«Talento Tech»

Desarrollo de Videojuegos

Unity 3D

Clase 06

Buenos Aires Ciudad

Clase N° 6 | Delegados y Eventos II

Temario:

- Introducción al Observer Pattern
- Generics
- EventManager

Objetivos de la clase

Introducir el Observer Pattern en Unity.

- Comprender el propósito y las ventajas del Observer Pattern en diseño de software.
- Explorar cómo se implementa en Unity utilizando Delegates y Events.

Entender el uso de Generics en C#.

- Explicar el concepto de Generics y su utilidad en la creación de clases y métodos reutilizables.

Implementar un EventManager para gestionar eventos.

- Diseñar un sistema centralizado para manejar eventos en un proyecto de Unity.
- Usar el EventManager para comunicar cambios entre distintos scripts, optimizando la arquitectura del código.

Observer Pattern

El **Observer Pattern** (o patrón observador) es un patrón de diseño que establece una relación entre objetos para que cuando uno de ellos cambie su estado, los otros sean notificados automáticamente. Es muy utilizado en programación para manejar eventos o actualizaciones en tiempo real, y es la base conceptual detrás de los **eventos** en lenguajes como C#.

Definición formal:

El patrón observador define una relación **uno a muchos** entre objetos, donde:

1. **Uno**: El objeto observado o sujeto (**Subject**).
    - Es quien tiene la información o el estado que otros quieren monitorear.
2. **Muchos**: Los objetos observadores (**Observers**).
    - Son quienes se registran (o suscriben) para recibir notificaciones sobre cambios en el sujeto.

Cuando el **sujeto** cambia su estado, automáticamente notifica a todos sus observadores.

Pensá en un canal de YouTube:

1. El canal es el **sujeto** (Subject).
2. Los usuarios suscritos son los **observadores** (Observers).

Cuando el canal sube un nuevo video, notifica automáticamente a todos los suscriptores.

Estructura básica:

1. **Sujeto**: La entidad que mantiene una lista de observadores y los notifica cuando ocurre un cambio.

2. **Observadores**: Los objetos que "se suscriben" al sujeto para recibir notificaciones.

3. **Desacoplamiento**: El sujeto no necesita conocer los detalles de los observadores, lo que facilita la escalabilidad y el mantenimiento.

Ejemplo Básico

Como venimos acostumbrados a la lógica básica del puntaje, crearemos un sujeto y un observador basados en esa idea:

Clase Sujeto: ScoreManager

```
public class ScoreManager{
    // Definimos un delegado explícito
    public delegate void ScoreChangedDelegate(int newScore);
    public event ScoreChangedDelegate OnScoreChanged;
    private int score;

    public void AddScore(int points) {
        score += points;
        // Llamamos al evento cuando se actualiza la puntuación
        if (OnScoreChanged != null)
            OnScoreChanged.Invoke(score);
    }
}
```

Clase Observador: ScoreUI

```
using TMPro;

public class ScoreUI : MonoBehaviour{
    [SerializeField] private TextMeshProUGUI scoreText;
 void Start(){
        OnEnable();
    }

  private void OnEnable(){
   //Nos suscribimos al evento para escuchar cambios en la puntuación
        FindObjectOfType<ScoreManager>().OnScoreChanged +=
UpdateScoreText;
    }

    private void OnDisable(){
//Nos desuscribimos cuando el objeto se desactiva para evitar problemas
        FindObjectOfType<ScoreManager>().OnScoreChanged -=
UpdateScoreText;
    }

    private void UpdateScoreText(int newScore){
        scoreText.text = "Score: " + newScore;
    }
}
```

Explicación del Ejemplo

Sujeto:

Definimos un delegado explícito:

```
public delegate void ScoreChangedDelegate(int newScore);
```

En este caso, el delegado dice: "Cualquier método que use este delegado debe aceptar un parámetro de tipo `int` y no devolver nada".

Creamos un evento basado en ese delegado:

```
public event ScoreChangedDelegate OnScoreChanged;
```

Este evento permitirá que otros scripts se suscriban y reciban notificaciones cuando ocurra un cambio en la puntuación.

