Design Analysis

Class Design

Analyzing the exercise I identified two big blocks of needs. The first one grouped the **logic around the activity flow**. Presenting the target value to the player, displaying all the options with the solution included, handling the option selection and acting based on whether the selection was right or wrong. Finally, managing the game win-lose situation built on the player choices. The second one grouped the **logic around the animation flow**. The same element will experience different animation processes (type of animations, number of animations, etc) and different elements will share the same animation process, simultaneously or in different timeframes.

Having these details identified I decided to build an Animation Manager system based on similar principles as the factory method is built around. For this specific scenario animations were going to affect Text elements but this could change and new needs may easily require other elements to also experience the same animations. However, I wanted this animation to share some conditions attributes as the animation duration. Due to this set of conditions I decided to go for an Abstract class representation of what I identified as an animation. Based on the component type of the animation's subject, the animation will execute the specific method of the children class. Let the RunFadeInOut() method serve as an example.

An element may have run through a single animation or a sequence of them. That is why I created the AnimationBundle for, to manage a sequence of *n* animations for a single object. Finishing the animation sequence triggers different effects on different parts of the application. I decided to use the observer pattern to notify those different elements that will act accordingly. For example, when the target number finishes its animation at the start of the application, then the set of options have to start showing up, or later on, when the set of options are already displayed, the interaction with the buttons must be enabled.

During the execution of the application several elements will go through different sets of animations. I created an AnimationManager static class so any element part of the execution will interact with a single instance of this AnimationManager. Then different instances of running AnimationsBundles would not interfere with each other, existing AnimationBundles could be reused and so on.

Then, about the activity flow. I created a Level class as a monobehaviour and defined it as a Singleton. I made it inherit Monobehaviour to manage the level's gameobjects through serialization. Since at the end I didn't make it run based on the update of the engine(works mainly under requests and events, I considered going without it, but thought it was more comprehensible this way, holding the serialization of the level's gameobjects representation.

The Level class is in charge of the level's "backend" main flow and leads it. The LevelVisuals apply the “frontend”. The level class manages the instantiation and destruction of the dynamic ***Option*** objects as well as enabling and disabling the interaction based on the current level status. The option objects are created as a Monobehaviour because they are going to be the representation in the game scene of those generated random values, which will be interactable for the user.

Last but not least, I included two TestClasses at the beginning of the development. The first one intended for the Edit Mode and the other one for the Play Mode. I would have liked spending more time on those classes. It’s obvious that they cover a really small part of the code and I consider testing a core part of the development, but I already saw myself exceeding the dedication time and analyzing the nature of the exercise, I chose to leave it like a representation of what had to be done, having them as I already mention available for both Edit and Play mode, and targeting both the animation logic and the game flow logic. Adding the rest unit tests on top of that wouldn’t interfere with the rest.