08/04/2022

Prueba Técnica Unity Developer

Mauro Fuentes

**Explicación General**

Para esta aplicación decidí seguir un patrón mediador. De esa manera logro que todos los botones y componentes concentren sus llamados en una clase *Controller*. En general intento seguir una lógica *tell, don’t ask* para todas las clases para reducir su acoplamiento y aumentar la cohesión. Es por eso que, a pesar que la clase *Controller* (el mediador) tenga muchos métodos, en realidad hace poco más que organizar las clases que controla. Solo da órdenes y encausa los eventos de las clases que controla. Su trabajo principal es controlar el orden de ejecución del juego y leer un diccionario.

Si bien es cierto que las clases controladas podrían implementar una interfaz para abstraer aun más el código, también es cierto que todas ellas están absolutamente desacopladas entre sí y del *Controller*. Esto permite, por ejemplo, crear otros controladores o más botones, sin necesidad de cambiar el código, solo extenderlo. Básicamente estas clases representan un *view*.

El lado negativo de tener esa clase *Controller* es que está acoplada a los miembros que maneja. Sin embargo, cuando el *Controller* da ordenes a las clases, golpea contra clases públicas tipo *getters* para esconder la implementación interna. Si algún campo tuviera un entorno público es porque su manipulación no es crítica.

Para guardar los datos de números cardinales y su equivalente en español decidí utilizar un diccionario<int, string>. Por ejemplo, *Cardinal* (1) y *Spanish* (“uno”). Que este diccionario sea un *ScriptableObject* nos permite generar N diccionarios de diferentes combinaciones. Además, elegí que la *key* sea un *int*, porque, en un futuro, nos facilitaría hacer el sistema más complejo con funciones aritméticas. Los números en palabras bien podrían estar en un JSON. O bien, escribir una clase que funcione como *parser* para no tener que hacerlo a mano.

Las ciclos de Unity que están vacíos (p. ej. *Start()*) están para poder desactivar momentáneamente la clases desde el Inspector.

Para pruebas de control usé un Debugger muy simple que prueba las clases en *runtime*.

**Cadena de ejecución**

Utilizaré ↳ para clases que llaman a otras clases y ⤑ para respuestas a eventos.

↳ StartGame()

↳ GetRandomSpanishNumber()

↳ RunStartAnimation() // fire and forget

⤑ OnStartAnimationFinished()

↳SetCardinalButtonsForNewData()

↳RunCardinalINAnimation // fire and forget

⤑OnCardinalInAnimatonFinished()

↳ActivateCardinalCanvas()

\_Stand by for the player\_

⤑ReactToCardinalButtons

↳Check

￫if max attemps RestartEspecial()

￫if win Restart()

￫else return

**Classes Lookup**

Los *lookup* son clases que utilizo para buscar una dependencia, cuando las dependencias usan componentes a fin. Por ejemplo, si necesito tres componentes con TMP y no sé cuál es cuál. En vez de barrer todos los objetos de la jerarquía, le asigno un tipo aun cuando quede anémica. El uso y la refactorización me indican si necesito asignarle comportamiento.

**Animaciones**

Al principio, había planificado hacer las animaciones a través de código, pero demoraba mucho más tiempo de lo pensado y, siguiendo su recomendación, lo utilicé para pulir el código.

Sí quiero destacar, que dejé la clase *Animator Controller By Code* porque me siento particularmente orgulloso de ella. Con pocas líneas de código, quedó preparado un sistema totalmente abstracto en donde únicamente se puede extender el código. La lógica en sí es limpia y no se requiere de ningún tipo de *if()*, *else()* y mucho menos de *enums* para elegir qué tipo de animación queremos.

Detrás de las animaciones escondí toda la implementación de código que solo debe conocer el *Animator Controller*. Lo que lamento es que, por haber cambiado de planes a último momento, esta clase es altamente redundante.