En la misma clase `ScoreManager`, añadimos un método para cambiar la puntuación y notificar a los suscriptores:

Declaramos una variable para almacenar la puntuación:

```
private int score;
```

Creamos un método para agregar puntos:

```
public void AddScore(int points)

{

    score += points; // Actualizamos la puntuación

    if (OnScoreChanged != null) // Verificamos si alguien está suscrito al evento

          OnScoreChanged.Invoke(score);  // Disparamos el evento, pasando la nueva puntuación }
```

`OnScoreChanged.Invoke(score)` notifica a todos los scripts suscritos al evento, enviándoles la nueva puntuación como parámetro.

Observador

En la clase `ScoreUI`, configuramos cómo reaccionará la interfaz de usuario al cambio de puntuación:

Creamos un método para actualizar el texto de la puntuación:

```
private void UpdateScoreText(int newScore)

{

    scoreText.text = "Score: " + newScore;

}
```

Este método será llamado cuando el evento se dispare. Recibe el nuevo valor de la puntuación como parámetro y actualiza el texto en pantalla.

Nos suscribimos al evento en `OnEnable`:

```
private void OnEnable()

{

    FindObjectOfType<ScoreManager>().OnScoreChanged += UpdateScoreText;

}
```

Esto significa que cuando el GameObject esté activo, el método `UpdateScoreText` será llamado cada vez que se dispare el evento `OnScoreChanged`.

El método **FindObjectOfType** en Unity se utiliza para buscar una instancia de un objeto en la escena que sea del tipo especificado. Este método recorre todos los objetos activos de la escena hasta encontrar uno que coincida con el tipo solicitado. Es útil cuando necesitas acceder a un componente o script en particular, pero no tienes una referencia directa a él.

Al método **OnEnabe lo podremos llamar en el Start()** o algún otro método que deseemos.

```
 void Start()

    {

        OnEnable();

    }
```

Nos desuscribimos en `OnDisable`:

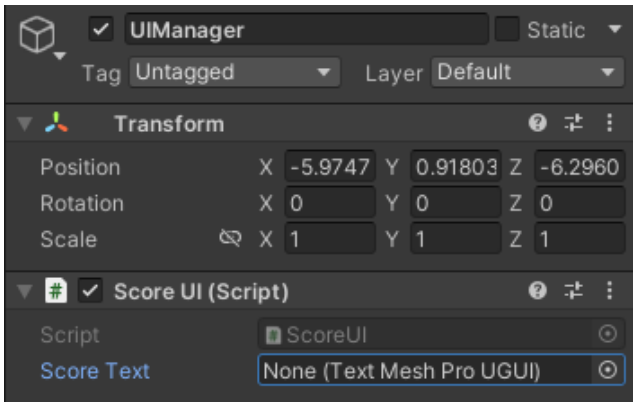
```
  private void OnDisable() {
//Nos desuscribimos cuando el objeto se desactiva para evitar problemas
 FindObjectOfType<ScoreManager>().OnScoreChanged -= UpdateScoreText; }
```

Esto evita problemas como llamadas a métodos de objetos que ya no existen. Es **importante** hacerlo cuando el objeto que posee el script es **destruido o deja de utilizarse**.

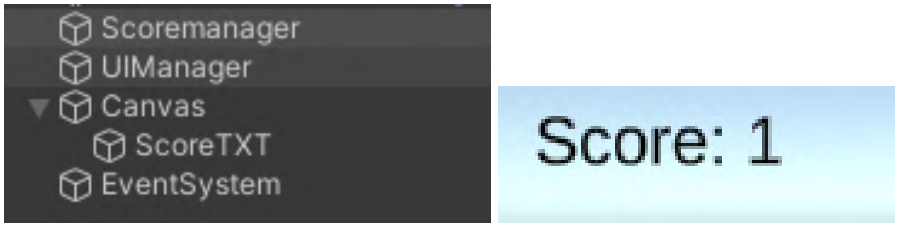
Configuración en Unity:

1. **Crear un objeto para el `ScoreManager`:**
    - Añade el script `ScoreManager` a un GameObject vacío, por ejemplo, llamado "ScoreManager".

