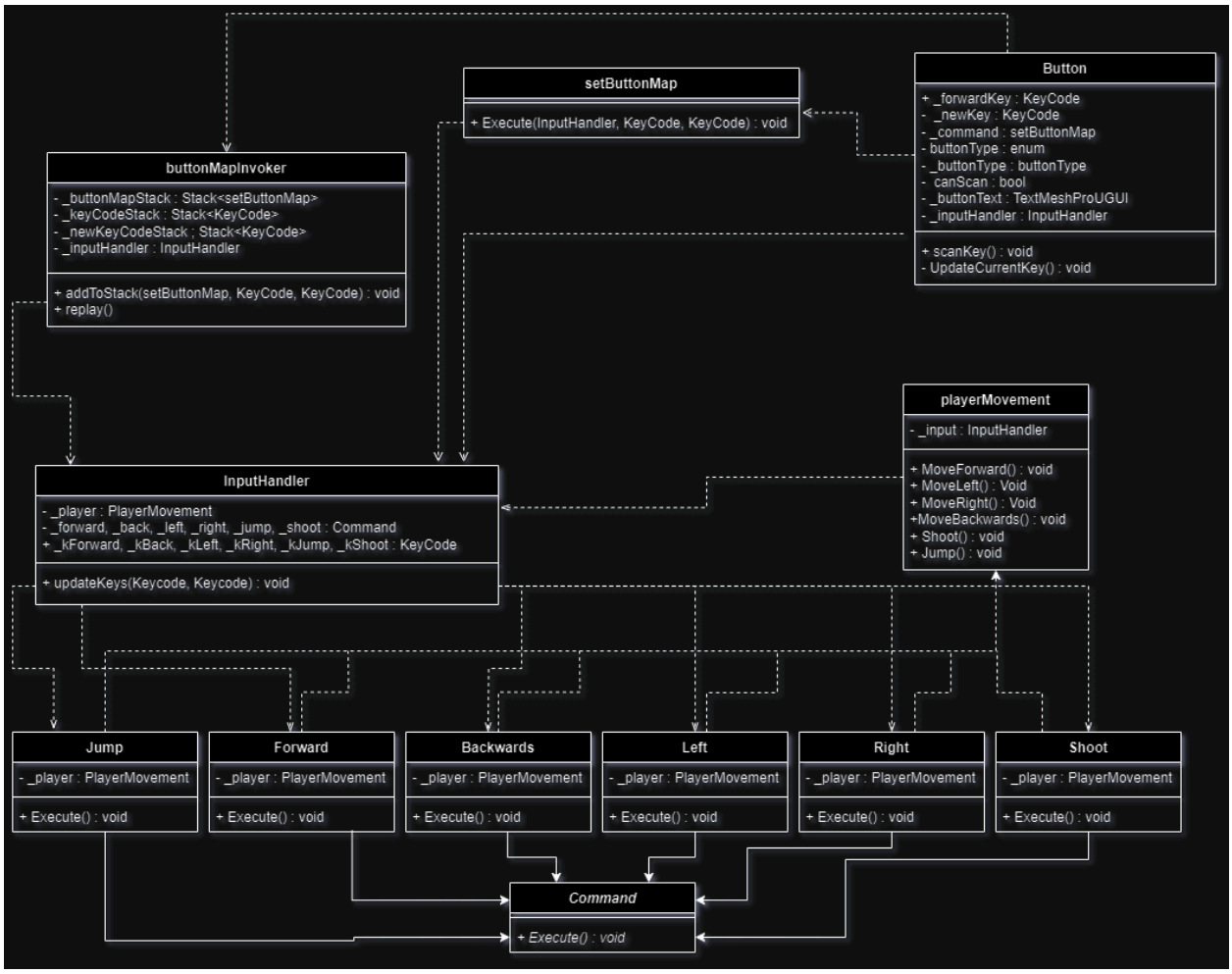
Game Engine Project Implementation Diagrams and Explanation

Andrew's Implementations

Command Implementation:
<u>Custom Button Mapping</u>

Diagram



Explanation:

