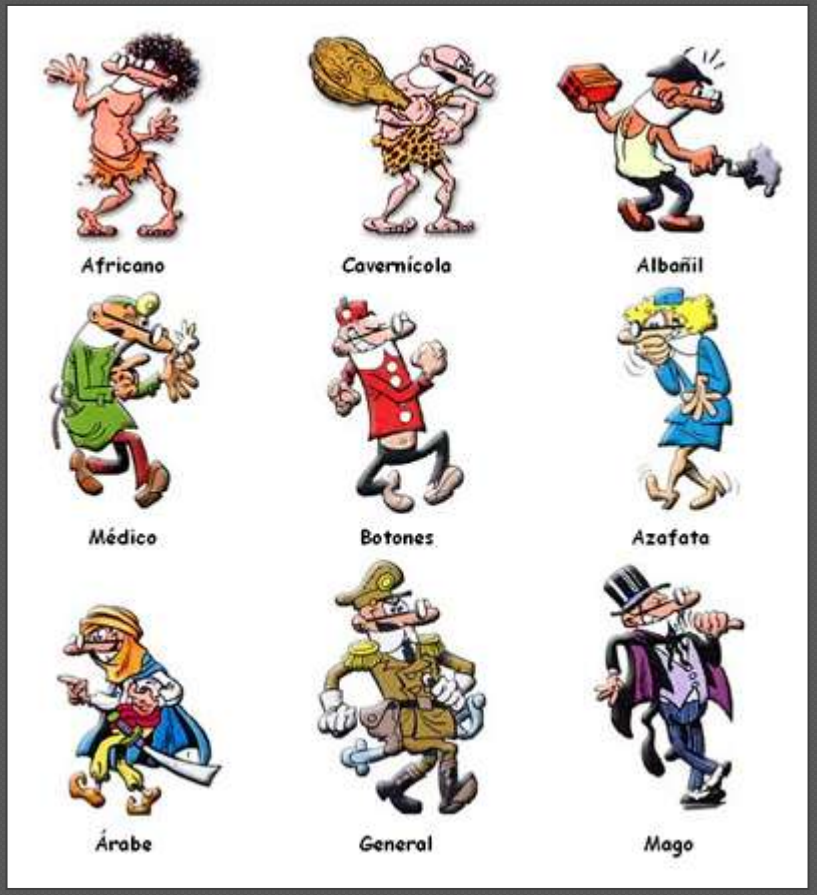
```
  public void update(Observable o, Object valor){
        //Actualización de la interfaz
  }

  public class OyenteVista1
        implements ActionListener {

       public void actionPerformed(ActionEvent e){
          //Procesamiento del evento
          (...)

          try {
              controlador.fijarValor(valor);
          } catch (ExcepciónRango e) {
              JOptionPane.showMessageDialog(null, e);
          }
       }
  }
}
```