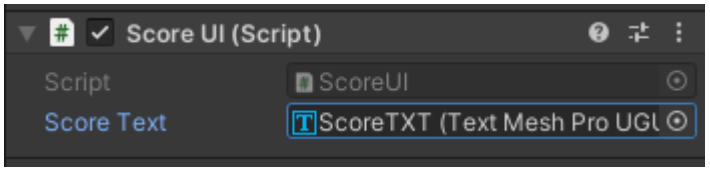
2. **Añadir el script `ScoreUI`:**
    - Añade el script `ScoreUI` a un GameObject que controle la UI.



3. **Crear la interfaz de usuario para mostrar la puntuación:**
    - Añade un texto de UI (`TextMeshPro`) a tu Canvas.



    - Arrastra este objeto de texto al campo `scoreText` del script `ScoreUI`.



Con esto tendremos un Observer Pattern aplicado de forma sencilla a un concepto familiar como el Score.

Generics

En C# y Unity, los **Generics Types** son una característica del lenguaje que permite definir clases, interfaces, métodos y estructuras que pueden funcionar con cualquier tipo de dato. La principal ventaja de los genéricos es que permiten la reutilización del código sin sacrificar la seguridad de tipos en tiempo de compilación. Esto ayuda a crear clases o métodos más flexibles y reutilizables, manteniendo la integridad de los tipos de datos.

¿Cómo funcionan los Generics en C#?
Un Generic se define mediante el uso de un tipo de marcador (comúnmente llamado parámetro de tipo) en lugar de un tipo específico. Este parámetro de tipo se reemplaza por un tipo concreto cuando se crea una instancia de la clase o se llama al método. Los parámetros de tipo se denotan con un tipo genérico en la definición.

Ejemplo de un tipo genérico en C#:

```
public class Caja<T>
{
    private T contenido;

    public void Guardar(T item) {
        contenido = item;
    }

    public T Obtener()
    {
        return contenido;
    }
}
```

En este ejemplo, T es un parámetro de tipo genérico. Puedes usar esta clase para almacenar cualquier tipo de objeto, por ejemplo:

```
    private void Start(){
        // Crear una caja de enteros y guardar un valor
        Caja<int> cajaDeEnteros = new Caja<int>();
        cajaDeEnteros.Guardar(10);
        int numero = cajaDeEnteros.Obtener();
        Debug.Log("Número guardado: " + numero);

        // Crear una caja de strings y guardar un valor
        Caja<string> cajaDeStrings = new Caja<string>();
        cajaDeStrings.Guardar("Hola Mundo");
        string mensaje = cajaDeStrings.Obtener();
        Debug.Log("Mensaje guardado: " + mensaje);
    }
```

Ejemplo en Unity:

Imagina que estás trabajando con varios componentes en un GameObject, como `Rigidbody`, `MeshRenderer`, etc., y necesitas un método genérico para acceder o agregar componentes según sea necesario. Aquí es donde los **Generics** resultan útiles.

Clase Genérica para Manejar Componentes

```
using UnityEngine;
public static class ComponenteHelper{
    // Método genérico para obtener o agregar un componente
    public static T ObtenerOAgregarComponente<T>(GameObject gameObject)
where T : Component{
        T componente = gameObject.GetComponent<T>();
        if (componente == null) {
            componente = gameObject.AddComponent<T>(); }
       return componente; }}
```

¿Cómo usar esta clase en Unity?

Pueden usar este método genérico para asegurarte de que un componente específico esté presente en un GameObject, y si no está, se agrega automáticamente.

```
using UnityEngine;
public class EjemploUsoComponenteHelper : MonoBehaviour{
    void Start() {
        // Obtiene o agrega un Rigidbody al GameObject actual
        Rigidbody rb =
ComponenteHelper.ObtenerOAgregarComponente<Rigidbody>(gameObject);
        rb.mass = 5f; // Configura alguna propiedad del Rigidbody
        // Obtiene o agrega un MeshRenderer al GameObject actual
     MeshRenderer mr =
ComponenteHelper.ObtenerOAgregarComponente<MeshRenderer>(gameObject);
        mr.material.color = Color.red; // Cambia el color del material
    }}
```

Explicación

- El método genérico `ObtenerOAgregarComponente<T>` utiliza un parámetro de tipo genérico T, que debe ser un componente (`where T : Component`).
- Si el GameObject ya tiene un componente del tipo solicitado, simplemente lo devuelve.
- Si el componente no existe, se agrega automáticamente al GameObject usando `AddComponent<T>()`.

