

Paradigmas de Programación

3.2. Orientación a Eventos

Departamento de Informática
Universidad de Valladolid

Curso 2020-21

Grado en Ingeniería Informática
Grado en Ingeniería Informática de Sistemas





Prog. Orientada a Eventos

Es un paradigma de programación en el que el **flujo del programa** está determinado por **eventos**:

- Acciones del usuario.
- Mensajes de otros programas.
- Activación de sensores.
- ...

Bajo este paradigma un programa consiste en un conjunto de bloques de código (subrutinas, métodos) que son invocados al producirse un tipo evento y que pueden acceder a un estado común.



Arquitectura Orient. a Eventos

- En una arquitectura orientada a eventos, existe una entidad externa al programa (el **controlador**), que se encarga de:
 - Detectar eventos.
 - Determinar los programas que están interesados en ellos y/o pueden manejarlos.
 - Enviar **mensajes** a esos programas o invocar al código del programa encargado de **manejarlos**.
- Arquitecturas típicas O.E.:
 - **Interfaces Gráficas de Usuario** (GUIs).
 - Entornos Web
 - Gestores de Bases de Datos Distribuidos
 - Sistemas Empotrados



Interfaces Gráficas de Usuario

- En una GUI, pueden existir múltiples aplicaciones ejecutándose simultáneamente:
 - Cada una de ellas tiene asignada una zona de la pantalla (ventana) y no deben dibujar fuera de sus límites.
 - Todas usan una serie de controles cuya apariencia y funcionalidad debe ser similar.
 - El acceso a recursos del sistema y a la entrada por parte del usuario (ratón, teclado) no puede dejarse a cargo de cada aplicación.
- Solución:
 - El sistema operativo es el que tiene el control sobre la entrada del usuario, los recursos, la visualización y funcionamiento de controles y el acceso a la pantalla.
 - Informa a las aplicaciones de los eventos relevantes.



Paso de Mensajes (I)

- En esta variante cada evento es traducido por el controlador en un **mensaje** (típicamente un registro u objeto) que encapsula los datos relevantes.
- Cada aplicación (proceso activo) tiene asociada una **cola de mensajes**.
- El mensaje se añade a las colas de mensajes de las aplicaciones a las que afecta el evento.
 - Porque estas aplicaciones han registrado su interés por el evento.
 - Porque el estado del sistema determina que es la aplicación relevante (foco del teclado, pulsación del ratón en región ocupada por la ventana)
 - Porque se han producido cambios que la afectan (redibujado por desplazamiento de otra ventana)



Ejemplo de mensaje

- Definición de un mensaje en Win32s (C):

```
typedef struct tagMSG {  
    HWND    hwnd;        // Identificador de ventana (control)  
    UINT    message;     // Tipo de mensaje  
    WPARAM  wParam;      // Datos del mensaje  
    LPARAM  lParam;      // Datos del mensaje  
    DWORD   time;        // Momento en que se ha producido  
    POINT   pt;          // Posición en pantalla  
} MSG;
```

WM_CHAR → Carácter pulsado (en wParam)
WM_CLOSE → Pulsado botón de cierre de ventana
WM_COMMAND → Acción sobre control (tipo e identificador en wParam)
WM_DROPFILES → Arrastre de fichero sobre control
WM_GETTEXT → Copiar texto de un control en buffer lParam
... (500 tipos más)



Paso de Mensajes (II)

- Una aplicación consiste en un bucle (**bucle de mensajes**) donde se accede (mediante una llamada a una subrutina del controlador) a la cola de mensajes y se ejecuta el código adecuado para **tratarle**.
- Mientras se trata un mensaje, el controlador puede añadir nuevos mensajes a la cola.
- El bucle termina cuando se recibe un **mensaje de finalización**.
- En sistemas de **multitarea cooperativa**, si al acceder a la cola ésta está vacía, se transfiere el control a otras aplicaciones (la llamada no retorna hasta tener mensajes).
- La aplicación puede enviar mensajes al controlador para obtener información o comunicarse con otras aplicaciones.



