**Taller 2 – David Barros:**

1. **Explicar por medio de un ejemplo de mínimo 5 funciones el concepto de delegado:**

RTA: En programación, un delegado es un tipo de dato que sirve para representar referencias a métodos con una firma específica. En muchos lenguajes, como en C# o Java, los delegados permiten pasar métodos como parámetro a otros métodos.

Esto sirve para facilitar la implementación de patrones de diseño como los patrones de delegado y eventos. Podríamos intentar explicarlo con algunas funciones hecha en Python como las siguientes:

◢◤◇◥◣◥◤◢◤◆◥◣◥◤◢◤◇◥◣

**// Definimos algunas funciones que realizan diferentes operaciones:**

**def suma (a, b):**

**return a + b**

**def resta (a, b):**

**return a - b**

**def multiplicación (a, b):**

**return a \* b**

**def división (a, b):**

**if b != 0:**

**return a / b**

**else:**

**return "No se puede dividir por cero"**

**def potencia (a, b):**

**return a \*\* b**

**// Creamos una función que toma un operador y dos operandos, y un delegado para la operación:**

**def operar (operación, a, b):**

**return operación(a, b)**

**// Ejemplo de uso: pasamos diferentes funciones como delegados a la función operar:**

**resultado\_suma = operar (suma, 5, 3)**

**resultado\_resta = operar (resta, 5, 3)**

**resultado multiplicación = operar (multiplicación, 5, 3)**

**resultado\_división = operar (división, 5, 3)**

**resultado\_potencia = operar (potencia, 5, 3)**

**// Mostramos los resultados:**

**print ("Suma:", resultado\_suma)**

**print ("Resta:", resultado\_resta)**

**print ("Multiplicación:", resultado\_multiplicación)**

**print ("División:", resultado\_división)**

**print ("Potencia:", resultado\_potencia)**

◢◤◇◥◣◥◤◢◤◆◥◣◥◤◢◤◇◥◣

Lo que tenemos aquí son las funciones de **“suma”, “resta”, “multiplicación”, “división”** y **potencia,** las cuales representan operaciones matemáticas diferentes. La función extra llamada **“operar”** toma una de estas funciones como delegado, junto con 2 operandos, y ejecuta en consecuencia la operación correspondiente.

1. **Consultar y crear un ejemplo donde se evidencie el uso de "event" en C# (comparar con un modelo de eventos definido en Corgi)**

RTA: Los eventos son una forma de comunicación entre objetos que permiten que uno de ellos notifique a otros objetos cuando ocurre algún evento en C#. Para lograr esto lo que se hace es emplear el uso de delegados, como en el siguiente código:

◢◤◇◥◣◥◤◢◤◆◥◣◥◤◢◤◇◥◣

**using System;**

**// Definimos un delegado que representa el tipo de método que manejará el evento:**

**public delegate void EventHandler();**

**// Clase que genera eventos:**

**public class EmisorDeEvento**

**{**

**// Declaramos un evento utilizando el delegado:**

**public event EventHandler EventoOcurrido;**

**// Método que desencadena el evento:**

**public void GenerarEvento()**

**{**

**Console.WriteLine("El evento está por ocurrir");**

**// Verificamos si hay suscriptores al evento antes de desencadenarlo:**

**if (EventoOcurrido != null)**

**{**

**// Disparamos el evento**

**EventoOcurrido();**

**}**

**}**

**}**

**// Clase que recibe y maneja los eventos:**

**public class ReceptorDeEvento**

**{**

**// Método que maneja el evento:**

**public void ManejarEvento()**

**{**

**Console.WriteLine("El evento ha sido manejado");**

**}**

**}**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**EmisorDeEvento emisor = new EmisorDeEvento();**

**ReceptorDeEvento receptor = new ReceptorDeEvento();**

**// Suscribimos el método ManejarEvento al evento EventoOcurrido:**

**emisor.EventoOcurrido += receptor.ManejarEvento;**

**// Generamos el evento:**

**emisor.GenerarEvento();**

**Console.ReadLine();**

**}**

**}**

◢◤◇◥◣◥◤◢◤◆◥◣◥◤◢◤◇◥◣

En este ejemplo, la clase **“EmisorDeEvento”** define un evento llamado **“EventoOcurrido”** utilizando el delegado **“EventHandler”.** La clase **“ReceptorDeEvento”** define un método **“ManejarEvento”,** que será llamado cuando el evento ocurra. En el método **“Main”,** creamos instancias de ambas clases, suscribimos el método **“ManejarEvento”** al evento **“EventoOcurrido”** y luego generamos el evento llamando al método **“GenerarEvento”** del objeto **“emisor”.**

Ahora, en general, en un modelo de eventos en Corgi o cualquier otro framework o motor de juego, también se pueden definir eventos y suscriptores para manejarlos de manera similar a como se hace en C#. La principal diferencia radicará en la sintaxis y la implementación específica del framework o motor de juego en cuestión.

