PROGRAMACIÓN AVANZADA EN UNITY3D

- Patrones de Desacoplamiento -

Enrique Sainz para Level-Up

Patrones de Desacoplamiento (I)

- Haciendo memoria, recordaremos que la S en **SOLID** indica el **Single Responsibility Principle**
  - Es decir: un objeto debe tener **una única responsabilidad**, y por lo tanto debe haber **un único motivo para cambiar** dicho objeto

- El objetivo de este principio es **desacoplar** la funcionalidad de las partes del código, de tal modo que:
  - **Aprender el funcionamiento** de un código es más fácil
  - **Entender la organización** del proyecto es más sencillo
  - **Arreglar una "historia"** suele implicar cambiar sólo uno o dos módulos muy fáciles de trasladar a nuestro "cerebro de primate"

Patrones de Desacoplamiento (II)

- Los siguientes patrones nos ayudarán a **desacoplar** el código y conseguir una **arquitectura mantenible** en nuestros proyectos
    - **Component:** permite que **una única entidad** implemente funcionalidades de **dominio muy diverso** sin acoplamiento
    - **Event Queue:** desacopla el **momento de generación** de un mensaje del **momento en el que se procesa**
    - **Service Locator:** provee de un **punto de acceso** a un **servicio**, independiente de la clase concreta que lo implemente

- Component -

Component: Definición (I)

- ***El patrón Component permite que una entidad actúe sobre diferentes dominios sin que éstos se acoplen entre ellos***

- En el **año 2000**, la **POO** estaba en plena efervescencia, y **C++** era el lenguaje favorito para el desarrollo de videojuegos
  - No tardaría mucho en llegar **Flash** y, con éste, **ActionScript**, lo que daría lugar a una riada de **mini-juegos online**; pero eso es otra historia

- También en el **año 2000** empezaba a ser evidente que los videojuegos iban a convertirse en algo **mastodóntico**
  - Desarrollos de varios años con equipos de 50+ personas
  - Nada que ver con los juegos de garaje de los años 80

Component: Definición (II)

- En esos momentos era muy habitual ver **clases que representaban un objeto** (**Programación Orientada a *Objetos***, ¿no?)
  - Preguntadme por el **CAleman** de **Commandos 2**

- Pero estas clases acababan teniendo una gran cantidad de **responsabilidades**
  - El **CAleman** podía
    - **coger** objetos, **usar** objetos, consumirlos...
    - **buscar** caminos, **seguir** caminos, **entrar** por puertas, **escalar** paredes...
    - **mover** la cabeza, **ver** enemigos, **recordarlos**, **perseguirlos**...
    - **disparar**, **cambiar** de arma, **conducir** coches, **sentarse** en el asiento del copiloto...

Component: Definición (III)

- El **dominio** de todas estas acciones es muy variado
  - Se corre un gran peligro de que **acaben dependiendo las unas** de las otras a través de variables circunstanciales
- Peor aún: los aliados y los propios commandos controlados por el jugador tenían que **reimplementar** gran parte del código
  - Teniendo las **mismas capacidades** que los alemanes y alguna más, era inevitable
- La solución está en los **Componentes**

Component: Implementación (I)

- La implementación se basa en las siguientes clases:
  - **Entity:** se trata de una clase preparada para acoger y ejecutar Componentes
    - El equivalente en **Unity** es el **GameObject**
    - El equivalente en **Commandos 2** es el **CBicho** (preguntadme por éste)
  - **Component:** es la clase base de la que derivarán los componentes; permite implementar eventos y generalmente una función **Update()**
  - **Componentes derivados:** cada componente proporciona acceso a una única funcionalidad
    - Acceso al **HW de audio**, o al **sistema de audio** que lo gestiona...
    - Acceso al **HW de video**, o al **sistema de rendering** que genera el stream de render...
    - Acceso a **gestión de movimiento y colisiones**...

Component: Implementación (II)

- La implementación de los componentes puede ser **extremadamente básica**
  - Pueden ofrecer simplemente acceso al **HW** o las **librerías** subyacentes

- Pero también puede ser **muy avanzada**, y facilitar enormemente el uso de las funcionalidades
  - **Es el caso de Unity**; la mayoría de componentes **ocultan las complejidades de implementación**, facilitando que usuarios con menos conocimientos técnicos
    - Por ejemplo, **mis alumnos** de la escuela de arte; aunque **no tienen orientación técnica**, pero sin duda pueden utilizar Unity

Component: Implementación (III)

- En general, es conveniente que la interfaz esté **lo más alineada posible con la solución**
  - Ocultando los **detalles de implementación** facilitamos **implementaciones alternativas** pero que funcionen con los mismos datos