Estructura de una aplicación

- Una aplicación Win32s consiste en:
 - Una función (WinMain) que define el punto de entrada del programa, y que realiza las siguientes tareas:
 - Registra la ventana principal (RegisterClass) indicando una subrutina (WndProc) encargada del proceso de cada mensaje.
 - Se crea y muestra la ventana principal y sus controles hijos (CreateWindow, ShowWindow, UpdateWindow).
 - Se ejecuta el bucle de proceso de mensajes:

```
MSG msg;  
while( GetMessage( &msg, NULL, 0, 0 ) ) {  
    TranslateMessage( &msg );  
    DispatchMessage( &msg );  
}
```

Llama a la subrutina
pasada en RegisterClass



Manejadores de Eventos

- En esta otra variante, la aplicación **registra** pares (**evento**, **manejador**) al controlador para indicarle que invoque al manejador cuando se produzca un evento de ese tipo.
- Un **manejador** suele consistir en una subrutina o método con unos parámetros predeterminados que reciben datos relevantes sobre el evento.
- El bucle de mensajes, si existe, está oculto al programador (suele usarse en frameworks que establecen una capa sobre el código nativo):
 - Entornos de desarrollo visuales (Delphi, Visual Studio)
 - Librerías AWT y Swing de Java
 - Framework GTK (llamada a **connect**), wxPython (llamada a **Bind**)



Ejemplo (Python – wxPython)

```
...
def crea_interfaz(self):
    # Creación del control
    boton = wx.Button(...);
    # Registro de evento <-> manejador
    boton.Bind(wx.EVT_BUTTON, self.pulsado);
    ...
def pulsado(self, event):
    # Manejador del evento de pulsación del botón
    ...
```



Ejemplo (Java – Swing)

```
public class Aplicacion implements ActionListener {  
    JButton boton;  
  
    public void creaInterfaz() {  
        // Creación del control  
        boton = new JButton("..");  
        // Registro de evento <-> manejador  
        boton.addActionListener(this);  
    }  
  
    // Método especificado en ActionListener  
    public void actionPerformed(ActionEvent e) {  
        // Manejador del evento de pulsación del ratón  
        if(e.getSource() == boton) { ... }  
    }  
}
```



Técnicas para P.O.E.

Para implementar la programación orientada a eventos es conveniente poder enviar referencias a código de la aplicación a otras entidades (controlador).

- **Callbacks:** Se puede almacenar referencias a subrutinas en variables (parámetros) de forma que se puedan invocar las subrutinas mediante ellas.
- **Orientación a Objetos:** Un objeto encapsula datos (atributos) y código (métodos). Es posible enviar código haciendo enviando un objeto y señalando uno de sus métodos como el manejador, usando herencia y polimorfismo.



Callbacks – Tipos Procedurales

- Un **Tipo procedural** es un tipo de datos que representa referencias a subrutinas con una determinada cabecera (parámetros y resultado).

```
type
  TProcFila = procedure(var Fil: Tfila; I,N: integer);

  procedure proc1(var Fil: Tfila; I,N: integer);
  begin
    ...
  end;

var
  f: Tfila; p: TProcFila;
begin
  p := proc1;
  p(f,1,5);    // Igual que proc1(f,1,5)
end.
```



Orientación a Objetos (I)

- La orientación a objetos permite enviar un objeto como parámetro con el objetivo de que se ejecute un método del objeto enviado.
- El problema de un enfoque directo es que entre la aplicación y el controlador se produce una **agregación cruzada**:

APLICACIÓN

```
• CONTROLADOR c
+ manejador(..)
+ registrador() {
    c.registra(this);
}
```

CONTROLADOR

```
• APLICACION a
+ registra(APLICACION app) {
    a = app;
}
+ control(..) {
    a.manejador(..);
}
```



Orientación a Objetos (II)

- La solución es usar herencia y polimorfismo para que el controlador no conozca los detalles del objeto que son irrelevantes para su tarea.