1. **Consultar que es Singleton, sus pros y sus contras. ¿Cómo se implementan solo en C#?. ¿Cómo se implementa en Unity en Unity y donde está presente en Corgi?)**

RTA: Los Singleton son patrones de diseño creacional, los cuales garantizan que una clase tenga una única instancia y proporciona un punto de acceso global a esa instancia. Esto significa que, sin importar de cuántas veces se solicite una instancia de la clase Singleton, siempre se devolverá la misma instancia.

En cuanto a las ventajas y desventajas, tendríamos lo siguiente:

**-Pros:**

**-Control de acceso global:** Proporciona un punto de acceso global para acceder a una instancia única desde cualquier parte del código.

**-Instanciación bajo demanda:** La instancia se crea solo cuando se solicita por primera vez, lo que puede ser útil para conservar recursos en aplicaciones grandes.

**-Evita la duplicación:** Evita la creación de múltiples instancias de la misma clase, lo que puede causar problemas de inconsistencia y consumo excesivo de recursos.

**-Contras:**

**Acoplamiento:** Introduce un acoplamiento fuerte entre las clases que utilizan el Singleton y la propia clase Singleton, lo que puede hacer que el código sea menos flexible y más difícil de probar.

**-Dificultad en la concurrencia:** La implementación incorrecta del Singleton puede llevar a problemas de concurrencia en entornos multiproceso.

**-Dificultad en la depuración:** Puede ser difícil de depurar si se abusa del patrón Singleton y se utilizan ampliamente en una aplicación.

En C#, los Singletons se implementan típicamente utilizando una combinación de un constructor privado, una propiedad estática que devuelve la única instancia de la clase y un bloqueo para garantizar la concurrencia segura. Visto con un ejemplo sería algo tal que así:

◢◤◇◥◣◥◤◢◤◆◥◣◥◤◢◤◇◥◣

**public class Singleton**

**{**

**private static Singleton instance;**

**private static readonly object lockObject = new object();**

**// Constructor privado para evitar la instanciación directa:**

**private Singleton() { }**

**// Propiedad estática para acceder a la instancia única:**

**public static Singleton Instance**

**{**

**get**

**{**

**// Verifica si la instancia ya existe:**

**if (instance == null)**

**{**

**// Utiliza un bloqueo para garantizar la concurrencia segura:**

**lock (lockObject)**

**{**

**// Verifica nuevamente dentro del bloque de bloqueo:**

**if (instance == null)**

**{**

**instance = new Singleton();**

**}**

**}**

**}**

**return instance;**

**}**

**}**

**}**

◢◤◇◥◣◥◤◢◤◆◥◣◥◤◢◤◇◥◣

Para Unity lo que se puede hacer es una implementación casi que de la misma manera que en C#. Aquí estaría esa versión:

◢◤◇◥◣◥◤◢◤◆◥◣◥◤◢◤◇◥◣

**using UnityEngine;**

**public class Singleton : MonoBehaviour**

**{**

**private static Singleton instance;**

**private static readonly object lockObject = new object();**

**// Constructor privado para evitar la instanciación directa**

**private Singleton() { }**

**// Propiedad estática para acceder a la instancia única:**

**public static Singleton Instance**

**{**

**get**

**{**

**// Verifica si la instancia ya existe:**

**if (instance == null)**

**{**

**// Utiliza un bloqueo para garantizar la concurrencia segura:**

**lock (lockObject)**

**{**

**// Verifica nuevamente dentro del bloque de bloqueo:**

**if (instance == null)**

**{**

**// Crear una instancia en la escena si no existe una:**

**GameObject singletonObject = new GameObject("Singleton");**

**instance = singletonObject.AddComponent<Singleton>();**

**DontDestroyOnLoad(singletonObject);**

**}**

**}**

**}**

**return instance;**

**}**

**}**

**private void Awake()**

**{**

**// Verifica si ya existe una instancia al despertar:**

**if (instance == null)**

**{**

**instance = this;**

**DontDestroyOnLoad(gameObject);**

**}**

**Else**

**{**

**// Destruye el objeto duplicado si ya existe una instancia:**

**Destroy(gameObject);**

**}**

**}**

**}**

◢◤◇◥◣◥◤◢◤◆◥◣◥◤◢◤◇◥◣

En Corgi se utiliza el patrón Singleton en algunas de sus clases principales para garantizar que ciertos componentes o sistemas del juego tengan una única instancia en la ejecución.

1. **Investigar y explicar un patrón de POO y un principio, como, ejemplo: Patrón MK, principio K.I.S.S, DRY, en forma de diagramas de clase:**

RTA:

**Patrón de Diseño: Strategy (Estrategia):**

Podríamos usar el patrón de diseño “Strategy”, el cual se centra en definir una famila de algoritmos, encapsular cada uno de ellos y luego hacerlos intercambiables. Esto nos permite que el algoritmo varie independientemente de los clientes que lo utilicen.