EventManager.

Habiendo aprendido este concepto más lo de la clase anterior, crearemos un EventManager con un **dictionary** que nos permitirá subscribirnos y de-suscribirnos a los eventos que deseemos.
La idea es permitir que los objetos puedan "suscribirse" a un evento identificado por un nombre para ser notificados cuando ese evento ocurra.

Como siempre, primero presentaremos el código y lo iremos desglosando parte por parte:

```
using System.Collections.Generic;
using UnityEngine;
using System;

public class EventManager3 : MonoBehaviour
{
    // Define un delegate explícito sin parámetros
    public delegate void EventDelegate<T>(T param);

    private static Dictionary<string, Delegate> eventDictionary = new
Dictionary<string, Delegate>();

    public static void Subscribe<T>(string eventName, EventDelegate<T>
subscriber)
    {
        if (eventDictionary.TryGetValue(eventName, out Delegate
thisEvent))
        {
            eventDictionary[eventName] = (EventDelegate<T>)thisEvent +
subscriber;
        }
        else
        {
            eventDictionary.Add(eventName, subscriber);
        }
    }

    public static void Unsubscribe<T>(string eventName,
EventDelegate<T> subscriber)
    {
        if (eventDictionary.TryGetValue(eventName, out Delegate
thisEvent))
        {
            var currentDel = (EventDelegate<T>)thisEvent - subscriber;
```

```
            if (currentDel == null)
            {
                eventDictionary.Remove(eventName);
            }
            else
            {
                eventDictionary[eventName] = currentDel;
            }
        }
    }

    public static void Notify<T>(string thisEvent, T param)
    {
       // Comprobamos si 'thisEvent' es de tipo 'EventDelegate<T>'
antes de intentar invocar el delegado
      EventDelegate<T> eventDelegate = thisEvent as EventDelegate<T>;

      if (eventDelegate != null)
      {
    // Si la conversión es exitosa, invocamos el delegado pasando
'param' como argumento
         eventDelegate.Invoke(param);
      }
      else
      {
    // Si 'thisEvent' no puede ser convertido a 'EventDelegate<T>', no
hacemos nada o podemos manejarlo aquí.
         Console.WriteLine("El evento no es del tipo correcto.");
      }
     }
}
```

Desglosando el código:

Declaración del delegado

```
public delegate void EventDelegate<T>(T param);
```

Un **delegado** es un tipo que define la forma de un método. Aquí, `EventDelegate` es un delegado genérico que acepta un parámetro de tipo T y no devuelve nada (`void`).

Por ejemplo:

- Si T es un `string`, un método asociado al delegado tendría la forma `void MyMethod(string param)`.

Diccionario para manejar eventos

```
private static Dictionary<string, Delegate> eventDictionary = new
Dictionary<string, Delegate>();
```

El diccionario `eventDictionary` almacena una lista de eventos. Cada entrada tiene:

- **Clave (`string`)**: Nombre del evento.
- **Valor (`Delegate`)**: Delegado que representa una lista de suscriptores para ese evento.

