TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE TIJUANA

DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN

INGENIERÍA EN SISTEMAS COMPUTACIONALES

TOPICOS AVANZADOS DE PROGRAMACIÓN

DELEGADOS VS INTERFACES

DR. DANIEL EDUARDO HERNÁNDES MORALES

MARCO ANTONIO VELAZQUEZ FIGUEROA

FRANCISCO JAVIER HERNANDEZ ORNELAS

Introducción

El programador no puede estar pendiente todo el tiempo de las “acciones” que realizan los objetos, a veces suceden cosas mientras no estamos viendo, y los objetos deben de poder responder a estas “acciones” o “eventos” de cierta manera, la idea principal es que un objeto publica un evento, mientras otros objetos escuchan dichas publicaciones y toman las acciones requeridas para que el sistema siga funcionando.

En base a este concepto, surge un paradigma de programación conocido como programación orientada a eventos.

En la programación orientada a objetos las clases cumplen dos roles principales:

- Editor (broadcaster), es la clase que determina cuando sucede algo y es necesario informar al resto del sistema. Esto se logra a través de intermediarios: delegados o interfaces.
- Suscriptores (subscribers), son los objetos afectados por las invocaciones del emisor. Un suscriptor puede empezar a escuchar o dejar de escuchar al emisor al utilizar los operadores += y -= en el delegado del emisor. Un suscriptor no conoce ni interfiere con el funcionamiento de otros suscriptores.

En esta practica se vio lo que eran los delegados e interfaces que toman el rol de “Broadcasters” en la programación orientada a objetos, con teoría vista en clase, y el ejercicio puesto por el profesor.

Marco teórico

Delegados:

Un delegado deriva de la clase *System.Delegate* y puede referenciar métodos que se correspondan con su signatura. Para trabajar con delegados son necesarios tres elementos básicos: la declaración del delegado, los métodos que podrán ser invocados por el delegado y la declaración de la instancia del tipo delegado que referencia uno de estos métodos.

Un delegado es una clase capaz de referenciar un método, tanto estático como no, que coincida con su signatura. La declaración de un delegado tiene el siguiente formato:

[atributos] [modificadores] ***delegate*** *tipoResult NomDelegado([tipoParam$_n$ param$_1$, … tipoParam$_n$]);*

Al declarar un delegado se describe el tipo de retorno, el número y tipo de los argumentos de los métodos que se van a poder referenciar por dicho delegado.

Ejemplo:

Public ***delegate*** *int operaciones(int x, int y)*

Es posible efectuar varias declaraciones de tipos de delegado con los mismos parametros y el mismo tipo de retorno. Estos tipos delegados serán considerados diferentes.

Ademas de declarar el tipo, para trabajar con delegados es necesario implementar métodos que coincidan exactamente con la declaración de tipo de delegado y crear instancias del delgado que referencien dichos métodos. Para instanciar un método con el que se desea establecer una asociación sin especificar los argumentos del mismo. Por último, la invocación se efectuará escribiendo el nombre de la instancia de tipo delegado seguida por los argumentos correspondientes encerrados entre paréntesis.

Ejemplo:

Operaciones op1 = ***new*** *operaciones(s.Suma)*

Para una clase es posible trabajar con un objeto delegado sin tener que conocer en tiempo de compilación el método que el delegado referenciará. Los delegados pueden considerarse semejantes a las interfaces si se considera que, análogamente a ellas, solo se especifican la signatura de un método y necesitan la posterior codificación de métodos compatibles con dicha especificación. Los delegados son la base para los eventos.

Interfaces:

Una interfaz es un sistema o dispositivo que utiliza entidades no relacionadas que interactúan. Ejemplos de interface son un mando a distancia para televisión, que es una interfaz entre el espectador y un aparato de televisión, un navegador de internet, que es una interfaz entre el internauta y el internet.

Una interfaz se considera como una clase especial, que se puede compilar en un archivo independiente tal como una clase ordinaria, donde se definen un conjunto de miembros sin especificar sus implementaciones.

