Cursos de Extensión Universitaria
UNIVERSIDAD DE OVIEDO

PROGRAMACIÓN ORIENTADA A OBJETOS CON C# EN LA PLATAFORMA .NET

Delegados y Eventos

César Fernández Acebal
acebal@ieee.org

Dpto. de Informática
OOTLab - Laboratorio de Tecnologías Orientadas a Objetos
www.ootlab.uniovi.es

Introducción

- No debemos confundir delegados y eventos
- Una cosa es el modelo de eventos de .NET y otra los delegados
  - Éstos son la forma escogida por la plataforma para implementar su modelo de eventos
  - Pero son un mecanismo genérico que puede tener aplicaciones bien distintas
    - Podemos asimilarlos a las típicas *funciones callback*

¿Qué vamos a ver?

- Comenzaremos viendo cómo manejar los eventos predefinidos en .NET
  - En concreto, trabajaremos con el evento *Paint*
- Una vez familiarizados, aprenderemos a definir nuestros propios eventos
- Por el medio, se irán introduciendo los conceptos sobre delegados que necesitamos para trabajar con los eventos
- Por último, se estudiarán los delegados por sí solos
  - Conceptos avanzados, otros usos posibles, etc.

Eventos

Comenzaremos viendo cómo podemos manejar los eventos predefinidos en .NET, para entender cuál es el modelo de eventos del CLR, basado en *delegados*.

Posteriormente, veremos cómo crear y tratar nuestros propios eventos.

Entrada dirigida por eventos

- Los programas de consola que necesitan interactuar con el usuario lo hacen a través de los métodos Read o ReadLine
  - De la clase Console
- Pero los programas escritos para entornos gráficos tienen un modelo de entrada de usuario diferente
  - Hay varios dispositivos de entrada diferentes
    - Ratón y teclado
  - Hay varios controles que interactúan
    - Botones, menús, barras de desplazamiento…

Técnica de muestreo (*serial polling*)

- En teoría, podría usarse la técnica de muestreo (*polling*) para controlar todos los dispositivos:
  - Se comprueba la entrada del teclado; si no hay nada, el ratón; si no, el menú; éste mira si hay algo en la entrada del teclado o del ratón… y así sucesivamente
    - Así es como lo hacían, por ejemplo, los programas en modo texto (no Windows) que hacían uso del ratón

Modelo dirigido por eventos

- Tal y como se ha implementado en *Windows Forms*, cada tipo de entrada está asociada a un método diferente de una clase
- Cuando ocurre algo a la entrada, se llama al método apropiado automáticamente

Mismo hilo de ejecución

- A primera vista, esto podría parecer caótico
- ¿No se corre el riesgo de que el programa se vea desbordado por muchas llamadas simultáneas a métodos diferentes?
  - No, porque todas ellas tienen lugar en el mismo hilo de ejecución
  - Sólo se atiende a un evento cuando finaliza el procesamiento del evento anterior
    - Es decir, cuando finaliza el método al que aquél dio lugar
    - No interrumpen, por tanto, la ejecución del programa

Modelo de eventos

- Una vez que una aplicación de ventanas hecha con *Windows Forms* se inicializa, todo lo que el programa hace lo hace en respuesta a eventos
  - En vez de ejecutarse secuencialmente
- En la plataforma .NET, los *eventos* son miembros de las clases
  - Junto con los *constructores*, *campos*, *métodos* y *propiedades*

Manejadores de eventos y delegados

- Cuando un programa define un método para procesar un evento, a este método se le denomina manejador de eventos
  - *"event handler"*
- Los parámetros del manejador se ajustan a una definición de un prototipo de función, lo que se conoce como delegado
  - *"delegate"*

El evento Paint

Comenzaremos viendo un evento muy importante, el *Paint*, y sobre él se dará una introducción al manejo de eventos en .NET

