

LPS: Modelo de Eventos

Create an Event

Step 1
Create Event

Step 2
Add Details

Step 3
Invite Friends



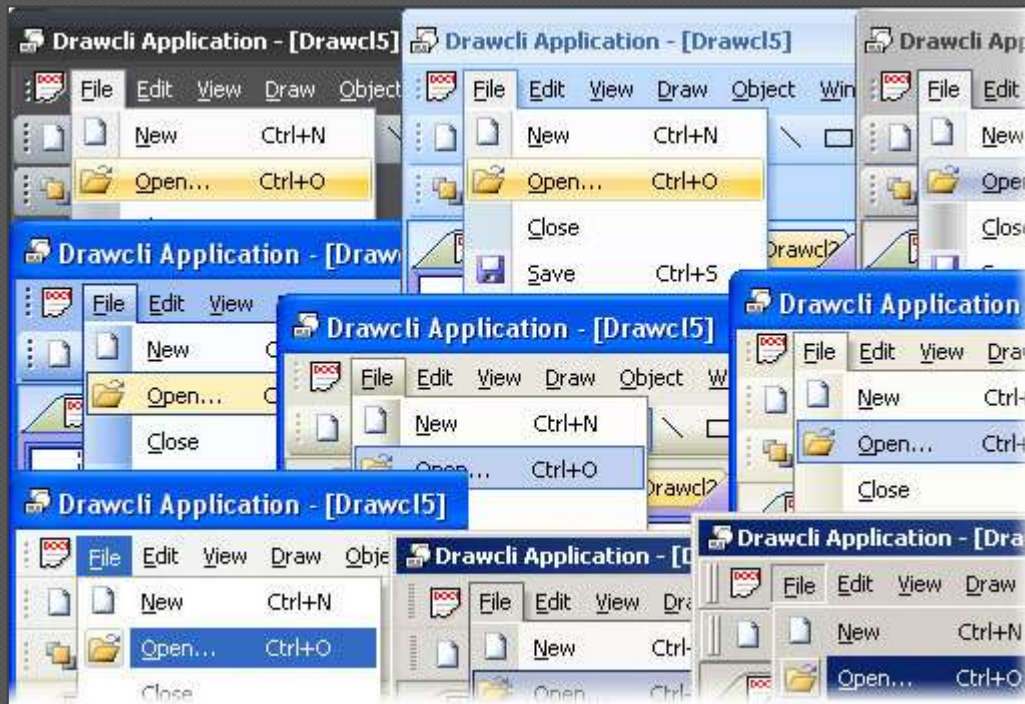
Federico Peinado
www.federicopeinado.es

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

Funcionamiento de una GUI



Funcionamiento de una GUI

- ◉ Una **interfaz gráfica de usuario** (GUI, *Graphic User Interface*) en Java funciona en 3 pasos:
 1. **Composición**
 - Se crea un contenedor para los elementos de la interfaz
 - Se crean y añaden los componentes de la interfaz que aportan la representación visual y la interacción (botones, menús, etc.)
 2. **Establecimiento de los gestores de eventos**
 - Se crean objetos que responderán a las acciones de los usuarios, llamando a los objetos “lógicos” de la aplicación
 3. **Ejecución en un hilo propio**
 - Se lanza un hilo en el se ejecutarán permanentemente los gestores de eventos, pendientes de las acciones del usuario
 - Por ello, el método principal de la aplicación suele crear la GUI, hacerla visible y terminar su ejecución sin más

Tipos de interacción

- ◉ **Clásica: Interacción controlada por la aplicación**
 - La aplicación es quien solicita al usuario los *datos de entrada* en el preciso momento en que las necesita
 - Típico de las aplicaciones de *línea de comandos* o *guiadas por menús* que hemos usado hasta ahora
 - Ventajas: Sencillo de programar
 - Desventajas: Rigidez, usabilidad más restringida
- ◉ **Moderna: Interacción controlada por el usuario**
 - El usuario es quien actúa sobre los *controles* ofrecidos por la aplicación en el momento y de la manera en que prefiere
 - Habitual en las aplicaciones con interfaces gráficas de usuario, como ocurre en Java + Swing
 - Ventajas: Énfasis en la experiencia del usuario
 - Desventajas: Más difícil de programar, debido a la **inversión del control**: el flujo de ejecución pasa de ser llevado por la aplicación a estar en manos del usuario

Modelo de eventos



Modelo de eventos

- ◉ La dificultad que añade esta *interacción controlada por el usuario* puede ser tratada programando según un **modelo de eventos**
 - Cada **acción del usuario** sobre la GUI producirá un evento
 - Movimiento del ratón
 - Clic del ratón sobre un botón del interfaz
 - Escritura de caracteres dentro de un campo de texto
 - ...
 - Un *evento* será **un objeto** que representa un **mensaje asíncrono** que tiene **otro objeto como destinatario**
 - Programar *basándonos en este modelo* consistirá en definir las operaciones correspondientes a la gestión de estos mensajes