No se pueden crear instancias de una interfaz. En la mayoría de los casos sin embargo se puede utilizar una interfaz de modo similar a como se utiliza una clase abstracta. La interfaz sería como una clase abstracta pura que solo almacenara miembros abstractos y en la que hay que tener en cuenta que:

- Todos los miembros son implícitamente públicos(no hay que declararlos públicos).
- Todos lo métodos son implícitamente abstractos(se especifica el descriptor del método y no hay que declararlos *abstract*).

Una definición de interfaz consta de dos componentes: la *cabecera* y el *cuerpo* de la interfaz. La cabecera de la interfaz declara diversos atributos acerca de la interfaz, tal como su nombre y si se extiende a otra interfaz. El cuerpo de la interfaz contiene las declaraciones de los miembros.

*[atributos] [modificadores] i**nterface** NombreInterfaz [: ListaDeInterfacesBase]*

{

//declaraciones de métodos, propiedades, indexadores y/o eventos

]

Desarrollo:

Delegados

Línea 56:

Se declara el delegado “Transformer” con las características que satisfacen las necesidades de nuestro codigo.

```
56          public delegate int Transformer(int x);
```

Línea 60:

Se declara el método “Doubler”, que se encargara de regresarnos el doble del número que se le mande.

```
60          static int Doubler(int x)
61          {
62
63              return x * 2;
64          }
```

Línea 86:

Se declara el método “Factorial”, que se encargara de calcular el factorial del número que se le mande.

```
86          static int Factorial(int x)
87          {
88              int r = 0;
89
90              if (x == 0)
91              {
92
93                  r = 1;
94
95              }
96              else
97              {
98
99                  r = x * Factorial(x - 1);
100
101             }
102
103             return r;
104         }
```

Línea 68:

Se declara el método “IsPair”, que se encargara de verificar si el número que se le ha mandado es par o impar, y regresara el número 0 si es impar, y el número 1 si es par.

```
68          static int IsPair(int x)
69          {
70              int flag = 0;
71
72              if (x % 2 == 0)
73              {
74                  flag = 0;
75              }
76              else
77              {
78                  flag = 1;
79              }
80
81              return flag;
82          }
```

Línea 108:
Se declara una clase llamada “Util”, donde se declara un método llamado “TransformAll”. El método “TransformAll” se encargara de invocar el delegado en todos los datos de un arreglo, y así se podrá utilizar el método en estos mismos datos.

```
108         class Util
109         {
110
111             //Método donde se invoca el delegado para todos los datos de un array
112
113             public static void TransformAll(int[] vals, Transformer t)
114             {
115                 for (int i = 0; i < vals.Length; i++)
116                     vals[i] = t.Invoke(vals[i]);
117             }
118         }
```

Línea 10:

Aquí se encuentra el método “Main”.

En la línea 12 se instancia un objeto de tipo delegado mandandole como argumento el método “Doubler”

En la línea 13 se instancia un objeto de tipo delegado mandandole como argumento el método “IsPair”

En la línea 14 se instancia un objeto de tipo delegado mandandole como argumento el método “Factorial”

En la línea 20 se declara el arreglo a utilizar para las pruebas con los delegados.

En la línea 22 se invoca el delegado mediante el objeto “d” con el método “TransformAll” de la clase “Util”, y así se calculara el doble de todos los datos del arreglo.

En la línea 32 se invoca el delegado mediante el objeto “i” con el método “TransformAll” de la clase “Útil”, y así comprobara los datos pares e impares de todos los datos del arreglo.

En la línea 42 se invoca el delegado mediante el objeto “f” con el método “TransformAll” de la clase “Util”, y así se calculara el factorial de todos los datos del arreglo.

```
10          Console.WriteLine("Actividad delegados \n");
11
12          Transformer d = Doubler;
13          Transformer i = IsPair;
14          Transformer f = Factorial;
15
16
17
18          Console.WriteLine("Números al doble ~ \n");
19
20          int[] arrayD = { 1, 2, 3, 4, 5 };
21
22          Util.TransformAll(arrayD, d);
23          foreach (int a in arrayD)
24          {
25              Console.WriteLine(a);
26          }
27
28          Console.WriteLine("\nEl número ingresado es par(0) | es impar(1) ~ \n");
29
30          int[] arrayI = { 1, 2, 3, 4, 5 };
31
32          Util.TransformAll(arrayI, i);
33          foreach (int a in arrayI)
34          {
35              Console.WriteLine(a);
36          }
37
38          Console.WriteLine("\nFactorial de los números ~ \n");
39
40          int[] arrayF = { 1, 2, 3, 4, 5 };
41
42          Util.TransformAll(arrayF, f);
43          foreach (int a in arrayF)
44          {
```