Evento Paint

- Informa a un programa cuando parte del área cliente de la ventana se vuelve inválida y debe volver a dibujarse
- ¿Cuándo se recibe un evento *Paint*?
  - Cuando la ventana acaba de crearse (toda el área cliente es inválida)
  - Cuando una ventana superpuesta a otra se mueve, descubriéndola
  - Cuando se restaura un programa minimizado

Delegado PaintEventHandler

- *PaintEventHandler*

```
public delegate void PaintEventHandler(
      object sender, PaintEventArgs args);
```

- Es un delegado definido en System.Windows.Forms
- Para manejar el evento *Paint* debemos definir un método en nuestra clase con los mismos argumentos y tipo de retorno que el delegado *PaintEventHandler*

Manejando el evento Paint

```
void MyPaintHandler(object sender,
                    PaintEventArgs args)
{
    ...
}
```

- ¿Cómo se añade este manejador del evento *Paint* a nuestra clase *Form*?

```
form.Paint += new PaintEventHandler(MyPaintHandler);
```

- La sintaxis general es

```
objeto.evento += new delegado(método);
```

- Y para quitarlo

```
objeto.evento -= new delegado(método);
```

Manejo del evento Paint de una ventana

- Ejemplo: PaintEvent.cs
- Programa que crea una ventana e implementa un manejador para el evento *Paint*
  - Cada vez que se produzca dicho evento pintará la ventana de color chocolate y mostrará un mensaje por consola

Argumentos del manejador de Paint

- object sender
  - Se refiere al objeto que origina el evento
    - En el ejemplo anterior, el objeto *form*
- PaintEventArgs e
  - Clase definida en System.Windows.Forms

Propiedades de *PaintEventArgs*

Tipo	Propiedad	Accesibilidad	Descripción
Graphics	*Graphics*	get	Salida gráfica
Rectangle	*ClipRectangle*	get	Rectángulo inválido

Clase Graphics

- Definida en *System.Drawing*
- Una de las más importantes de la biblioteca *Windows Forms*
- Es la clase que hay que usar para dibujar texto y gráficos
- Normalmente la primera línea del manejador del evento *Paint* será algo como

```
Graphics graphics = e.Graphics;
```


Qué no debemos poner en el manejador de Paint

- El método puede ser llamado muy frecuentemente
  - Debe poder redibujar el área cliente rápidamente y sin verse interrumpido
- ¡No llamar a MessageBox.Show!
  - Podría cubrir parte del área cliente, lo que resultaría en otro evento *Paint*… y así sucesivamente
- Por lo mismo, no incluir llamadas a Console.Read o Console.ReadLine
  - Console.Write o Console.WriteLine sí son seguras

Otras formas de dibujar

- No siempre tiene por qué estar todo el código de dibujado en el manejador de *Paint*
- Podemos obtener un objeto *Graphics* mediante el método CreateGraphics de la clase *Control*
- También podemos necesitar generar un evento *Paint* manualmente
  - Llamaremos al método Invalidate

Un manejador, varias ventanas

- Ejemplo: PaintEventTwoForms.cs
- Este programa crea dos ventanas que usan ambas el mismo manejador del evento *Paint*
  - El manejador mostrará un mensaje diferente según haya sido llamado de una u otra
- ¿Cómo saber cuál es la que lanzó el evento?
  - Por medio del primer parámetro de *PaintEventHandler*
    - `object sender`

Varios manejadores para el mismo evento

- TwoPaintEventHandlers.cs
- Este programa asocia varios manejadores con el mismo evento
- Cuando hay más de un manejador, se llama a todos secuencialmente
  - En el orden en que se añadieron

Eventos y métodos "On…"

- Hemos visto el método general de añadir manejadores de eventos:
  - Primero, definiendo un método con los mismos parámetros y tipo de retorno que el delegado del evento

```
void MyPaintEventHandler(Object sender,
                         PaintEventArgs e)
{
    // Código de dibujado
}
```

  - A continuación, creábamos un objeto delegado

```
form.Paint += new
PaintEventHandler(MyPaintEventHandler);
```

