Singleton

I applied the singleton pattern to the InputHandler within my game. I decided that this would be a good use case as there should only ever be one of these at a time, and if there were more than one present in the scene, it could cause issues regarding player input.

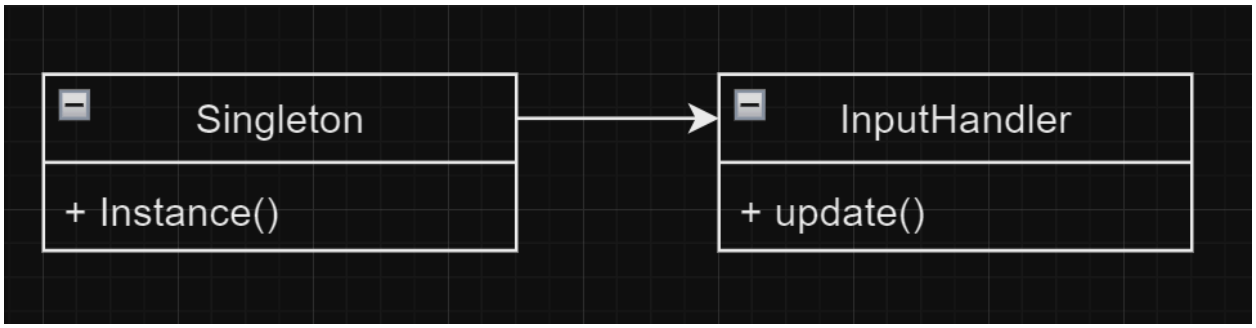
To do this, I created a base singleton class that inherits from MonoBehavior. This Singleton class simply checks to see whether or not a class of this type exists in the scene already, and if it does, it destroys the new one.

```
public class Singleton<T> : MonoBehaviour where T : Component
{
    private static T _instance;
    public static T Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = FindObjectOfType<T>();
                if (_instance == null)
                {
                    GameObject obj = new GameObject();
                    obj.name = typeof(T).Name;
                    _instance = obj.AddComponent<T>();
                }
            }
            return _instance;
        }
    }
    Event function
    public virtual void Awake()
    {
        if (_instance == null)
        {
            _instance = this as T;
            DontDestroyOnLoad(gameObject);
```

```
            _instance = FindObjectOfType<T>();
            if (_instance == null)
            {
                GameObject obj = new GameObject();
                obj.name = typeof(T).Name;
                _instance = obj.AddComponent<T>();
            }
        }
        return _instance;
    }
}
Event function
public virtual void Awake()
{
    if (_instance == null)
    {
        _instance = this as T;
        DontDestroyOnLoad(gameObject);
    }
    else
    {
        Destroy(gameObject);
    }
}
```

To apply this to my InputHandler, rather than deriving from the MonoBehavior class, I changed it to a Singleton class of type InputHandler. Since the Singleton class derives from MonoBehavior, this didn’t change any functionality within the script.

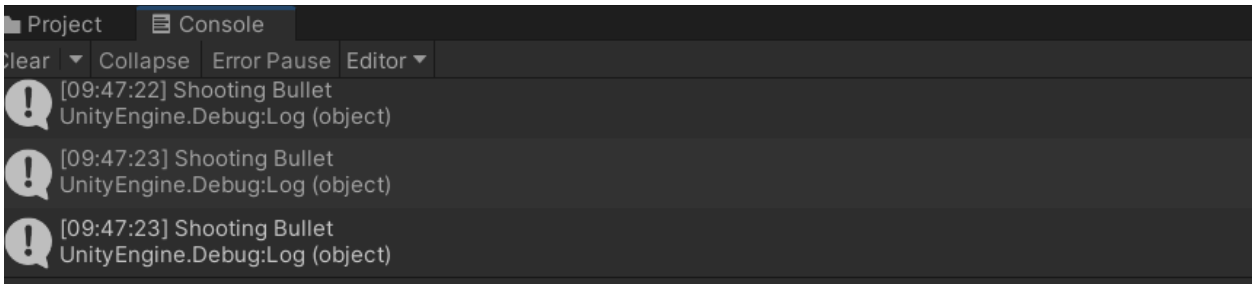
```
namespace Singleton
{
    1 asset usage   1 usage
    public class InputHandler : Singleton<InputHandler>
    {
        private PlayerController _playerController;
        private global::Command.Command _buttonA, _buttonD, _buttonW, _buttonS, _buttonSpace;
```



Command Pattern

I applied the Command pattern to the movement and shooting functionality of the player's tank. This seemed like a valid area to apply this pattern because as the game is developed more, more controls will need to be added, and using a command pattern makes this easier. It will also make it easier to swap control schemes in the future.

To implement this design pattern, I created a base Command class that houses an "execute()" function, and for each direction/control, I created a class that derives from this base class, which overrides the execute function, allowing me to decide what action occurs when this input is pressed.

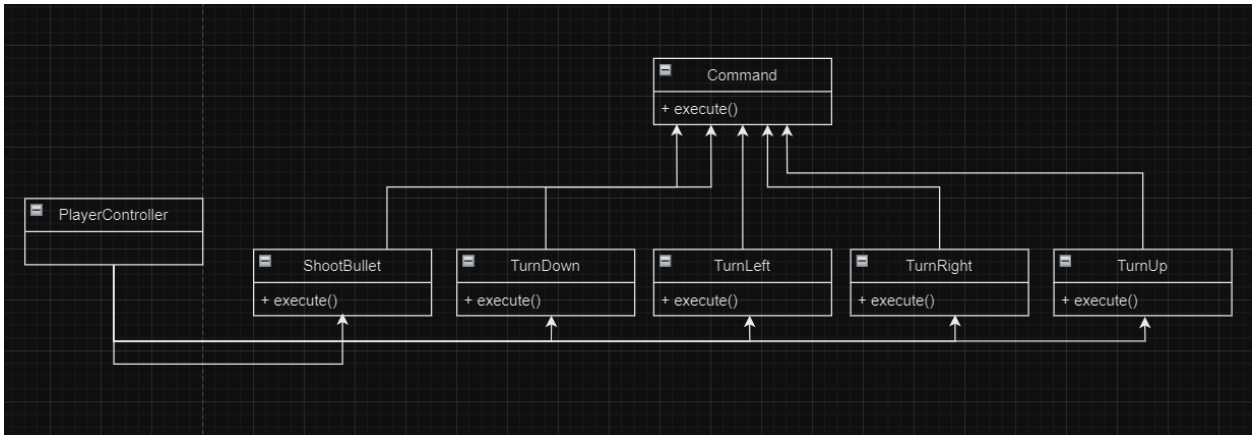


```
namespace Command
{
    6 usages   5 inheritors
    public abstract class Command
    {
        Frequently called   5 usages   5 overrides
        public abstract void Execute();
    }
}
```

```
namespace Chapter.Command
{
    1 usage
    public class TurnUp : global::Command.Command
    {
        private PlayerController _controller;

        1 usage
        public TurnUp(PlayerController controller)
        {
            _controller = controller;
        }

        Frequently called   0+5 usages
        public override void Execute()
        {
            _controller.Turn(PlayerController.Direction.Up);
        }
    }
}
```



Observer Pattern

While I didn't have time to attempt the implementation of the observer pattern, if I did have time, I would have applied it so that there were UI updates according to how many enemies the player kills. I would create a subject class that holds the list of observers, in this case, the player. Everytime the player hits an enemy, it would notify the UI elements (the observers), and let them know to update (their score, enemies killed count, etc).