Fuentes y oyentes

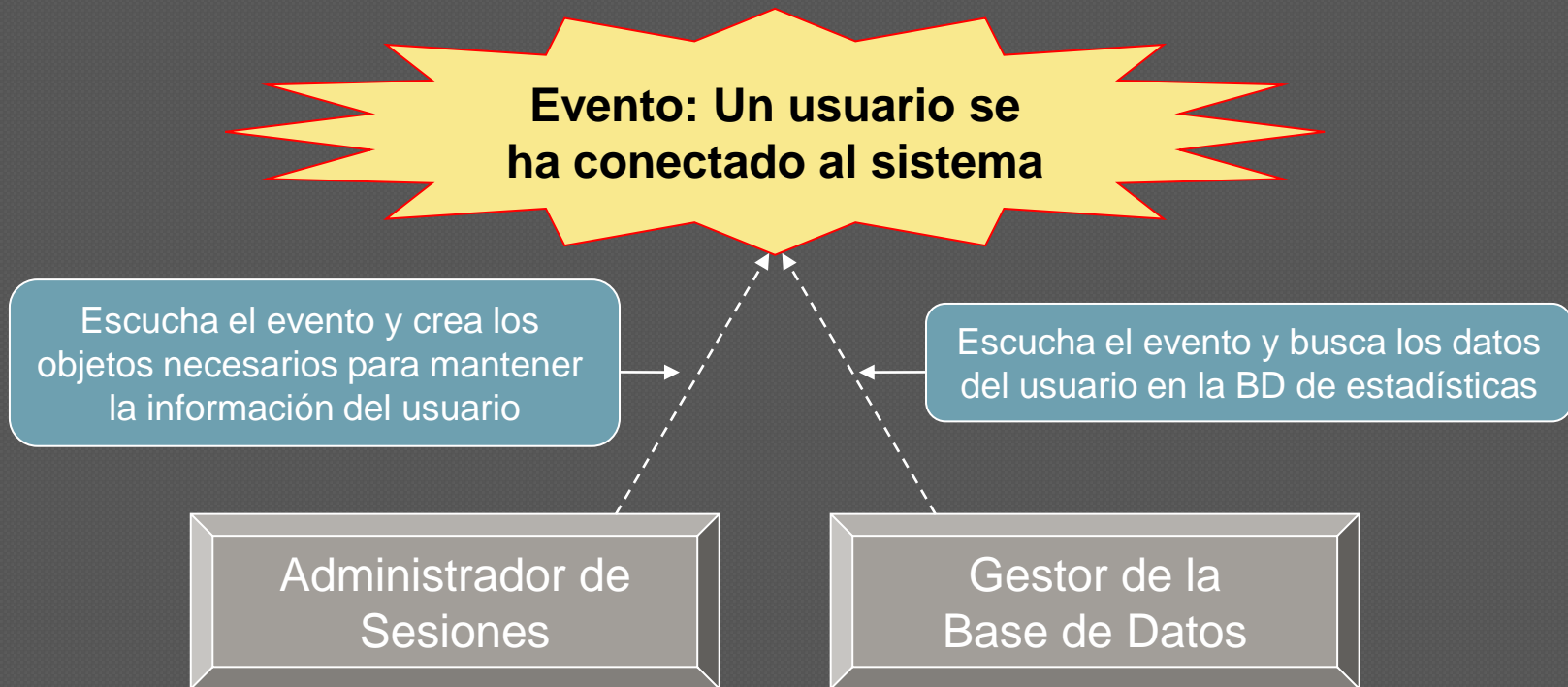
- ◉ Los eventos se generan en **objetos llamados “fuentes”** y delegan la responsabilidad de gestionarlos en otros **objetos llamados “oyentes”**
 - Los oyentes se “registran” en las fuentes para cierto tipo de eventos
 - Habitualmente esto se traduce en implementar los métodos de ciertos interfaces
 - La fuente *notifica* sus eventos a todos los oyentes convenientemente registrados, pero no comprueba qué se hace con ellos ni quien lo hace
 - Invocando un método del oyente, pasándole el evento como argumento
 - El oyente recibe el evento, un objeto que encapsula información:
 - *Tipo de evento* (e.g. un clic derecho del ratón)
 - *Referencia a la fuente del evento* (e.g. un botón de “Aceptar”)
 - *Instante de tiempo en el que se produjo el evento*
 - *Posición en pantalla (x, y) asociada al evento*, si es que la hay
 - ...