Eventos y métodos "On…"

- Pero en las clases que derivan de *Control* tenemos otra posibilidad
- Podemos redefinir el método protegido *OnPaint*

```
protected override void OnPaint(
        PaintEventArgs e)
{
    // Código de dibujado
}
```

- Cada uno de los eventos definidos en *Windows Forms* tiene su correspondiente método protegido
  - *"On"* + el nombre del evento

Eventos y métodos "On…"

- Estos métodos se declaran como protegidos
- Esto implica que sólo podemos acceder a ellos desde una clase derivada
- La documentación de Windows Forms nos recomienda que siempre que redefinamos un método "On…" llamemos al mismo método de la clase base

```
base.OnPaint(e)
```

- Ejemplo: InheritedFormOnPaint.cs

Modelo de eventos de .NET

Una vez que hemos visto cómo utilizar eventos y delegados predefinidos, examinaremos más en profundidad el modelo de eventos de la plataforma y cómo podemos definir nuestros propios eventos y delegados

Delegados

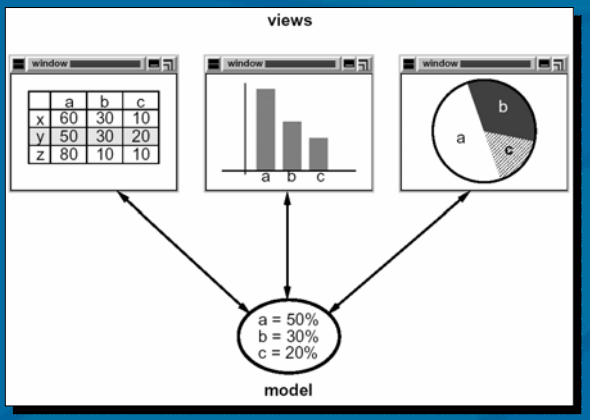
- Los eventos en .NET se basan en el modelo de delegados
  - Es una implementación del patrón de diseño Observer (Publish-Subscribe)
    - Libro "Design Patterns. Elements of Reusable Object-Oriented Software", de Erich Gamma et al.
      - Traducción española de César Fernández Acebal y Juan Manuel Cueva Lovelle:
        - Patrones de Diseño, Editorial Pearson Educación, 2003

Patrón Observer

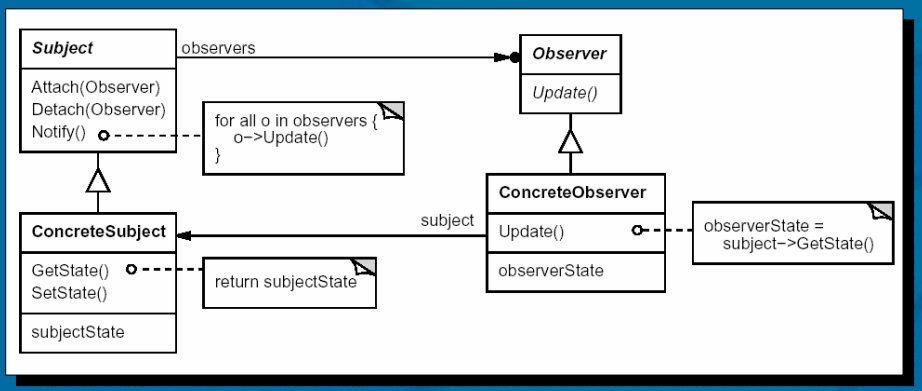
Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependen de él.