El diagrama entonces se tendría que ver algo así como en el siguiente ejemplo:

◢◤◇◥◣◥◤◢◤◆◥◣◥◤◢◤◇◥◣



◢◤◇◥◣◥◤◢◤◆◥◣◥◤◢◤◇◥◣

En este diagrama, ocurre lo siguiente:

**Strategy (Estrategia):** Define una interfaz común para todos los algoritmos soportados. Es la clase base de la que se heredan todas las estrategias concretas.

**ConcreteStrategy (EstrategiaConcreta1, EstrategiaConcreta2, etc.):** Implementa los algoritmos específicos utilizando la interfaz definida por Strategy.

**Context (Contexto):** Mantiene una referencia a un objeto Strategy y puede invocar cualquier algoritmo definido por Strategy.

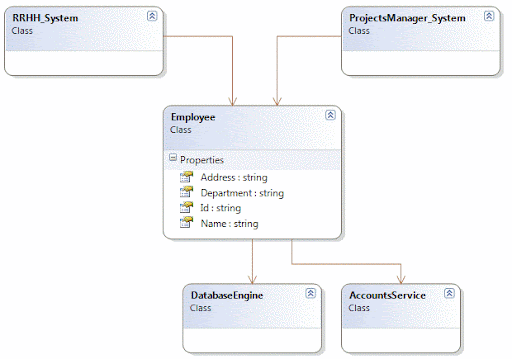
Este patrón permite cambiar dinámicamente el comportamiento de un objeto alterando la estrategia que utiliza.

**Principio de Diseño: Single Responsibility Principle (SRP):**

El Principio de Responsabilidad Única (SRP) establece que una clase debe tener una sola razón para cambiar, es decir, una clase debe tener una sola responsabilidad.

El ejemplo sería el siguiente:

◢◤◇◥◣◥◤◢◤◆◥◣◥◤◢◤◇◥◣



◢◤◇◥◣◥◤◢◤◆◥◣◥◤◢◤◇◥◣

En general, este diagrama de clases sigue el Principio de Responsabilidad Única (SRP). Cada clase tiene una única responsabilidad bien definida:

**RRHH System:** Proporcionar una interfaz para el usuario para interactuar con el sistema de RRHH.

**ProjectsManager\_System:** Proporcionar una interfaz para el usuario para interactuar con el sistema de gestión de proyectos.

**Employee:** Almacenar información sobre un empleado.

**Database Engine:** Almacenar y recuperar datos del sistema de RRHH.

**AccountsService:** Autenticar y autorizar a los usuarios del sistema de RRHH.

1. **¿Cómo funciona el ciclo de vida de un script en Unity?:**

RTA:

Un script asociado a un **“GameObject”** pasa a través de diferentes etapas durante la ejecución del juego. Este ciclo de vida consta de varios métodos predefinidos que Unity llama en momentos específicos, lo que permite controlar el comportamiento del script en diferentes puntos del tiempo de ejecución del juego. Aquí está el ciclo de vida básico de un script en Unity a grandes rasgos:

**1. \*\*Awake():\*\*** Este método se llama cuando se inicializa el script, justo antes de que se active el **“GameObject”** al que está adjunto. Es útil para inicializar variables y configuraciones necesarias para el funcionamiento del script.

**2. \*\*Start():\*\*** Se llama al comienzo del primer frame en el que el script está activo. Es útil para realizar inicializaciones adicionales que necesitan esperar a que otros objetos estén listos, como encontrar referencias a otros “**GameObjects**”.

**3. \*\*Update():\*\*** Este método se llama una vez por frame y es donde colocas la lógica principal de tu script que debe ejecutarse continuamente mientras el **GameObject** esté activo. Aquí se pueden manejar las entradas del jugador, actualizar el estado del juego, realizar cálculos de física, etc.

**4. \*\*FixedUpdate():\*\*** Se llama en intervalos de tiempo fijos y se utiliza para realizar cálculos de física o movimientos de objetos que no deben depender de la velocidad de fotogramas del juego. Es ideal para manipulaciones de física, como mover un objeto con **Rigidbody**.

**5. \*\*LateUpdate():\*\*** Se llama después de que se hayan actualizado todos los frames y se utiliza comúnmente para realizar acciones que deben ocurrir después de que se hayan realizado todas las actualizaciones en el frame actual, como ajustar la posición de la cámara para seguir a un objeto.

**6. \*\*OnDisable():\*\*** Se llama cuando el script o el **GameObject** al que está adjunto se desactiva. Es útil para limpiar recursos o detener procesos que ya no son necesarios cuando el objeto se desactiva.

**7. \*\*OnDestroy():\*\*** Se llama justo antes de que el script sea destruido. Aquí puedes liberar cualquier recurso que esté siendo utilizado por el script antes de que desaparezca.

El ciclo de vida permite controlar el comportamiento de los scripts en diferentes momentos durante la ejecución del juego, lo que brinda flexibilidad para realizar acciones específicas en cada etapa del proceso.