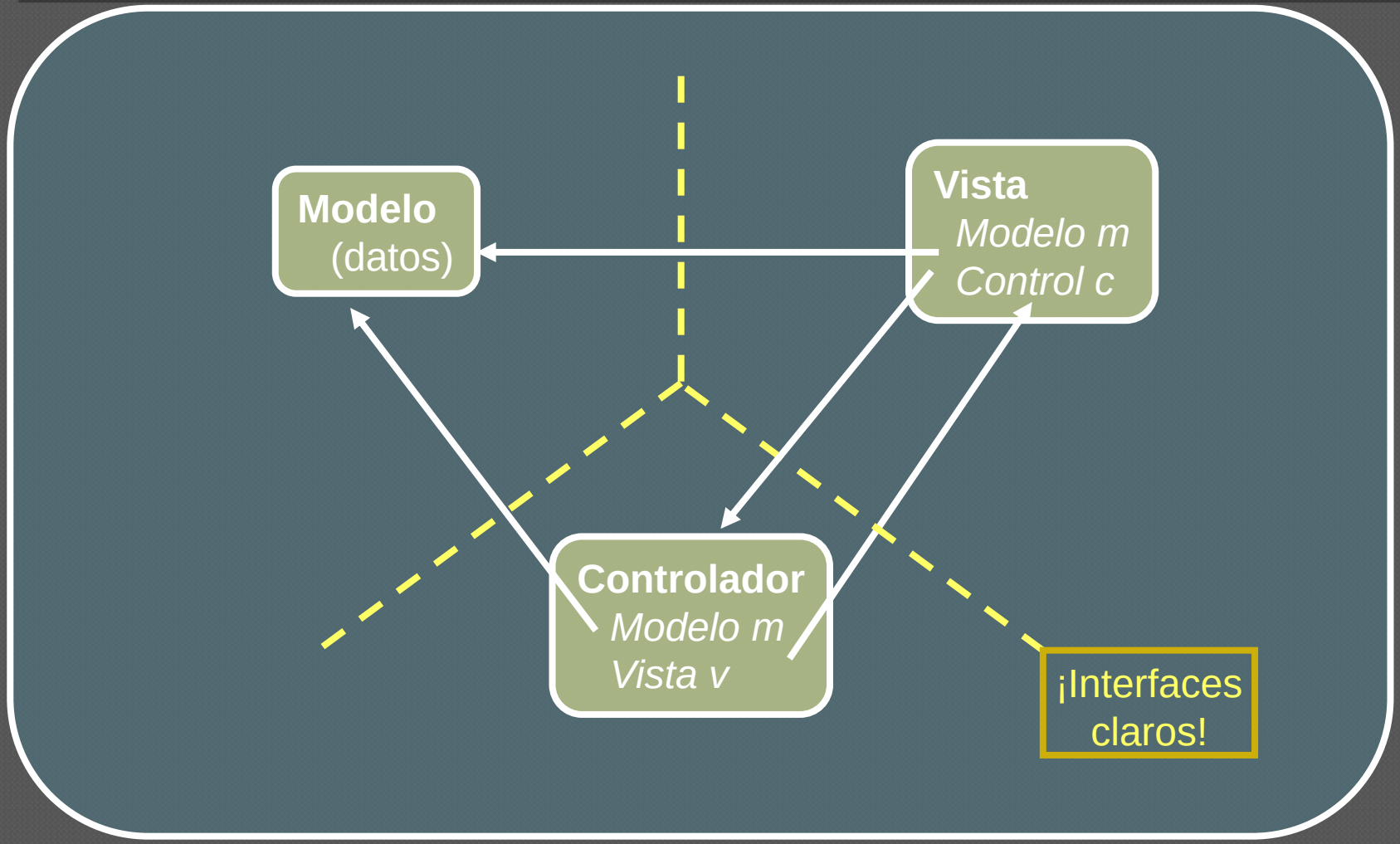
Variaciones



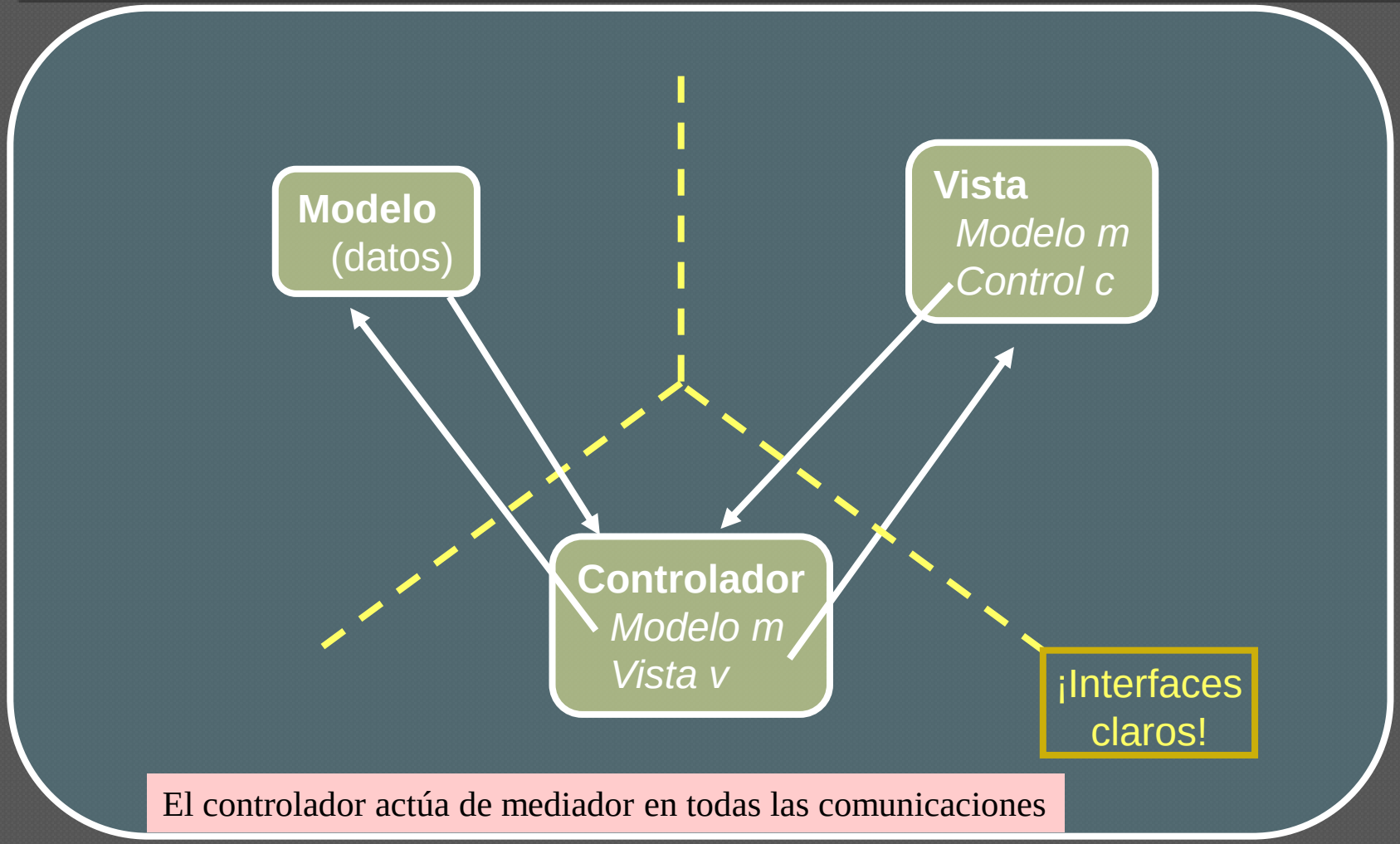
¿Modelo estricto?

- Existen múltiples variaciones e interpretaciones
  - Lo importante es separar los tres elementos
- Problemas típicos:
  - ¿El usuario interactúa con el controlador?
    - El usuario puede considerarse parte de la vista
    - Un servidor remoto puede ser parte de la vista
  - ¿El controlador envía información a la vista?
    - Puede interesar enviar información directamente a la vista desde el controlador (mostrar mensajes de error)
    - Pero entonces el controlador necesita una lista de vistas
  - ¿La vista accede al modelo para preguntar el estado?
    - La vista puede tener una replica del modelo y actualizarla
    - Los eventos pueden llevar una referencia del estado
    - La vista puede tener una referencia al modelo para hacerle preguntas
      - ¡Pero nunca para cambiarlo!