MVC

- El patrón de diseño *Observer* es la base del patrón arquitectónico Modelo-Vista-Controlador (MVC)



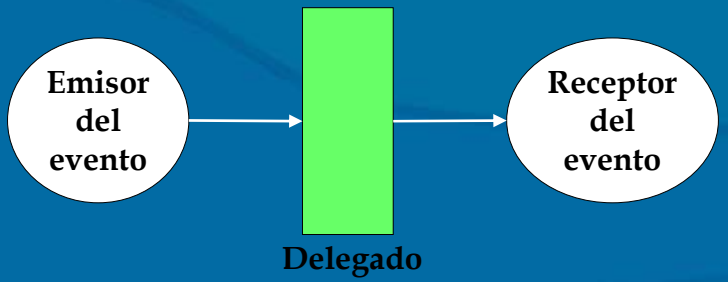
Estructura del patrón Observer



Eventos y delegados

- Un evento es un mensaje enviado por un objeto para indicar que ha ocurrido algo
  - Puede ser causado por la interacción del usuario o por la propia lógica del programa
- En el modelo de eventos, el objeto que lanza el evento no sabe qué objeto lo recibirá (manejará)
- Hace falta un intermediario
  - En .NET, este papel lo desempeñan los delegados

Eventos y delegados



- Un delegado es una *clase* que guarda una referencia a un método
  - Similar a un puntero a función, o a una función callback, pero seguro con respecto al tipo (orientado a objetos)

Ejemplo de delegado

- Una clase *Clock* lanza un evento cada vez que cambia la hora (cada segundo)

```
public delegate void TimeChangedEventHandler(
        object sender, TimeChangedEventArgs e);
```

- Convenios de la plataforma .NET
  - <*nombre evento*>EventHandler
  - Toma dos parámetros:
    - El objeto que da lugar al evento
      - object sender
    - Los datos del evento
      - <*nombre evento*>EventArgs e

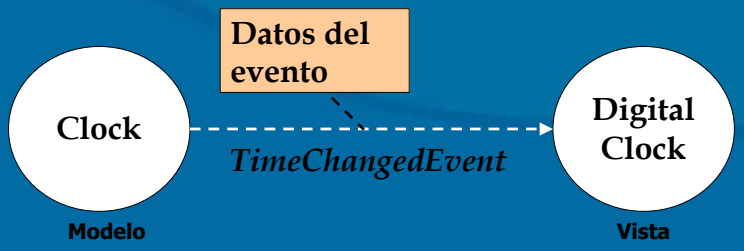
Datos del evento

- El segundo parámetro debe ser una clase hija de System.EventArgs

```
public class TimeChangedEventArgs: EventArgs
{
    ...  // Datos asociados al evento
}
```

- O la propia clase *EventArgs*, en caso de que el evento no tenga ningún dato asociado
  - En ese caso, al llamar al delegado le pasaríamos null

Conectando emisor y receptor



- ¿Cómo le dice la vista del reloj (el reloj digital) a su modelo que está interesada en un evento suyo?
  - Debe proporcionar un manejador del evento
  - Y registrar dicho manejador en la clase origen de éste

Declarar el delegado

- Lo primero es declarar el delegado asociado al evento
- Esto se hace fuera de ambas clases
  - Ni en la emisora ni en la receptora
  - Suele ponerse justo antes de la definición de la clase emisora

```
public delegate void TimeChangedEventHandler(
        object sender, TimeChangedEventArgs e);

class Clock { ... }
```


Declarando el evento

Clase emisora

- Previamente, fuera de la clase, hubo que declarar un delegado para ese evento

  A continuación, dentro de la clase emisora, se declara el evento en sí:

```
class Clock {
  ...
  public event TimeChangedEventHandler TimeChanged;
  ...
}
```

- Una vez declarado, la clase puede tratar al evento como otro campo cualquiera
  - (del correspondiente tipo delegado)
- Dicho campo podría ser *nulo* si no hay ningún cliente suscrito al evento
  - En decir, si no se asoció ningún delegado a dicho evento

Lanzando el evento

Clase emisora

- Y debe también implementar un método llamado On… que lance el evento

```
protected virtual void OnTimeChanged(
        TimeChangedEventArgs e)
{
    if (TimeChanged != null)
        TimeChanged(this, e);
}
```