Esquema



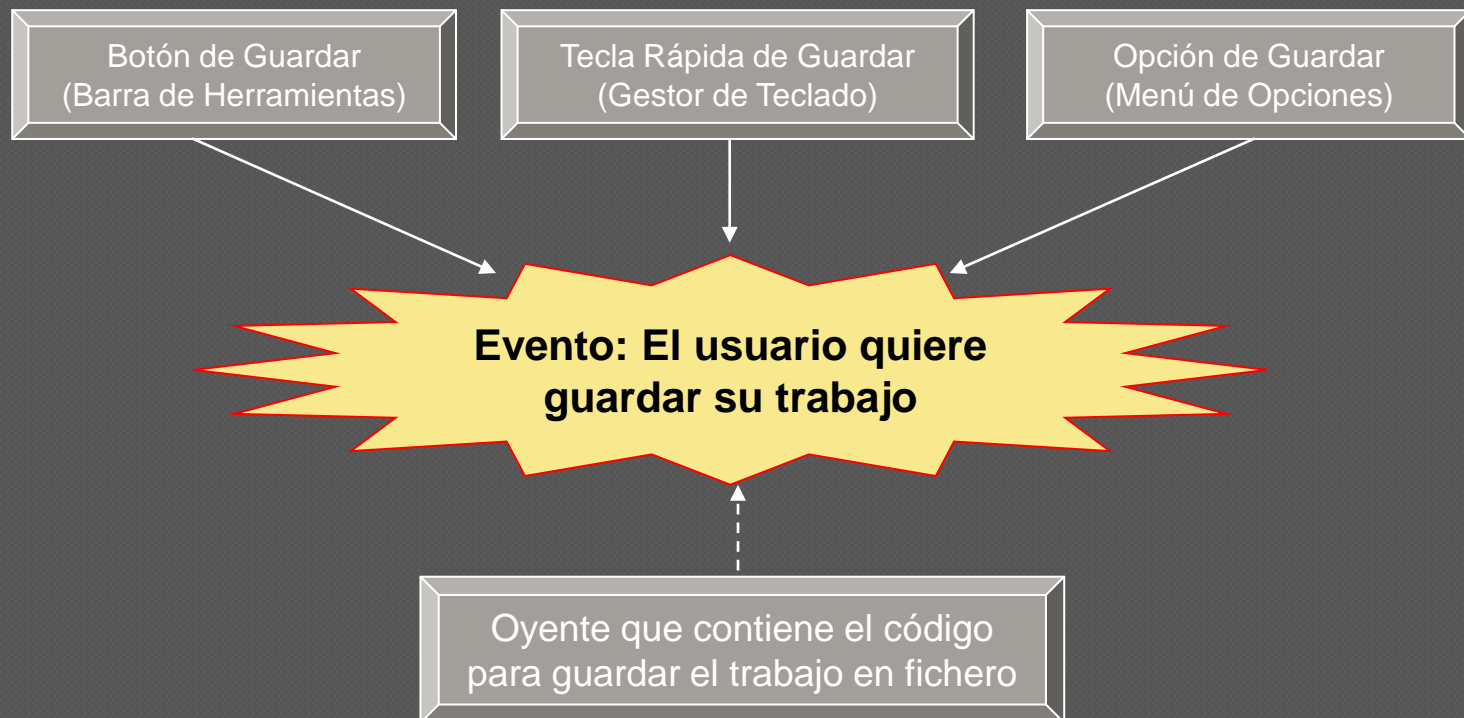
Ventajas

- **Flexibilidad:** Varios objetos de distintas clases pueden registrarse como oyentes de un mismo tipo de evento



Flexibilidad

- **Flexibilidad:** Varios objetos fuente de distintas clases pueden notificar el mismo tipo de evento



Modelo de eventos en Java



Modelo de eventos en Java

- ◉ Desde Java 1.1 (1997), usado en AWT
- ◉ Objetivos
 - Ser **simple y fácil de aprender**, pero a la vez versátil
 - Permitir una **separación clara entre distintos fragmentos de código** sin impedir que se comuniquen
 - Facilitar la creación de **código robusto** para la gestión de eventos y flujos de programa asíncronos
 - Ofrecer **mecanismos de delegación** de responsabilidades

Fuentes y oyentes

○ Fuente (*Source*)

- Cada fuente define el tipo de eventos que notifica y proporciona métodos para registrar y eliminar oyentes de dicho tipo
 - `fuente.addTipoEventoListener(oyente)`
Añade un oyente a la lista de los registrados
 - `fuente.removeTipoEventoListener(oyente)`
Elimina un oyente de la lista de los registrados

○ Oyente (*Listener*)

- Cada oyente implementa el interfaz `TipoEventoListener` y uno o más métodos a los que la fuente llama cuando *notifica* que se ha producido dicho tipo de evento

```
public void TipoEventoPerformed(TipoEventoEvent e) {  
    // Código que implementa la respuesta al evento "e"  
}
```