Múltiples referencias



Controlador como mediador

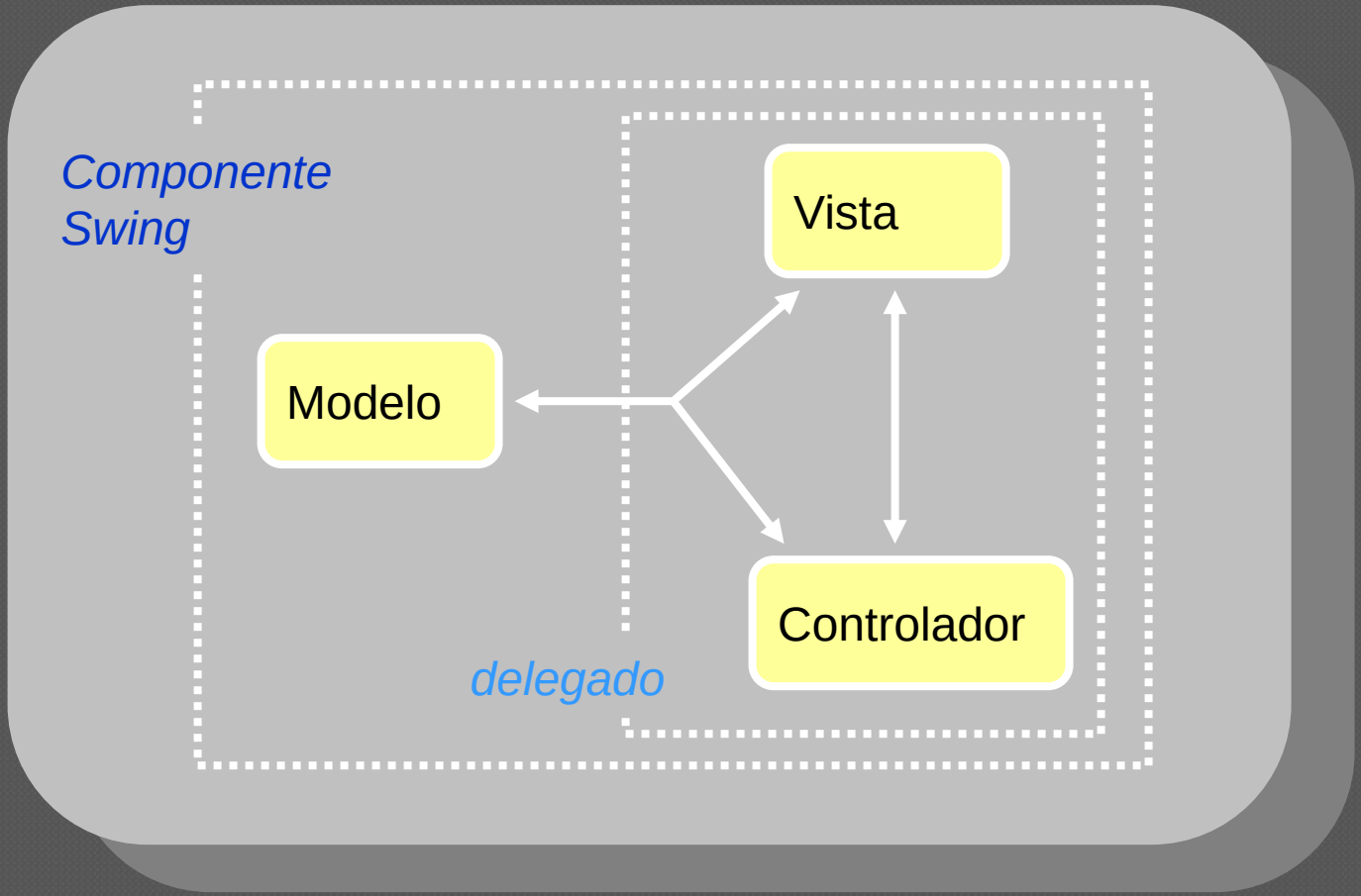


El controlador actúa de mediador en todas las comunicaciones

Swing: Modelo-Delegado

- En Swing se utiliza una adaptación de esta arquitectura de modo que la vista y el controlador se agrupan en el componente (Delegado) pero el modelo se mantiene separado permitiendo comportamientos muy sofisticados
  - Por ejemplo, como los modelos gestionan y almacenan los datos existe la posibilidad de compartir un mismo modelo entre varios componentes. Cada uno de los componentes puede modificar el modelo y dicha modificación se reflejará de forma automática en el resto de los componentes que comparten dicho modelo
  - Estos modelos de datos son especialmente importantes en los componentes que trabajan con texto y en las listas
- El modelo se consulta y actualiza con métodos get<Model> / set<Model> (donde <Model> depende del tipo de componente)
- El delegado se consulta/actualiza con los métodos getUI / setUI