Método `Subscribe`

```
public static void Subscribe<T>(string eventName, EventDelegate<T>
subscriber)
{
    if (eventDictionary.TryGetValue(eventName, out Delegate thisEvent))
    {
        eventDictionary[eventName] = (EventDelegate<T>)thisEvent +
subscriber;
    }
    else
    {
        eventDictionary.Add(eventName, subscriber);
    }
}
```

Este método permite **suscribir** un método a un evento:

- **Parámetros**:
    1. `eventName`: Nombre del evento.
    2. `subscriber`: Método que desea suscribirse.
- **Funcionamiento**:
    1. Usa `TryGetValue` para buscar el evento por su nombre:
        - Si existe, combina (+) el nuevo suscriptor con los ya existentes.
        - Si no existe, crea una nueva entrada en el diccionario.
- **Nota**: El operador + en delegados agrega un nuevo método a la lista de invocaciones del evento.

```
public static void Subscribe<T>(string eventName, EventDelegate<T>
subscriber)
```

`public static`: Este método es accesible desde cualquier parte del código sin necesidad de crear una instancia del `EventManager`.

<T>: Se utiliza un tipo genérico T, lo que significa que este método puede trabajar con eventos que acepten cualquier tipo de parámetro (como `int`, `float`, o incluso un objeto personalizado).

`eventName`: Es una cadena que identifica el evento al que se desea suscribir.

`subscriber`: Es el método o función (representado por un `delegate`) que se ejecutará cuando el evento ocurra.

```
if (eventDictionary.TryGetValue(eventName, out Delegate thisEvent))
```

`eventDictionary`: Es el diccionario que almacena todos los eventos registrados. La clave es el nombre del evento (`string`), y el valor es un `Delegate`, que puede ser un delegado explícito o genérico.

`TryGetValue`: Intenta buscar un evento con el nombre especificado (`eventName`) en el diccionario.

- Si el evento existe, lo asigna a la variable `thisEvent` y devuelve `true`.
- Si el evento no existe, `thisEvent` será `null` y la condición será `false`.

```
eventDictionary[eventName] = (EventDelegate<T>)thisEvent + subscriber;
```

Si el evento **ya existe** en el diccionario:

1. **`(EventDelegate<T>)thisEvent`**: Convierte(Castea) el `Delegate` existente (`thisEvent`) en un tipo más específico: `EventDelegate<T>`. Esto es posible porque sabemos que los eventos registrados deben coincidir con este tipo genérico.
2. **`thisEvent + subscriber`**: Agrega el nuevo método (`subscriber`) a la lista de métodos asociados a este evento. En C#, los `delegates` son combinables; puedes agregar funciones mediante el operador `+`.
3. **`eventDictionary[eventName]`**: Actualiza la entrada del diccionario para incluir al nuevo suscriptor.

```
else
{
    eventDictionary.Add(eventName, subscriber);
}
```

Si el evento **no existe** en el diccionario:

1. Se crea una nueva entrada en el diccionario usando el nombre del evento (`eventName`) como clave.
2. Se asigna el nuevo suscriptor (`subscriber`) como el valor inicial del evento.

Método `Unsubscribe`

```
public static void Unsubscribe<T>(string eventName, EventDelegate<T>
subscriber)
{
    if (eventDictionary.TryGetValue(eventName, out Delegate thisEvent))
    {
        var currentDel = (EventDelegate<T>)thisEvent - subscriber;
        if (currentDel == null)
        {
            eventDictionary.Remove(eventName);
        }
        else
        {
            eventDictionary[eventName] = currentDel;
        }
    }
}
```

Este método permite **desuscribirse** de un evento:

- **Parámetros**:
    1. `eventName`: Nombre del evento.
    2. `subscriber`: Método que desea desuscribirse.
- **Funcionamiento**:
    1. Busca el evento en el diccionario:
        - Si existe, elimina (`-`) el suscriptor de la lista de delegados.
    2. Si no quedan más suscriptores (`currentDel == null`), elimina el evento del diccionario.
    3. De lo contrario, actualiza el diccionario con la nueva lista de delegados.

Método `Notify`

```
 public static void Notify<T>(string eventName, T param){
 if (eventDictionary.TryGetValue(eventName, out Delegate thisEvent)){
//Intentar convertir el delegado a EventDelegate<T> antes de invocarlo
   if (thisEvent is EventDelegate<T> eventDelegate){
        eventDelegate.Invoke(param);
    }
    else{
 Debug.LogWarning($"El evento '{eventName}' no es del tipo esperado.");
     }
 }
 else {
Debug.LogWarning($"No hay suscriptores para el evento '{eventName}'.");
  }
}
```

`if (eventDictionary.TryGetValue(eventName, out Delegate thisEvent))`

Qué hace: Busca si el evento identificado por el nombre `eventName` existe en el diccionario `eventDictionary`.