- También puede ser interesante proporcionar **interfaces segregadas**
  - Control de volumen **lineal** frente a control de volumen **por ganancia** (dB)
  - Control de luminosidad **lineal** frente a control de luminosidad **fotográfico** (lux)

Component: Ventajas

- Se reduce drásticamente la **necesidad de duplicar código**
- Se **fragmenta la funcionalidad** de tal modo que, una vez se dispone de los componentes, generar nuevos objetos implica:
  - Añadir unos cuantos **componentes ya existentes**
  - Añadir **muy poco código** de integración entre componentes
- Los componentes, correctamente desacoplados, **son muy reutilizables**

Component: Desventajas

- Los componentes tienden a ser **más complejos**, y se genera **más código**
  - Mantener el **desacoplamiento** implica **no poder aprovechar** ciertos datos o estados de los otros dominios
  - El código resultante es **más mantenible**, pero generarlo requiere **ingeniería avanzada**

- Esto implica **menos eficiencia** en rendimiento
  - Ocupa **más memoria**, el ejecutable es **más pesado**, son **más órdenes** que procesar...

- Pero a día de hoy gestionar **todos los componentes de un mismo tipo** secuencialmente permite **optimizar** con el patrón **Locality**

- Event Queue -

Event Queue: Definición

- ***Desacopla el momento de envío de un mensaje del momento en el que se procesa***

- Imaginemos que tenemos un sistema de sonido con una función ***PlaySound()***
  - Esta función **lanza un sonido**...
  - ... pero si éste **no está disponible** (porque hay que cargar aún los datos) **detiene la ejecución del thread actual** hasta que estos se cargan...
  - ... provocando un **parón** en la ejecución del juego

- ¿Cómo podemos cumplir con ***PlaySound()*** sin detener la ejecución?

Event Queue: Implementación (I)

- La respuesta es una **Cola de Eventos**
  - Almacenaremos cada **petición/notificación** en una cola que iremos procesando por orden de entrada
  - Esta cola puede implementarse como una lista, pero quizá lo más sencillo es un **Ring Buffer** (buffer en anillo)
    - Se reserva **suficiente espacio** en un **array**
    - Se gestiona el **índice al evento actual y al siguiente evento vacío**
      - **Incrementamos** apropiadamente cada índice, **ciclando** cuando sea necesario
  - En el caso del sonido, haremos **peticiones de carga asincronas**, y cuando éstas estén disponibles, lanzaremos el sonido

Event Queue: Implementación (II)

- ¿Por qué no podemos utilizar **delegates** y punto?
  - Si hablasemos de Unity, podemos utilizar el patrón **Observer** para resolver muchos de estos casos
  - Pero el libro está orientado a **programación avanzada en C++**
  - Con una implementación en C++ hay una **alta probabilidad de tener problemas de concurrencia** que den lugar a situaciones complejas
    - Como la **finalización de carga** durante la ejecución de un hilo que no es el que conviene a nuestro sistema de sonido
      - *Los sistemas de sonido suelen gestionarse en su propio hilo*
    - O la llamada a **PlaySound()** cuando el hilo está ocupado

Event Queue: Ventajas

- La cola nos permite **gestionar “en diferido”** las peticiones y notificaciones

- Con muy poco esfuerzo extra, podemos no sólo desacoplar el momento, sino también **desacoplar el receptor del evento**
  - A través de un sistema **similar al Observer**, pero en el cual se observa la cola de eventos a la espera de **determinar qué eventos** podemos gestionar

- Y se puede incluso añadir una gestión extra para casos especiales:
  - **Sonidos que se acumulan** y en los que queremos gestionar su prioridad
  - **Mensajes que sobreescriben otros mensajes**, permitiéndonos gestionar sólo los más nuevos

Event Queue: Desventajas

- El código **se vuelve más complejo** a corto plazo
  - Ya no es tan fácil como **llamar a una función**
- Pero una **implementación genérica** de la **Cola de Eventos** puede ayudar a **amortizar** este esfuerzo a medio y largo plazo

- Service Locator -

Service Locator: Definición (I)

- ***El patrón Service Locator provee de un punto de acceso a servicios sin acoplar dichos servicios a la clase que los implemente***

- En ocasiones, necesitamos una **funcionalidad transversal**
  - Por ejemplo **el audio**: se puede lanzar todo tipo de sonidos desde **todas partes**, tanto durante la partida como durante los menú

- Podríamos resolver esto con un **Singleton**, o incluso con una **clase estática**
  - Pero esto haría que **todo el código dependiera** de la implementación de estos...
  - ... y cualquier cambio en ellos generaría una **ola de cambios** en todas partes

Service Locator: Definición (II)

- Para estos casos, podemos utilizar un **Service Locator** que proporcione la funcionalidad a través de un servicio
  - Nosotros podemos **pedir servicios** al **Service Locator**
  - El **servicio** siempre tendrá una **clase base** para proveer la funcionalidad...
  - ... pero la **implementación** de ésta quedará oculta tras el **Service Locator**

