# PROGRAMACIÓN AVANZADA EN UNITY3D

# - Patrones Revisitados -

Enrique Sainz para Level-Up

# Patrones de Diseño: el Libro

- ***Design Patterns: Elements of Reusable Object-Oriented Software*** es un libro en el que se introducen varias estructuras reutilizables que solucionan un número de **situaciones comunes** durante el desarrollo de cualquier proyecto

- En este episodio veremos **varios de esos patrones**, comentando cómo algunos pueden ser abusados (**¿Singleton?**), mientras que otros pueden estar usándose menos de lo que deberían (**Command**)

# Los Patrones

- **Command:** Encapsula una petición como si fuera un objeto
- **Flyweight:** Separa los datos de objetos entre comunes a todos los objetos de un mismo tipo, y específicos a cada instancia
- **Observer:** Escucha (u observa) posibles notificaciones que puedan señalizar otros objetos
- **Prototype:** Consiste en crear objetos clonando otros objetos ya existentes denominados prototipos
- **Singleton:** Un único objeto, accesible estáticamente, proporciona servicios y mantiene el código limpio
- **State:** Para un objeto, establecemos un grafo de estados tales que, enumerando las condiciones que le permiten cambiar a otro estado

# - Command -

# Command - Definición

- En POO, un **Command** es la estructura que **convierte una llamada en un objeto**
  - Podríamos definirlo como un **callback** hecho objeto

# Command - Implementación

- Se puede hacer de varias maneras
  - **Estructuras específicas**
    - Si las acciones que se han de realizar no son específicamente llamadas a funciones, se puede hacer una implementación con **enumerados y parámetros**
  - **UnityEvents**
    - implementación en el editor
    - permiten especificar el objeto y la función que tiene que ser llamada
    - permiten añadir más de un objeto y función
    - también permiten sumar/restar listeners por código

# Command - Escenarios de Uso ( I )

- Imaginemos un juego en el que diferentes **acciones del jugador** pueden dar lugar a una **cutscene**
  - **GardenWorld:** al regar una planta puede lanzarse una cutscene para que se vea cómo crece el árbol de la vida
- Imaginemos que la misma acción puede dar lugar a **varias cutscenes**...
  - **GardenWorld:** al regar un cierto número de plantas puede lanzarse una cutscene que indica que ya se puede viajar con las balizas espaciales
- Al convertir el lanzamiento de estas cutscenes en **Commands**, podemos **secuenciarlas**, evitando que haya **conflictos** con el control de las cámaras

# Command - Escenarios de Uso ( II )

- **Undo / Redo:** en juegos de tablero o de cartas, el patrón **Command** puede ser especialmente valioso
  - **Ojo: no es trivial**, ya que las consecuencias de realizar el comando también tienen que poder deshacerse, y éstas pueden afectar a gran cantidad de objetos

- **Action Binding:** en lugar de utilizar el típico código en el cual comprobamos si **cierto control está pulsado**, podemos implementar un sistema unificado de controles que recibe **Commands**
  - Básicamente, lo que hace el nuevo **Input System** de Unity

# Command - Desventajas

- **Overhead** de llamadas y de uso de memoria
  - Eso en el año 2000 era importante; **hoy tenemos bastante CPU y memoria**
  - Aún así, cuidado si estamos tratando **cientos o miles** de objetos de esta manera

- **Exceso de flexibilidad**, si nos despistamos convertiremos cualquier llamada en un **command**

- En caso de los **UnityEvents**, desde el código no se sabe qué llamadas se hacen desde el editor
  - No se puede **optimizar** fácilmente el código (por ejemplo descartando código que no se usa, por ejemplo)

# - Flyweight -

# Flyweight - Definición

- **Flyweight** es un patrón por el cual nos aprovechamos del **parecido** o de la **repetibilidad** de un elemento para:
  - Ahorrar **memoria** de almacenamiento
  - Ahorrar **caudal** de datos
  - Ahorrar **tiempo** de desarrollo
  - Acortar **tiempos de carga**

- Ideal cuando tenemos muchos objetos, pero todos ellos comparten gran cantidad de datos
  - Ejemplo: **los árboles de un bosque**
    - En la realidad, **todos los árboles son diferentes**; pero en un videojuego **no se le da importancia al árbol individual**, así que podemos **repetir** sin que nadie se de apenas cuenta