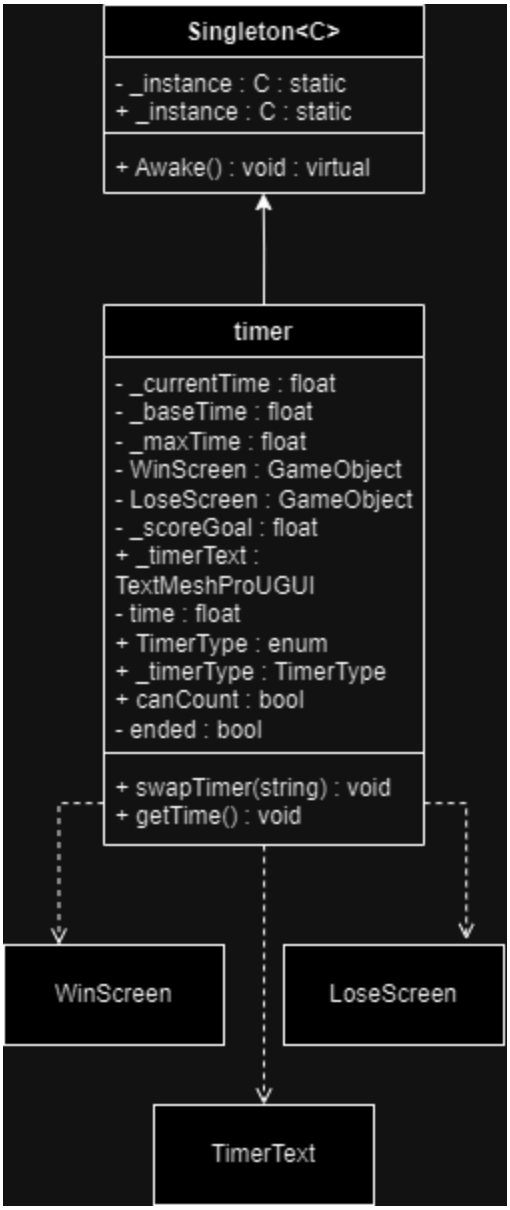
One of the ways that the command design pattern was implemented was with a custom button mapping system. The way I went about this was to first make an abstract Command class that had an abstract function called Execute. After that I made 6 different classes, one for each input action that the player could do in the game. Each of these classes inherited the base Command abstract class. The playerMovement class, instead of having the logic inside for if the player hits a certain button a certain action plays, it has 6 different methods that perform the certain action when called. The different command classes have overridden Execute methods that, when called, call the corresponding method in the playerMovement script. I.e: when the Execute method is called in the Forward class, it runs the MoveForward method in the playerMovement script. The InputHandler class is the object that actually deals with and executes these commands. In this class, it has variables saved for the different commands and the different keys that correspond to those commands (i.e: _kForward = W). If able to, the InputHandler always checks for if any of the stored keys are being pressed, if they are then it executes the corresponding command and also saves it for use in a different system. The InputHandler also has a method called UpdateKeys that takes in two keycodes. What this method does is first compares the first keycode with all of the currently saved keycodes. If it finds a match it swaps that saved keycode with the second keycode. The buttonMapInvoker script deals with storing and undoing any button key swap the player has done. It has two methods, addToStack which adds a command to a command stack, and two keycodes to two separate keycode stacks. It also has a replay method that, when called, runs the execute command in the latest command added to the command stack (if there is any in the stack) and fills in it's keycode requirements with the latest keycodes in the keycode stacks. Since it's an undo, it swaps the new and old keycodes around so instead of first sending the current keycode then the new keycode it sends the new keycode first then the "current" (old now) keycode to essentially just swap the latest change by the player. After that it pops the latest item in each stack. The actual button mapping is handled by the Button script. It's made so that it can be placed onto a button gameobject, and stores a base KeyCode that determines which KeyCode it swaps. It has multiple methods, one of these is called UpdateCurrentKey that checks to see what internal enum value is selected for the object (these values are: forward, backward, left, right, shoot, and jump) then sets it's keycode variable to the corresponding keycode on that action in the inputhandler. (i.e. if enum value is set to forward, it sets the internal key variable to the _kForward variable of InputHandler). It has another method that, when called, allows the script to start scanning for keypresses. If any key is pressed it runs the execute command in a secondary command script called setButtonMap. This method takes in an inputhandler and two keycodes then runs the UpdateKeys function in the inputhandler and inputs the two keycodes, the first one being the current key variable saved in the Button script and the second one being the new key just entered by the player. After that it runs the addToStack method in the buttonMapInvoker script with the command being run, the current keycode that it changed, and the new keycode that the player inputted. Right after that it makes it so it is no longer scanning for keypresses until the ScanKey method is called again. It was implemented like this so that other implementations of the command pattern using player inputs could be easily added since every player input action is handled by commands. Aside from player inputs, the input handler was designed the way it was, where instead of hard searching for individual key presses it searches for keypresses

saved to variables so that the keys could be easily modified which allowed the custom key mapping system to work and function properly. The Button portion of this and the buttonMapInvoker were done the way they were (i.e each key change is a command that gets saved) so that there's a log of every single key change done by the player that can be easily undone by them in case they need to do that. Our interactive scene benefits from this because it allows the user to customize their experience and gives the game an extra depth.