Service Locator: Casos de Uso

- Ofrecer **diferentes implementaciones** para un Servicio
  - **Mismo servicio**, pero hace cosas **extra**, o **diferentes**, o utilizando **otras librerías**
    - Eg: servicio de **dibujado de sprites** sobre **DirectX, OpenGL, Metal, Vulcan...**
- Ofrecer **diferentes interfaces** (Servicios) para **una implementación**
  - **DirectX o XNA** ofrecen acceso a una **misma funcionalidad** (core gráfico) a través de diferentes Servicios - **versiones** cada vez superiores de sus módulos
  - Esto asegura poder **evolucionar la funcionalidad** (el core gráfico) sin que los programadores tengan que **evolucionar el código** a cada cambio
- Ofrecer servicios **actualizados de manera dinámica**
  - Los servicios pueden ser **estáticos**, pero también **cargarlos desde una DLL**
    - Caso de **DirectX** y sus **librerías RunTime**

Service Locator: Implementación (I)

- Esta podría ser una implementación sencilla:

```
class Locator
{
	public:
		static Audio* getAudio() { return service_; }
		static void provide(Audio* service) { service_ = service; }

	private:
		static Audio* service_;
};
```

- En este caso, la clase **Audio** sería la **interfaz**, y cualquier derivada de **Audio** podría ser utilizada como **implementación**

Service Locator: Implementación (II)

- Existe la posibilidad de que haya problemas si no se ha provisto de **ninguna implementación** aún.
  - Punteros a null, o referencias nulas
  - Gestionar estos casos es farragoso, e incrementa el código
- Una manera de evitarlos es disponer de una **implementación nula**

```
class NullAudio: public Audio
{
	public:
		virtual void playSound(int soundID) { }
		virtual void stopSound(int soundID) { }
		virtual void stopAllSounds() { }
};
```

```
class Locator
{
	public:
		static void initialize() { service_ = &nullService_; }
		static Audio& getAudio() { return *service_; }
		static void provide(Audio* service)
		{
			if (service == NULL) { service_ = &nullService_; }
			else { service_ = service; }
		}
	private:
		static Audio* service_; static NullAudio nullService_;
};
```

Service Locator: Implementación (III)

- Existen diferentes opciones a la hora de **proveer los servicios**:
  - Se proveen **en tiempo de compilación** o linkado
  - Los provee el software que los usa
    - La librería **ZLib** permite proveer un **servicio de lectura de ficheros**
      - ¿O era **libpng**? Ya no me acuerdo...
    - Esto permite que los ficheros provengan de las **funciones normales** del operativo (***fopen(), fclose(), fread()...***) o (por ejemplo que no tiene que ver con el **PanicBCN**) de un **fichero compactado y cifrado** con un sencillo **algoritmo XOR**
  - Se obtienen de **librerías dinámicas (DLLs)** ex-profeso
    - Un protocolo permite **reconocer** dichas librerías y pedir a estas que **provean** los servicios

Service Locator: Ventajas

- Nos obliga a diseñar una **interfaz genérica**, **adecuada al problema** y no a la implementación de la solución
- Podemos hacer muchos **cambios a la implementación** del servicio sin que esto **afecte al resto del código**
- Podemos ofrecer **implementaciones alternativas** que generen funcionamientos extra
  - Siguiendo con la idea del audio: ¿qué tal si el servicio Audio también **generase Logs**, o incluso señales Midi que trabajaran con **actuadores neumáticos**?
- Podemos **mapear Servicios antiguos** (Audio) a nuevos servicios (AudioV2) o a Servicios que utilicen una **implementación diferente** (AudioFMOD)

Service Locator: Desventajas

- Mirando el código, **no es evidente** qué servicio va a proveer la funcionalidad en ese momento ni qué va a hacer

- Los **cambios en la funcionalidad** importantes implicarán **cambios en la interfaz** del Servicio, que pueden llevar inevitablemente a **cambiar el código** en todas (o casi todas) partes

- El **Service Locator** sigue siendo una variable global, y su complejidad puede ser alta
  - Vale la pena estudiar en cada caso si es mejor o no que un **Singleton**

Service Locator: en los Motores de Juego

- En la actualidad, los **motores de juegos** ofrecen una interfaz a muy alto nivel, **por encima del código**
  - La funcionalidad la proveen **nodos o componente**

- El **Service Locator** puede estarse utilizando por debajo, pero a nivel del propio lo más parecido - y con distancia - sería el **Package Manager**
  - Y no proporciona esa **capa de abstracción**

- Aún así, como inspiración para **ofrecer diferentes interfaces** para una misma **funcionalidad en evolución**, es interesante