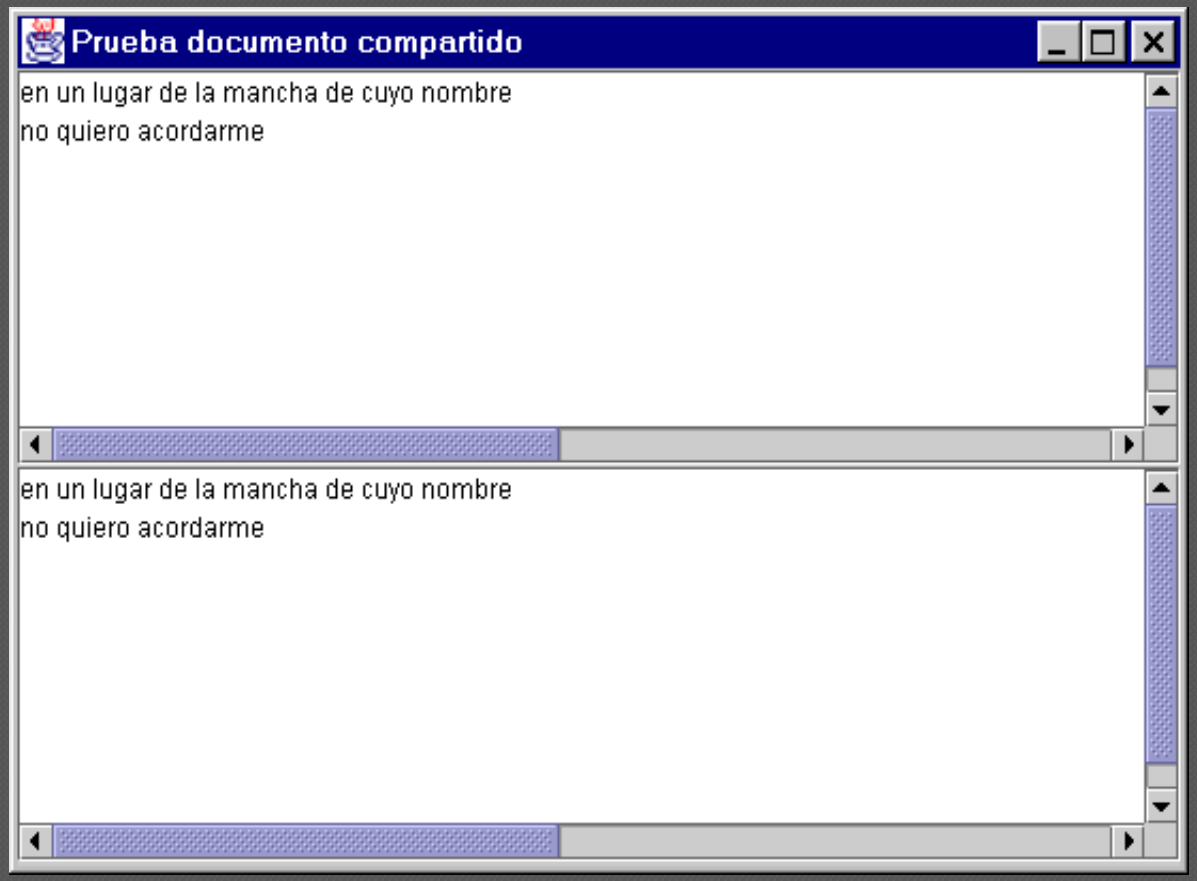
Swing: Modelo-Delegado



Ejemplo de modelo compartido en Swing

```
public class ModeloCompartido {
    JTextArea areaTexto1, areaTexto2;
    JFrame ventana;
    Document documento;
    public ModeloCompartido(String titulo){
   ventana = new JFrame(titulo);
   Container panelContenido = ventana.getContentPane();
   areaTexto1= new JTextArea(10,100);
   documento=areaTexto1.getDocument();
   areaTexto2= new JTextArea(documento);
   areaTexto2.setColumns(100);
   areaTexto2.setRows(10);
   panelContenido.setLayout(new BoxLayout(panelContenido,
   BoxLayout.Y_AXIS));
   panelContenido.add(new JScrollPane(areaTexto1));
   panelContenido.add(Box.createGlue());
   panelContenido.add(new JScrollPane(areaTexto2));
   panelContenido.add(Box.createGlue());
   ventana.setSize(500, 300);   ventana.setVisible(true);
    }
   public static void main(String args[]) {
      ModeloCompartido aplicacion = new ModeloCompartido("Prueba documento
   compartido");  }
}//ModeloDocumentoCompartido
```

Ejemplo de modelo compartido en Swing



Críticas, dudas, sugerencias…

Federico Peinado

www.federicopeinado.es