# Flyweight - Implementación

- En general, basta con dividir una clase en **dos partes**
  - Los datos **específicos** de cada instancia
  - Los datos **comunes** a todas las instancias

- En **Unity** ya disponemos de soporte para **Flyweight** en varios sabores
  - Prefabricados
  - Scriptable Objects
  - Soporte para instantiated rendering

# Flyweight - Casos de Uso ( I )

- Reducción de **tiempo de carga**
  - Al utilizar **Prefabs de Unity**: las propiedades de los objetos **no** se deserializan por separado, sino que...
    - ... se **deserializa** el prefab...
    - ... para cada instancia se **clona** el prefabricado...
    - ... y sólo se deserializan los **cambios** de la instancia respecto al prefab

- Reducción de **caudal** de datos hacia la gráfica
  - Direct3D y OpenGL soportan **instanced rendering**
    - Pasamos el modelo y las texturas **una única vez**
    - Después pasamos **una transformada** por elemento
    - Es una **mejora sustancial** frente a tener que pasar **todos los polígonos de cada objeto**

# Flyweight - Casos de Uso ( II )

- Reducción de **uso de memoria**
  - Podemos usar **ScriptableObjects** para guardar los **datos comunes** a varios objetos
    - Un **ScriptableObject** representa un **único fichero** en disco
    - Si lo cargamos, sus datos ocupan un **único elemento**
      - ¡Aunque haya más de un objeto que los referencie!
    - Opuestamente, **dos prefabricados** en memoria ocupan memoria **individualmente**

- Reducción de **tiempo de desarrollo**
  - Inevitablemente, **repetir elementos lleva a reducir tiempo** de desarrollo
    - No es lo mismo **modelar cada árbol** de un bosque por separado, que utilizar **uno o dos modelos** repetidos

# - Observer -

# Observer - Definición

- Un **Observer** es un objeto que se engancha a un **Subject** y espera a que éste realice algún tipo de **notificación**
  - Por ejemplo: un **HUD (observer)** que “observa” a un **jugador (subject)** y que realiza algún tipo de **acción** cuando la cantidad de vidas de éste cambia

- De este modo, el **Subject** puede notificar lo que crea necesario a uno o varios **Observadores** - ***¡o incluso a ninguno!***
  - El hecho de que un evento en concreto **deje de ser importante** no hace necesario tener que dejar de notificarlo: *el coste de no llamar a una función es bastante negligible...*

# Observer - Implementación

- El libro propone utilizar dos clases base: **Subject** y **Observer**, de tal manera que:
  - el **Subject** dispone de una función ***AddObserver()*** con la que los observers se dan de alta
  - el **Observer** dispone de una función ***OnNotify()*** que los Subjects utilizarán para notificar que ha ocurrido algún tipo de evento
- De este modo, el nivel de desacoplamiento es muy alto, ya que únicamente las clases Subject y Observer se conocen entre ellas
  - No así sus derivadas
- En **Unity**, podemos utilizar **UnityEvents** o incluso delegates con el mismo propósito

# Observer - Escenarios de Uso ( I )

- Logros
  - Los logros son tremendamente variados, involucran todo tipo de objetos y componentes
  - Utilizar una estructura de Observer para recibir notificaciones cuando ciertos eventos se han dado es interesante
    - ¡Ojo! Hay que evitar lanzar notificaciones específicas (por ejemplo: las vidas del player han llegado a 0)
    - En su lugar, preferiremos notificaciones genéricas (por ejemplo: la cantidad de vidas de una entidad ha cambiado)
      - El sistema de logros ya determinará si la **entidad** es el jugador, y si la **cantidad de vidas** a la que se ha llegado es 0 para lanzar el logro **“Tu primer game over chispas”**

# Observer - Escenarios de Uso ( II )

- HUD
  - El HUD llega a mostrar una **gran cantidad de información** proveniente de muchas **fuentes diferentes**
  - Si todos los objetos que se muestren en el HUD tuvieran que conocerlo y llamar a **funciones específicas** se estaría generando un **alto acoplamiento**
  - En cambio, sería muy apropiado que todos estos objetos fueran **Subjects** observables por el **HUD**

# Observer - Desventajas

- **Demasiada gente observando**
  - Existe un cierto peligro de que **demasiada** gente escuche...
  - ... o de que los que observan **tarden demasiado** en procesar las **notificaciones**
  - ¡También puede provocar cierta cantidad de **caché-miss** - comprometiendo el rendimiento!

