Name: Ryan O’Connor  
Date: October 19, 2014

2pts each

Explain a scenario/use case for each of the following design patterns (come up with a use case that has not been talked about in class or in the power points):

1. Adapter

If you’ve got a code base with an already working retail system that now needs to accept Bitcoin, you can create an adapter class to adapt your new Bitcoin class to work with your Payment class. Since C# doesn’t support multiple inheritance, the Adapter pattern can be used to implement polymorphism between incompatible classes.

1. Bridge

The bridge pattern could be used for the “Choose Player” screen in a fighting video game. The implementation is the fighter that you see on screen and the implementation classes are the list of fighters at the top of the screen that you scroll through. As you scroll through the list of players, the implementation points to a different implementation class of the “player” or “fighter” abstraction where each fighter have their own unique properties such as strength, endurance, special moves, etc.

1. Composite

The composite design pattern is useful for when a user does not care about the difference between a composition of objects and a single object. This can be useful for hierarchies of data such as music storage. You can organize your music by Playlists which can hold Songs or more Playlists. Each Playlist has the ability to hold as many Songs and Playlists as needed, so this pattern can create a tree like structure with each node holding an array of nodes. So for instance you could break down your music with a root of “Playlists”, which holds playlists “Spring”, “Summer”, “Fall”, and “Winter”. Each can hold playlists “Jazz”, “Rock”, “Classical”, etc.

1. Decorator

If you need to add behavior to an object without affecting the original functionality of the object or other objects in its class, then the decorator pattern is useful. In a fighting video game where you’ve got an object for a fighter “Ryu” and classes for his attacks such as “Kick”, you can add a decorator that can give his “Kick” class different behavior if he is powered-up, poisoned, tired, etc. His “Kick” decorator wrapper would hold an instance of “Kick” and return all of “Kick”s parameters to kicks decorator class “KickPoweredUp” where you could implement methods to increase/decrease “Kick”s strength parameter, maybe its target range, its speed, etc.

1. Façade

The façade pattern would be useful to tie together API’s with their own class. For instance, if you’ve got fragmented graphics API’s in your library you can pull them all together by placing instances of them within their own façade class “Graphics”.

1. Flyweight

The flyweight pattern could be useful for rendering objects in a video game. In a car racing video game environment you have many recurring objects that need to be displayed very quickly such as sections of pavement, buildings, trees, guard rails, etc. With the flyweight pattern you don’t need to create a new building for each portion of the track. When your environment needs a “PoliceStation” or “GasStation” it creates it, and if the environment needs that building again at a different part of the map it will point back to that same instance without needing to create an entire new one.

1. Proxy

The proxy design pattern could be useful for security purposes or to hide paid functionality in a “trial membership” version of an application. If you’ve got photo editing software that requires you to pay to use their “3D” feature set, they can build a “TrialVersion” proxy class to interface with the real application engine and exclude the “3D” API.