Interfaces

Línea 10:

Se declara una interface con el nombre de “ITransformer” y dentro se declara un método con el nombre de “Transform”.

```
10      public interface ITransformer
11      {
12          int Transform(int x);
13      }
```

Línea 15:

Se declara la clase “Doubler” heredando de la interface “ITransformer”, y dentro se utiliza el método “Transform” heredado desde la interface, pero aquí nos devuelve el doble del valor ingresado.

```
15          class Doubler : ITransformer
16          {
17              public int Transform(int x) { return x * x; }
18          }
```

Línea 27:

Se declara la clase “Factorial” heredando de la interface “ITransformer”, y dentro se utiliza el método “Transform” heredado desde la interface, pero aquí nos devuelve el factorial del valor ingresado.

```
27          class Factorial : ITransformer
28          {
29              public int Transform(int x)
30              {
31                  for (int i = x-1; i > 0; i--)
32                  {
33                      x *= i;
34                  }
35                  return x;
36              }
37          }
```

Línea 19:

Se declara la clase “IsPair” heredando de la interface “ITransformer”, y dentro se utiliza el método “Transform” heredado desde la interface, pero aquí nos devuelve el número 0 si el número ingresado es par, y devuelve 1 si no lo es.

```
19          class IsPair : ITransformer
20          {
21              public int Transform(int x)
22              {
23                  if (x % 2 == 0) {return 0;}
24                  else return 1;
25              }
26          }
```

Línea 39:

Se declara una clase llamada “Util”, donde se declara un método llamado “TransformAll”. El método “TransformAll” se encargara de invocar el delegado en

todos los datos de un arreglo, y así se podrá utilizar el método en estos mismos datos.

```
39          public class Util
40          {
41              public static void TransformAll(int[] values, ITransformer t)
42              {
43                  for (int i = 0; i < values.Length; i++)
44                  {
45                      values[i] = t.Transform(values[i]);
46                      Console.WriteLine(values[i]);
47                  }
48                  Console.WriteLine();
49              }
50          }
```

Línea 51 (MAIN):

Después se crean los objetos a partir de las 3 clases y se llama a ejecutar el método TransformAll, que ejecutará los métodos Doubler, IsPair y Factorial usando un arreglo de números.

```
51          static void Main(string[] args)
52          {
53
54              Doubler S = new Doubler();
55              IsPair I = new IsPair();
56              Factorial F = new Factorial();
57
58              int[] valuesS = { 1, 2, 3, 4, 5 };
59              int[] valuesI = { 1, 2, 3, 4, 5 };
60              int[] valuesF = { 1, 2, 3, 4, 5 };
61
62              Util.TransformAll(valuesS, S);
63
64              Util.TransformAll(valuesI, I);
65
66              Util.TransformAll(valuesF, F);
67
68              Console.ReadKey(false);
69          }
```

Conclusiones

Conclusión de Marco:

En esta practica me pude dar cuenta de las similitudes entre usar un delegado y una interface al momento de escribir código, lo que me ayudó un poco más a entender cómo utilizar los delegados y no perderme tanto al entender estos mismos. Una utilidad muy grande que le da ventaja al uso de delegados en los programas, es poder llamar varios métodos desde un mismo delegado, esto me enredo un poco al principio, pero una vez que le entendí, miré muchas posibles utilidades para estos mismos. Al momento de escribir el código seguía sin entender esta funcionalidad de poder llamar varios métodos con un solo delegado, por eso no lo hice así aquí.

Creo que el uso de los delegados te abre muchas puertas al momento de escribir código, así como ventajas frente a las interfaces.

Conclusión Francisco:

Me parece especialmente útil el uso de interfaces al momento de realizar alguna aplicación que tenga distintos usos o herramientas derivadas de una misma base, o que regresen un valor similar.
Además de que es muy simple su estructura y fácil de entender.

Bibliografía

Luis Joyanes Aguilar, Matilde Fernández Azuela. *C# Manual de programación.* Mc Graw-Hill, 2002.