- **¿Demasiado desacoplado?**
  - En busca del máximo desacoplamiento, podríamos incurrir en que toda comunicación se realizase con **Observers**
  - Pero existen muchos casos en que varios objetos **claramente** trabajan juntos
    - **Cables eléctricos** con **generadores eléctricos** con **motores electricos**
  - Es mejor dejar la estructura Observer para cuando los objetos en juego **no están claramente relacionados**

# - Prototype -

# Prototype - Definición

- En POO un **Prototipo** es un objeto que se utilizará para instanciar otros objetos con las mismas funciones y datos
  - En C++ podemos entender que **una clase es un prototipo**...
  - ... pero en el **libro original** de ***Patrones de Diseño***, la idea sería más parecida a un **auténtico objeto** que tuviera **punteros a funciones y a datos**, y que llamase a dichas funciones a través de los punteros
    - Lo que se dice **un lío muy gordo**: el propio autor **no está a favor de los Prototipos entendidos en este sentido**

- La enorme **ventaja** de un **Prototipo** sería que, al hacer **copias** de éste y trabajar con dichas copias, n**o es necesario construir** un nuevo objeto

- ¡Este patrón es **mucho más interesante orientado a datos**!

# Prototype - Implementación ( I )

- Los prototipos orientados a datos se implementan usualmente con serialización.
- Imaginemos el siguiente objeto en un fichero serializado:

```
{
	Tipo: "CocheDeCarreras"
	Posicion: (100, 120)
	RotacionZ: 90
	Componente:"Motor"
	{
		Velocidad: 100
		Frenada: 200
	}
}
```

- Podemos deserializar el objeto cada vez que lo necesitemos...
- ...o podemos mantener una instancia en memoria del objeto y clonarla

# Prototype - Implementación ( II )

- Podemos ir aún más lejos:

```
{
        Prototipo: “CocheDeCarreras”
        Posicion: (120, 180)
}
{
        Prototipo: “CocheDeCarreras”
        Posicion: (100, 280)
}
```

- En este ejemplo, no solo se clonaría una instancia del ejemplo anterior, sino que además se deserializarían los datos de posición

# Prototype - Prefabricados

- A esta alturas habréis intuido que **Unity utiliza este patrón en los Prefabricados**

- Esto tiene ventajas en términos de **espacio en disco** y **tiempo de carga**...

- ... pero también a nivel de **desarrollo**
  - los diseñadores pueden **repetir un mismo objeto** hasta la saciedad...
  - ... con pequeñas o grandes **variaciones...**
  - ... lo cual es **muy conveniente** para ayudar a la **familiaridad** durante la partida

# - Singleton -

# Singleton - Definición

- Un **Singleton** es un objeto del que existe **una única instancia**, y que está **siempre accesible**

- El acceso a dicha instancia se realiza a través de un miembro estático de la clase, con lo que es accesible desde todo el código sin necesidad de transmitir referencias
  - Esto **en lenguajes como C++** ayuda a r**educir la dependencia** a nivel de ***#includes*** en clases intermedias
    - Ejemplo: si una **rueda** necesita una referencia a la **carretera**, no es necesario que **CFabrica**, ni el **CCoche**, ni el **CMotor**, ni **CEjeDeDistribucion** conozcan la clase **CCarretera**

# Singleton - Implementación

- La implementación aconsejada del singleton es de este estilo:

```
class CCarretera
{
	static CCarretera instance;

	// Lazy initialization
	public static CCarretera GetInstance()
	{
		if (!instance) { instance = InitInstance(); }
		return instance;
	}

	CCarretera InitInstance()
	{
		CCarretera carretera = new CCarretera();
		// Todo: Initialization

		return carretera;
	}
}
```

# Singleton - Ventajas

- **Acceso desde todo el código**
- Si la instancia no se usa, **tampoco se crea**
- Se pueden crear **derivadas** de la clase para obtener **diferentes comportamientos**

# Singleton - Crítica ( I )

- **Es una variable global elegante, pero global**
  - Si la usamos **en medio de un proceso**, nunca sabremos **quién más ni desde dónde** la está utilizando
    - **Un problema para localizar bugs** que puedan venir de su uso
  - Al ser global, invita a los recién llegados al proyecto a **utilizarla más de lo necesario**
    - Genera el **acoplamiento** que creíamos que habíamos arreglado
  - Suele **llevarse mal con el multihilo**
    - Al ser accesible desde todas partes, en cualquier momento, es difícil poner coto a su uso en **multihilo**
      - Problemas de **condiciones de carrera**, **abrazos mortales**, etc.