Tipos de eventos

- ◉ Cada tipo de evento se crea con una **clase Java**
 - Algunas clases representan todo un grupo de eventos (e.g. **MouseEvent** puede representar un “mouse up”, “mouse down”, “mouse drag”, “mouse move”, etc.)
- ◉ En Java existen **jerarquías de eventos y de oyentes predefinidas**
 - Raíz de la jerarquía de los eventos:
java.util.EventObject
 - Extendiendo esta clase pueden definirse nuevos eventos
 - Interfaz común para todos los oyentes:
java.util.EventListener

Ejemplo con Swing

```
/**
 * Clase Swing que implementa una ventana principal
 * sencilla que contiene una etiqueta y dos botones
 */
public class VentanaSimple extends JFrame {
    JLabel etiqueta;           // Etiqueta
    JButton botonHola;        // Botón de interacción
    JButton botonAdios;       // Botón de interacción

    // Panel contenedor de los anteriores elementos
    JPanel panel;
```



Ejemplo con Swing

```
/**
 * Constructor de la clase VentanaSimple
 */
public VentanaSimple () {
    // Se crean los componentes de la ventana
    etiqueta = new JLabel("Etiqueta inicial");
    botonHola = new JButton("Hola");
    botonAdios = new JButton("Adiós");
    panel = new JPanel();

    // Se añaden los componentes al panel
    panel.add(etiqueta);
    panel.add(botonHola);
    panel.add(botonAdios);

    // añade el panel a la ventana principal de la aplicación
    getContentPane().add(panel);

    // Se crea el oyente de la "acción" de los botones y se registra
    OyenteAccion oyenteBoton = new OyenteAccion();
    botonHola.addActionListener(oyenteBoton);
    botonAdios.addActionListener(oyenteBoton);
}
```


Ejemplo con Swing

```
/**
 * Método principal de la clase VentanaSimple
 */
public static void main(String args[]) {
    // Se crea un objeto de tipo VentanaSimple
    VentanaSimple ventana = new VentanaSimple();
    ...
}

/**
 * Oyente de eventos de acción de botón
 * (Clase interna privada de VentanaSimple)
 */
private class OyenteAccion implements ActionListener {
    public void actionPerformed (ActionEvent evento){
        // Se obtiene el botón fuente del evento
        JButton boton = (JButton) evento.getSource();
        // Se modifica la etiqueta según el botón pulsado
        etiqueta.setText("Botón pulsado: " + boton.getText());
    }
}
```

Interfaces de múltiples eventos

- A menudo las interfaces de los oyentes **agrupan muchos métodos para responder a eventos muy distintos**
 - Ejemplo: `MouseListener` agrupa los métodos para todos los eventos relacionados con el ratón
 - `mouseClicked()`
 - `mouseEntered()`
 - `mouseExited()`
 - ...
- Problema: Estas interfaces **nos obligan a implementar muchos métodos (vacíos)** aunque nosotros sólo queramos tratar un tipo concreto de ellos

```
class OyenteRaton implements MouseListener {
    public void mouseClicked(MouseEvent e) {
        // Procesar clic del ratón
    }
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
}
```

Clases adaptadoras

◉ Solución: En Java lo que suele hacerse es utilizar **clases adaptadoras**

- Implementaciones parciales de las interfaces de oyentes, donde el código de todos los métodos está *vacío* o es *trivial*
- Ejemplo: **MouseAdapter** (que implementa la interfaz mencionada antes, **MouseListener**)

```
class OyenteRaton extends MouseAdapter {  
    public void MouseClicked(MouseEvent e) {  
        // Procesar clic del ratón  
    }  
    // Los demás métodos están vacíos en MouseAdapter  
}
```

¿Demasiadas clases?

- Las GUIs pueden llegar a tener muchos componentes gráficos y eventos posibles, y se necesita **una clase diferente para cada oyente**
 - En Swing estos componentes son fuentes, aunque también pueden ser oyentes
 - Esto obliga a definir muchas clases oyentes, causando confusión al programador
- La solución en Java es usar clases anidadas (*interiores*, e incluso *anónimas*)

```
public class AdaptadorClaseInterior extends JFrame {
    public AdaptadorClaseInterior() {
        setTitle("Ventana que se puede cerrar");
        setSize(300, 100);
        addWindowListener(
            new WindowAdapter() { // Gestor anónimo de ventana
                public void windowClosing(WindowEvent e) {
                    System.exit(0); // Salida del programa
                }
            } );
        setVisible(true);
    }
    ...
}
```


Críticas, dudas, sugerencias...



Federico Peinado

www.federicopeinado.es