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 de que la clase *Controller* (el mediador) tenga muchos métodos, básicamente organiza, da órdenes y encauza 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 extendiendo el código, sin necesidad de cambiarlo. 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 órdenes a las clases, golpea contra clases públicas tipo *getters* que esconden los datos de 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 como *strings* bien podrían estar en un JSON, o bien, se podría escribir una clase que funcione como *parser* para evitar hacerlo a mano.

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

Para pruebas de control, usé un *Debugger* muy simple que testea 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 afines. Por ejemplo, si necesito tres componentes con TMP y no puedo distinguirlos. En lugar de barrer todos los objetos de la jerarquía, asigno un tipo a las clases (aun cuando estas internamente queden anémicas). El uso y la refactorización me indican si necesito asignarles comportamiento.

**Animaciones**

Al principio, había planificado hacer las animaciones a través de código, pero demoraba mucho más tiempo de lo esperado. Siguiendo su recomendación preferí usar ese tiempo 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.

Lamentablemente, la clase *Animator Controller* tiene algunas redundancias debido a que cambié de planes a último momento.