# Singleton - Crítica ( II )

- **Resuelve dos problemas aunque tengas uno**
  - El Singleton promete una única instancia, y acceso desde todo el código
  - Normalmente, se utiliza Singleton para el segundo motivo, pero...
    - ¿... por qué tiene que ser una única instancia? ¿No pueden ser varias y además tener acceso desde todo el código? ¿O incluso ninguna?
    - ¿... y si quiero asegurarme de tener una única instancia, pero no quiero ofrecer acceso a todo el mundo desde todas partes?

- **La Inicialización Perezosa (Lazy Initialization) nos quita el control**
  - Simplemente no sabemos cuándo ocurrirá
  - Según la complejidad del proceso de inicialización, o del tiempo que dure, puede suponer un impacto en nuestro programa

# Singleton - Alternativas ( I )

- **¿Necesitas un Singleton?**
  - Muchos Singletons son Managers de XXXX que proveen servicio a las instancias de XXXX...
  - ... servicios que, en muchos casos, las propias XXXX podrían darse a si mismos

- **Para tener una única instancia**
  - Podemos utilizar un booleano estático para saber si la clase ha sido ya instanciada:
    - En el constructor de la clase (o en el Awake() de un MonoBehaviour) comprobamos dicho booleano
      - Si es falso - ¡genial! Es la primera y, por tanto la única instancia; lo ponemos a true
      - Si es true - ¡fatal! Ya existe una instancia, así que la actual no es única; aserción, excepción, y hasta destrucción
  - Este sistema no asegura que exista una única instancia (la implementación original de Singleton sí) pero es suficiente en muchos casos

# Singleton - Alternativas ( II )

- **Para  tener acceso conveniente a una instancia**
  - Normalmente utilizamos un singleton para **acceder a una instancia de manera conveniente** desde todas partes
  - Pero hay alternativas...
    - **Pasarla por parámetro:** fácil y conveniente... pero podemos encadenar montones de llamadas que van pasando una instancia a gran profundidad
    - **Obtenerla desde la clase base:** si nuestro sistema tiene una estructura de clases amplia, es probable que una llamada a la clase base nos permita acceder a instancias de otras clases
    - **Obtenerla desde alguna variable que ya sea global:** ningún juego sin su variable GameLogic global
    - **Obtenerla de un “Service Locator”:** que básicamente es un sistema destinado a proveer de acceso a instancias de clases (de tipo servicio, pero clases al fin y al cabo)

# Singleton - Reflexión Final

- **Tal como se define en el libro** de **Game Design Patterns**, **Singleton** da más problemas de los que quita

- Pero **la idea general, y la conveniencia de un acceso global elegante** son interesantes

- La alternativa de utilizar **variables estáticas** en la clase, pero con aserción y excepciones **en lugar de Inicialización Perezosa** es válida en la mayoría de ocasiones
  - **Y en Unity más todavía**

# - State -

# State - Definición

- En POO un **estado** es **un objeto** que representa un **momento en el tiempo** en el que un objeto superior - **la máquina de estados -** se encuentra

- Cada estado tiene un número de **estados de destino** a los que la máquina de estados **transicionará** si se dan ciertas **condiciones**

- Ejemplo: los **estados de un personaje** de un juego de plataformas:
  - En pie
  - Agachado
  - Caminando
  - Corriendo
  - Saltando
  - Cayendo
  - Muriendo

# State - Implementación ( I )

- La implementación **más básica** de los estados es:
  - **una única clase**
  - ***enumerado*** para definir los **posibles estados**
  - ***switch()*** para determinar el **código que se ejecuta** en cada estado
    - incluido comprobar las **condiciones** de cambio de estado