Resultado:

- Si existe, almacena el delegado asociado al evento en la variable `thisEvent`.
- Si no existe, pasa al bloque `else`.

`if (thisEvent is EventDelegate<T> eventDelegate)`

Qué hace: Verifica si `thisEvent` puede convertirse al tipo `EventDelegate<T>`, que es el tipo genérico esperado para este evento.

Resultado:

- Si la conversión es exitosa, guarda el delegado convertido en la variable `eventDelegate` y continúa.
- Si la conversión falla (por ejemplo, si se intenta notificar con un tipo de parámetro incorrecto), pasa al bloque `else`.

`eventDelegate.Invoke(param);`

- **Qué hace**: Llama al delegado, que a su vez ejecuta todos los métodos suscritos al evento, pasando el parámetro `param` como argumento.

`Debug.LogWarning($"El evento '{eventName}' no es del tipo esperado.");`

- **Qué hace**: Genera un mensaje en la consola de Unity advirtiendo que el evento no tiene el tipo esperado. Esto ayuda a identificar problemas de tipo durante el desarrollo.

Debug.LogWarning($"No hay suscriptores para el evento '{eventName}'.");

- **Qué hace**: Si el evento no está registrado en el diccionario (es decir, no tiene suscriptores), genera un mensaje en la consola indicando que no hay nadie escuchando para ese evento.

Con este último paso, podremos crear un Event Manager que ayude a organizar todos nuestros eventos.

Más Eventos en TalentoLab:

Después de implementar los primeros eventos en Nexus, el cliente quedó satisfecho, pero pronto surgió un nuevo desafío. Con el creciente número de interacciones y eventos, el código del juego comenzó a volverse desorganizado y difícil de escalar. El cliente notó que mantener una comunicación eficiente entre los diferentes sistemas será crucial para el éxito del proyecto, especialmente si el juego continúa creciendo en complejidad.

Es aquí donde TalentoLab propone llevar la comunicación del juego al siguiente nivel con tres herramientas fundamentales: el Observer Pattern, el uso de Generics y la implementación de un EventManager. Estas herramientas permitirán organizar los eventos del juego de manera centralizada y flexible, asegurando que el código sea limpio, escalable y fácil de mantener.

Ejercicios prácticos:

La solicitud del cliente:
El cliente quiere que el equipo reorganice los eventos existentes utilizando un **EventManager**. Este nuevo sistema debe ser capaz de gestionar eventos actuales y futuros con mayor eficiencia, asegurando que todo el juego funcione como un ecosistema bien integrado.

¡Roberta quedó fascinada por tus resultados! Por lo tanto, te ofrece todo el crédito por el trabajo, dejándote crear el sistema completo.

El cliente ha solicitado una demostración sobre cómo los eventos actuales de Nexus (como la pantalla de "Game Over", el desbloqueo de zonas secretas o la obtención de Score) pueden ser reorganizados y gestionados mediante el nuevo **EventManager**.

Sugerencia: Para facilitar el trabajo, utiliza los eventos hechos en la clase anterior o proba utilizando el “Score Manager” del principio de esta clase

Materiales y recursos adicionales.

Generics:
https://learn.unity.com/tutorial/genericos#

Delegates:
https://learn.unity.com/tutorial/delegados#

Events:
https://learn.unity.com/tutorial/eventos-w#5e419557edbc2a0a62170fe6

Preguntas para reflexionar.

1. ¿Qué tan necesario puede ser un Manager?
2. ¿En que nos simplifica el uso de Events?
3. ¿Qué beneficios nos puede dar el uso de Generics?

Próximos pasos.

En la próxima clase empezaremos a trabajar el GameDesign. Estas clases fueron de mucho código y trabajos desafiantes, así que pasaremos a momentos de reflexión y creatividad sin perder el punto de vista técnico.

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad