Assignment 1

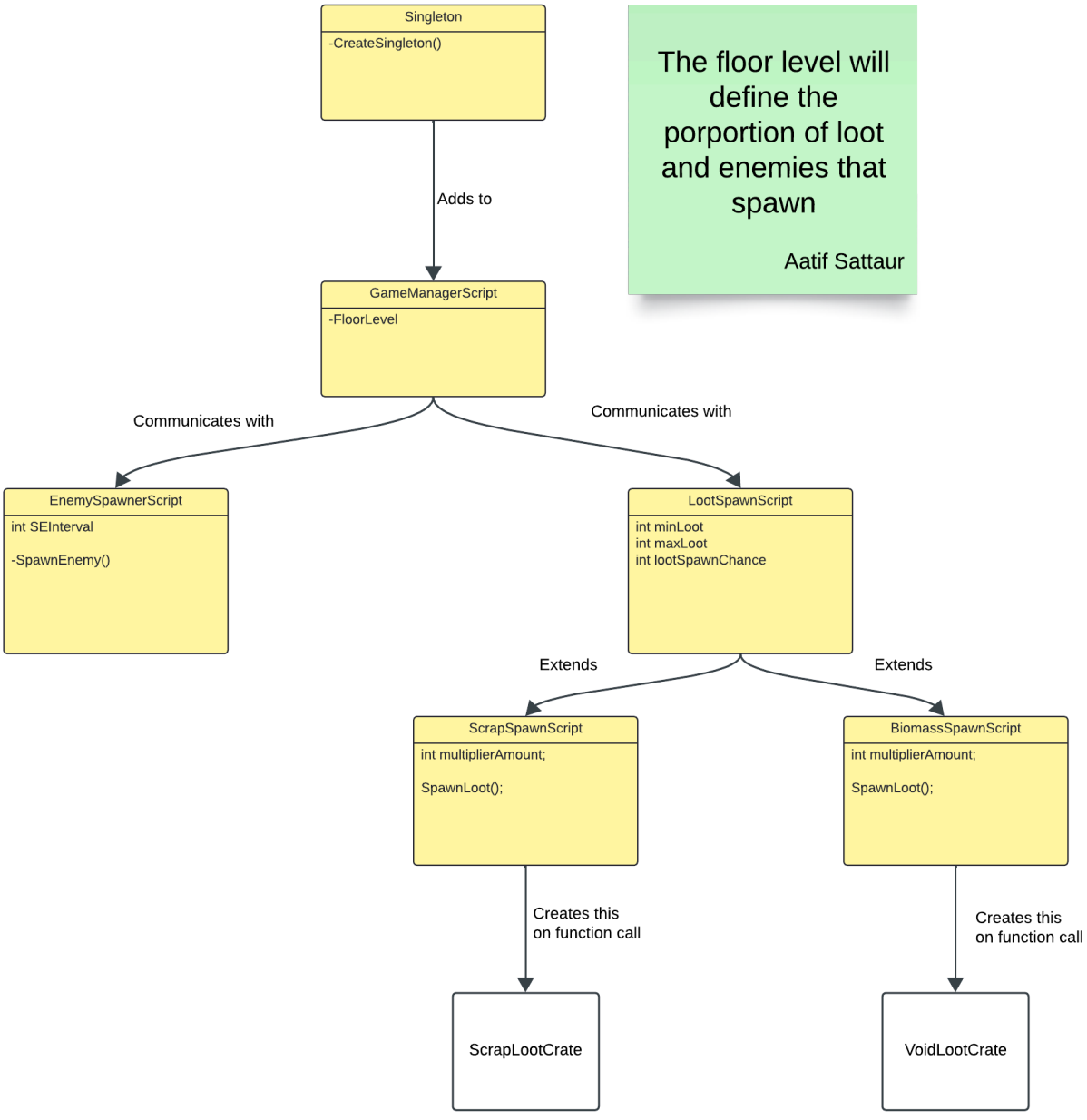
Interactive Media Scenario Information **(10/100)**

For our scenario we are building foundations for our GDW project made in Unity. The game is a horror-themed first-person shooter rogue-lite game where up to 3 players battle against another player-controlled monster over a set of procedurally generated floors. In our game, we plan to have asymmetric online multiplayer, proximity voice chat, weapons, resource collection, procedurally generated floors, tasks/puzzles, crafting, cosmetics, and monster mutations. We decided directly before the reading week that we would be switching to Unity, rather than continuing with Unreal, so we've lost a lot of time and have had to catch up. As a result, we've had to sacrifice visuals in favor of implementing game foundations due to time constraints.

Singleton **(10/100)**

GameManager **[Aatif]**

Diagram



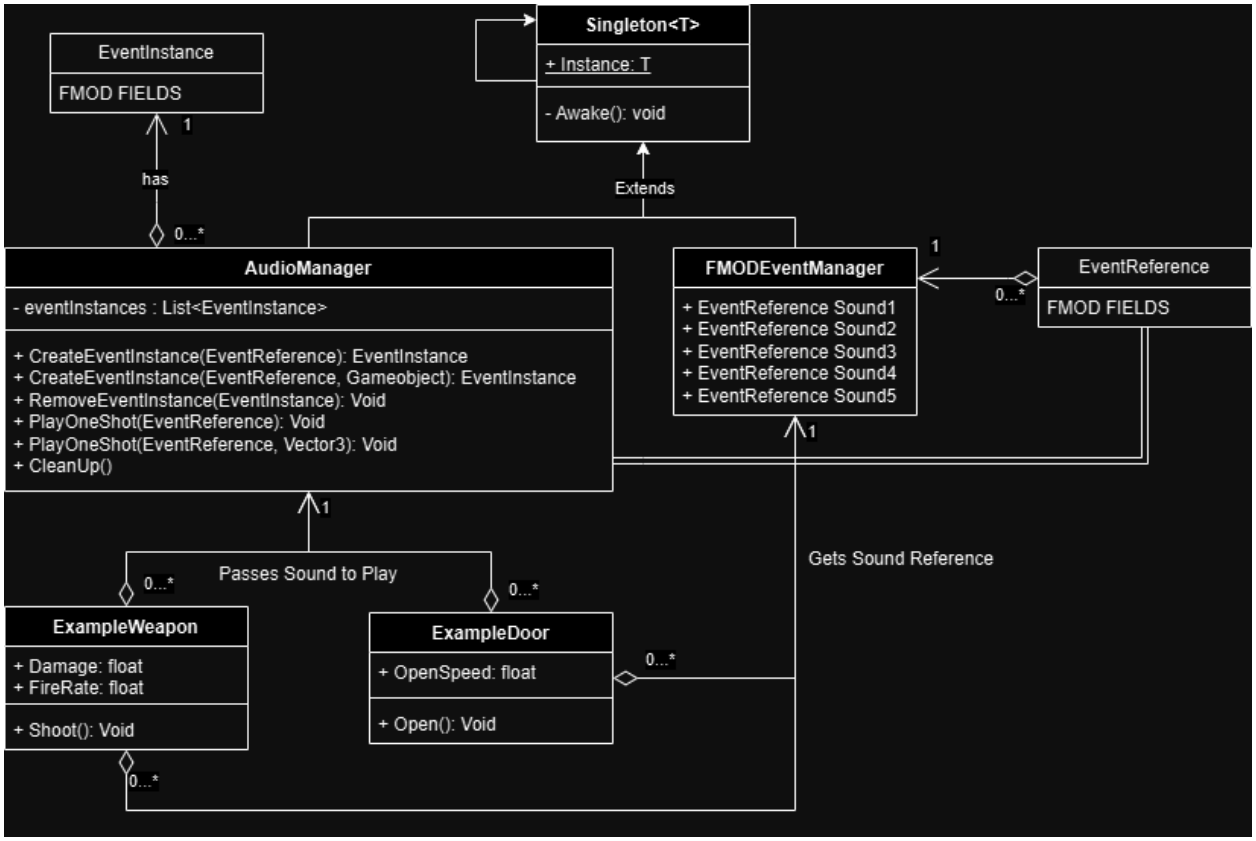
Explanation

For this singleton I assigned it to a gameobject that there will only ever be one of, the game manager, even though our game is online multiplayer there should only ever be one instance of

the game manager providing information to all players. Firstly the game manager makes sure only one instance of it exists, from there other classes access its static level counter which other classes then base their functions off of. The singleton is used here to ensure that at least 1 of these game managers exist, this is important because a lot of script function based off of the game manager.