- Este sistema es interesante, pero lleva a duplicar **gran cantidad de código**...
  - ... y en especial gran cantidad de ***switch()***, como ya se advertía en el libro **Clean Code**

# State - Implementación ( II )

- Una implementación **más interesante** de un estado consiste en que **el propio estado es una clase** que implementa las siguientes funciones
  - ***Enter()* -** esta función se utiliza para avisar de que se ha entrado en el estado
    - Sirve para inicializar variables (e.g. temporizadores) o para lanzar animaciones
  - ***Update()* -** esta función se llama en cada fotograma
    - Sirve para **actualizar el estado interno** del estado o de la máquina de estados, o del objeto que implementa dicha máquina
    - También sirve para comprobar las **condiciones** de cambio de estado y hasta ejecutar dicho cambio de estado
  - ***Exit()* -** esta función se utiliza para avisar que, a causa de que se cumple alguna condición, se está procediendo a finalizar el estado
    - Sirve para liberar recursos, o para realizar acciones de juego (“dropear loot”, dejar de transportar un objeto...)

# State - Implementación ( III )

- Los estados se pueden implementar como **derivadas de una clase abstracta base**:

```
abstract class State
{
	public abstract void Enter();
	public abstract void Update();
	public abstract void Exit();
}
```

- Además, los estados pueden ser:
  - **Estáticos:** si muchos datos de un estado son **compartidos**, podemos tener **los datos específicos de la máquina en un objeto**, y que el estado opere sobre ellos
    - ***Flyweight!***
  - **Instanciados:** cada estado tiene todos sus datos y referencias por separado, permitiendo **mayor flexibilidad** en general

# State - Implementación ( IV )

- En cuanto a la máquina de estados, tenemos dos opciones:
  - Implementarla con una clase específica que haga de contenedor de estados y gestione el estado actual
  - “Tejer” una red de estados, estáticos o instanciados, que se referencian entre ellos, generando así la propia máquina

¡ No es un extra !

¡ El autor ha metido más de lo que él mismo quería !

# - Extra! -

# - Máquinas de Estados -

# Máquinas de Estados Finitas

- Una máquina de Estados Finitas tiene un número finito de estados
- Además, la máquina sólo puede estar en uno de ellos
- Esto resuelve muchos casos, pero no todos
  - ¿Qué hacemos con los estados de un personaje de plataformas cuando, además, puede llevar o no llevar un arma? ¿Duplicamos cada estado?

# Máquinas de Estados Concurrentes

- Similares a las máquinas de estados, pero pueden estar en más de un estado

- Estos diferentes estados suelen estar separados en familias
  - Estado de movimiento frente a estado de disparo

- Aunque pueden haber estados que coincidan
  - Estado de movimiento
  - Estado de arma en mano derecha
  - Estado de arma en mano izquierda

# Máquinas de Estados Jerárquicas

- Las máquinas de estados jerárquicas están definidas por tener una estructura de clases derivadas de estados base que proporcionan partes de funcionamiento común

- Ejemplo:
  - State
  - GroundState : State, AirState : State
  - Crouched : GroundState, Standing : GroundState, Falling : AirState

- En este ejemplo, GroundState podría comprobar si el objeto no se encuentra en el suelo para pasar al estado Falling
  - Esto es común a todos los GroundStates

# Autómatas “Pushdown”

- Una extensión interesante de las **FSMs** es el **Autómata Pushdown**, que soluciona una limitación de las **FSMs**: **no disponen de historial de estados**

- El **Autómata Pushdown** consiste en disponer de la posibilidad de **“empujar”** el nuevo estado en una **pila**, para después **extraerlo y volver al estado anterior**

- Interesante como es, **no existen muchas aplicaciones** realmente interesantes para esta extensión **en videojuegos**
  - *O al menos yo no he podido encontrar una todavía*

# ¿Qué tan útiles son las FSMs?

- Las **FSMs** son conocidas por su **uso en IAs**, pero la tendencia actual *(¡el libro es del 2011!)* son los **Árboles de Comportamientos** y los **Sistemas de Planificación**
  - Behaviour Trees y Planning Systems

- Aún así siguen siendo **muy útiles** en:
  - IA
  - Gestión de Inputs
  - Navegación entre menús
  - Parseo de texto
  - Protocolos de redes
  - ...