Singleton Implementation

Timer

Diagram



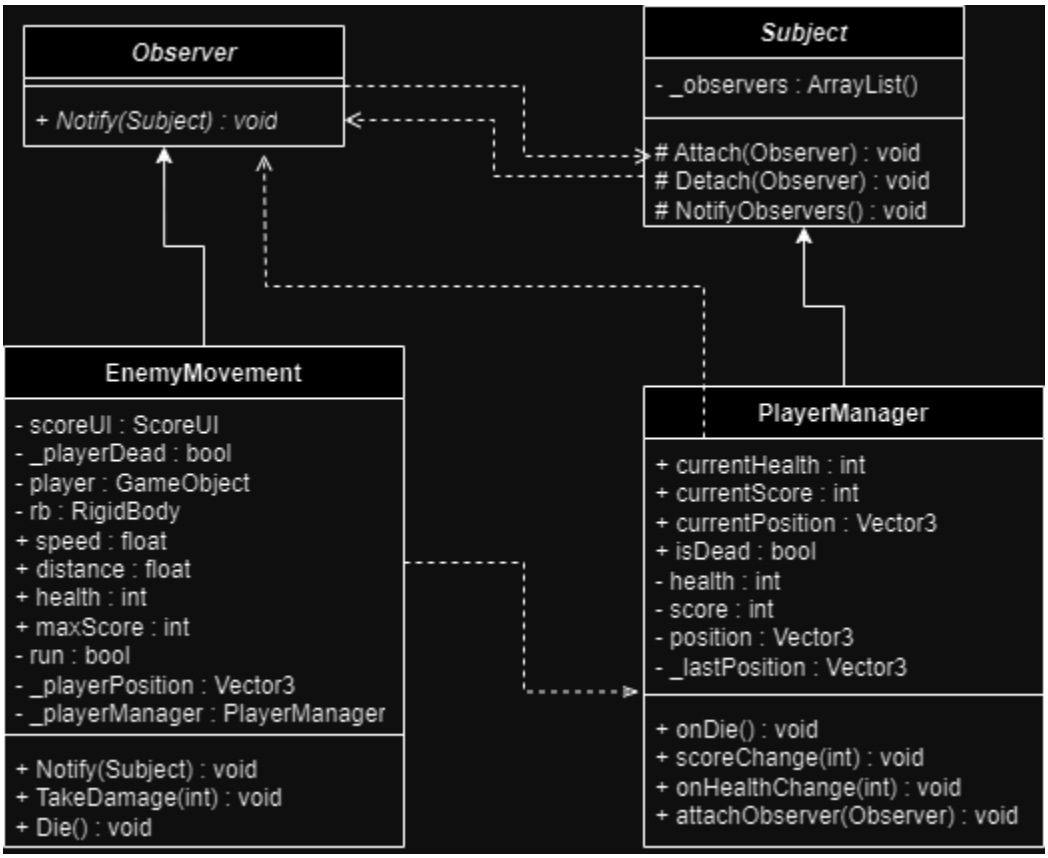
Explanation

One of the ways that the singleton pattern was implemented was with the timer system. The way this was implemented was by first creating the base singleton class that takes in a variable instance of its own type by first checking to see if the instance variable is null and if it is it searches and sets itself as an object of its own type. If it's still null it creates a new GameObject and adds itself as a component. If the original check is false it just returns the instance variable. On awake, it checks again if _instance is null, if it is it sets _instance to itself then adds the gameObject the script is a part of to DontDestroyOnLoad. If it isn't null it destroys itself. This allows the singleton to only have one of itself in the scene at all times. Timer works by inheriting this and then has all of the timer functionality inside of it. It was implemented this way because the timer is not only going to be a constant in the scene but there's only going to ever be one of

it so to avoid accidentally having more than one I made it a singleton. Making it a singleton also makes it a global Instance which makes it easier for other scripts to access its parts. Since the timer is crucial for the win/lose state, this feature makes implementing different systems easier.