- Sólo se puede lanzar el evento desde la propia clase donde fue declarado

Implementando un manejador

Clase receptora

- Se implementa en la clase interesada en recibir y tratar el evento
- Hay que implementar un método con la misma signatura que el delegado

```
public void TimeChanged(object sender,
                        TimeChangedEventArgs e)
{
    ...  // Mostrar la nueva hora
}
```

- Y crear una nueva instancia del delegado pasándole dicho método como parámetro

```
new TimeChangedEventHandler(TimeChanged)
```

Registrando el manejador

Clase receptora

- ¿Cómo se registra el manejador con la clase que da lugar al evento?
- El compilador habrá generado en aquella clase un miembro con el evento
  - En nuestro caso, *TimeChanged*
- La forma de enlazar el manejador con dicho evento es

```
theClock.TimeChanged +=
    new TimeChangedEventHandler(TimeChanged);
```


Ejercicio: Reloj

- Crear una clase Clock que implemente un reloj
  - Para ello, haremos uso de la clase Timer
  - (Será el modelo, en el patrón MVC)
- Cada segundo, deberá lanzar un evento indicando que la hora ha cambiado
  - TimeChangedEvent
- Crearemos una clase ConsoleClock que se suscriba a dicho evento y muestre la nueva hora cada vez que ésta cambia
  - Será la vista, en el patrón MVC

Ampliación del ejemplo anterior

- Veamos cómo al mismo modelo podemos asociarle varias vistas diferentes
- Además del reloj de consola, crearemos un reloj digital y otro analógico
- Las tres vistas se suscribirán al mismo evento, y se actualizarán en consonancia

Otros usos de los delegados

Los delegados son el mecanismo de implementación de eventos en Microsoft .NET Framework, pero también tienen otros usos.

A continuación profundizaremos en los tipos delegados y veremos otros usos de ellos diferentes de la programación de eventos.

Definición de delegado

- Un delegado es un estructura de datos que referencia a un método estático o a un método de instancia junto con una referencia a la instancia sobre la que se ejecutará
- Un delegado se parece a un puntero a función en C o C++
  - Permite encapsular una referencia a un método dentro del objeto delegado
  - Podemos pasar el objeto delegado a otro código que llamará al método referenciado por éste
    - Sin saber en tiempo de compilación qué método será invocado
  - A diferencia de aquéllos, los delegados son seguros con respecto al tipo (y completamente orientados a objetos)

Los delegados son tipos

- Una declaración de delegado define un tipo que encapsula un método con unos parámetros y un tipo de retorno determinados
- Son, por tanto, como “funciones anónimas”
- Podemos pensar en un delegado como una forma de nominar la signatura de un método
  - ```
    delegate long OperacionIntInt(int uno,
                                  int otro);
    ```
  - ```
    delegate string ComoCadena();
    ```

Aclaración terminológica

- Con 'delegado' designamos tanto al tipo del objeto (esto es, a la signatura del método) como a las instancias concretas de ese tipo
  - A diferencia, por ejemplo, de 'clase' y 'objeto'

```
OperacionIntInt sumarEnteros = new
    OperacionIntInt(Sumar);

ComoCadena comoCadena = new
    ComoCadena(Describir);
```

Ejemplo: Librería

- Supongamos una base de datos de libros
  - Simulada con la clase BookDatabase
    - Contendrá una colección de libros
- Queremos recorrer dicha colección para hacer dos operaciones distintas
  - Calcular el precio total de todos sus libros
  - Imprimir el título de cada uno de ellos

¿Cómo se haría en Java?

- Es decir, ¿cómo se realiza una operación sobre cada elemento de una colección?
- La forma usual de hacer un recorrido es:

```
Iterator iterator = list.iterator();
while (iterator.hasNext())
{
    ListObject object =
        (ListObject) iterator.next();
    // ... hacer la operación que corresponda
}
```

Problemas

- Ese enfoque nos obliga a realizar un recorrido por cada operación a realizar
- Los delegados permiten que la lógica de recorrido sea independiente de la operación a realizar sobre cada elemento de la colección