Observer Implementation
<u>EnemyMovement & PlayerManager</u>

Diagram



Explanation

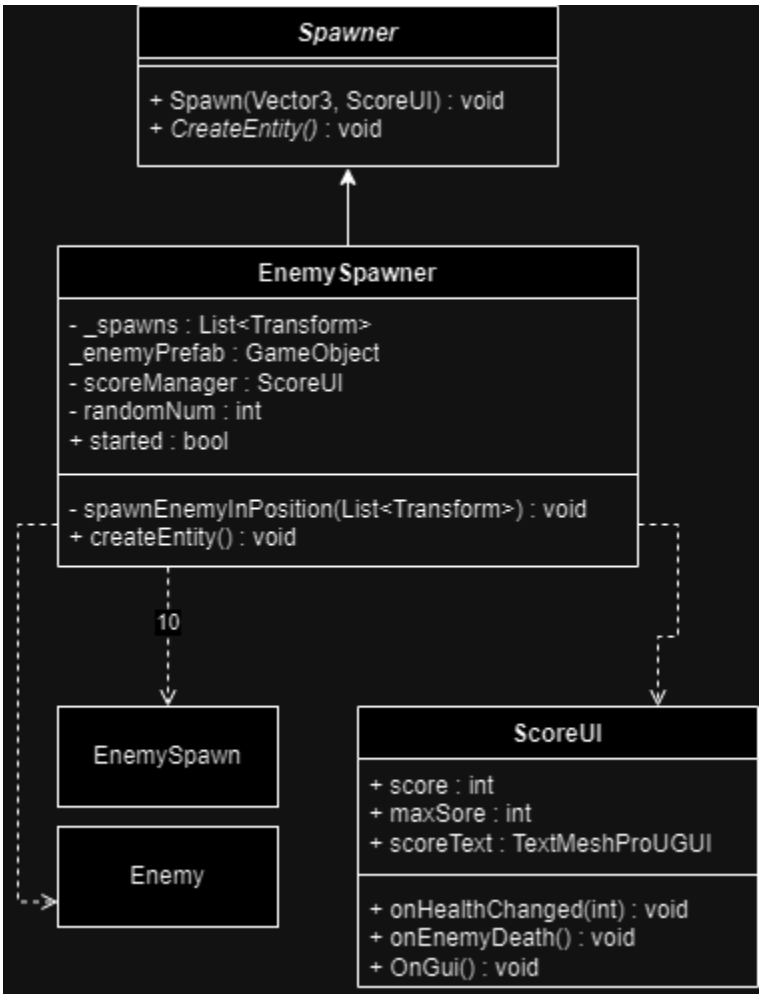
One of the ways the observer design pattern was implemented was with the EnemyMovement and PlayerManager relationship. The way I implemented this was by first creating two abstract classes *Observer* and *Subject*. The *Subject* class works by having an array of observers and three methods. Attach() takes in an observer and adds it to the array, Detach() does the opposite and NotifyObservers() runs the notify() method in every observer stored in the array. The *Observer* class has a single method, Notify() is an overridable method that takes in a subject, which would be the subject that's calling it. PlayerManager has methods to notifyObservers on different state changes such as health, score, life. But the main logic for this implementation is when the playerManager's position is not equal to its lastposition, it notifies

the observers then sets its lastposition to its currentposition. By notifying the observers to these state changes it allows the enemyMovement to get an accurate reading on if the player has moved from their position or if the player is still alive and to update their own position accordingly. It was implemented this way so that the enemy can observe state changes in the player states to accurately move around the map but also in the future if we decide to add other features that need to read player states we can easily implement that in. This has benefited the scene in these ways as well; it allows the enemies to observe the player in an encapsulated and organized manner but also will help us in the future of the project.

Factory Implementation

Enemy Spawning

Diagram



Explanation

One of the ways that the Factory design pattern was implemented was through the EnemySpawning system. The way I implemented this was by first creating an abstract *Spawner* class that has an overridable method called *CreateEntity()* and another method called Spawn which instantiates an object set by CreateEntity() at the position of the Vector3 given in the field. The EnemySpawner script inherits from that base class. It overrides the CreateEntity method to set the object as the Enemy prefab and essentially spawns an enemy every 10 seconds randomly at one of the enemySpawn positions. It was implemented like this so that any system that needs to spawn or create something can just inherit from the Spawner class and override the CreateEntity method to include whatever object they need to create. A way this could be utilized in the future could be a script that summons multiple types of things and can reference multiple Spawner objects that all have overridden methods that create different types of objects. This has benefitted the interactive scene by creating a smoother and more robust enemy spawning system but also allows for more modularity in the future as described previously.