AudioManager **[Denzil]**

Diagram



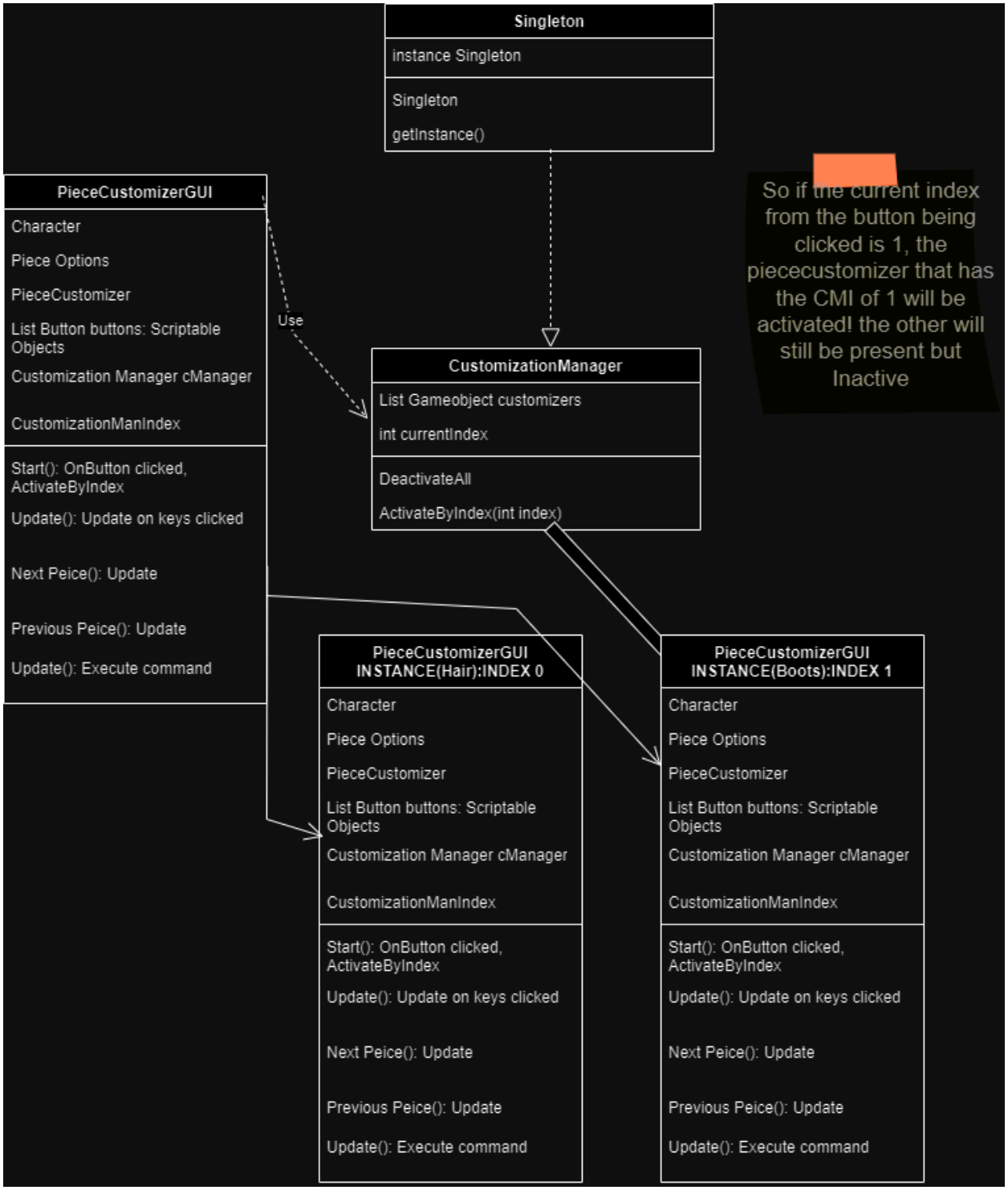
Explanation

To implement this AudioManager singleton, I first decided to create a base Singleton<T> class. I did this because all singletons usually share one base functionality. That being a static Instance variable that they set equal to themselves when they are initialized. By creating this base class, I can save time and avoid code duplication when creating new singleton classes. After defining the base singleton class, I started work on implementing the AudioManager class. The objective of an AudioManager singleton is usually to hold references to all game audio and contain methods for playing those sounds. My implementation is a little different, because it separates the holding audio portion and the sound playing methods into two different singletons, however, the end accomplishment is the same. Since this AudioManager is for my group's GDW game, and since we decided to use FMOD for audio, I had to adapt the AudioManager to account for

that, so instead of taking in audio clips, the AudioManager's methods take in event references which are references to the sounds created in FMOD Studio. To match this, the singleton tasked with holding all sound, is called the FMODEventManager and it holds all public event references. Once I had created sounds in FMOD Studio, I could assign them in the FMODEventManager so that other classes could reference them. As for the methods in the AudioManager, I essentially created simple methods related to playing audio (PlayOneShot), creating an continuous playing sound (CreateEventInstance), removing that continuous playing sound (RemoveEventInstance), and cleaning up all of the event instances when the scene is changed or reset (CleanUp). These methods allow classes to play audio in multiple different scenarios. For music, I can create an event instance, for footstep sounds, I can store a reference to the EventReference and play the audio at any position, and for a random sound, I can play it once whenever needed. This AudioManager singleton was implemented this way to allow ease of use and handle any audio situation. The interactive media experience benefits from the AudioManager through the ability to play audio. Audio is a key component of games that can help shape their experience, and without the AudioManager, setting up audio clip references on each object and having to add audio playing functionality to each of them would become unscalable, and get out of hand very fast.

CustomizerManager [Michael]

Diagram



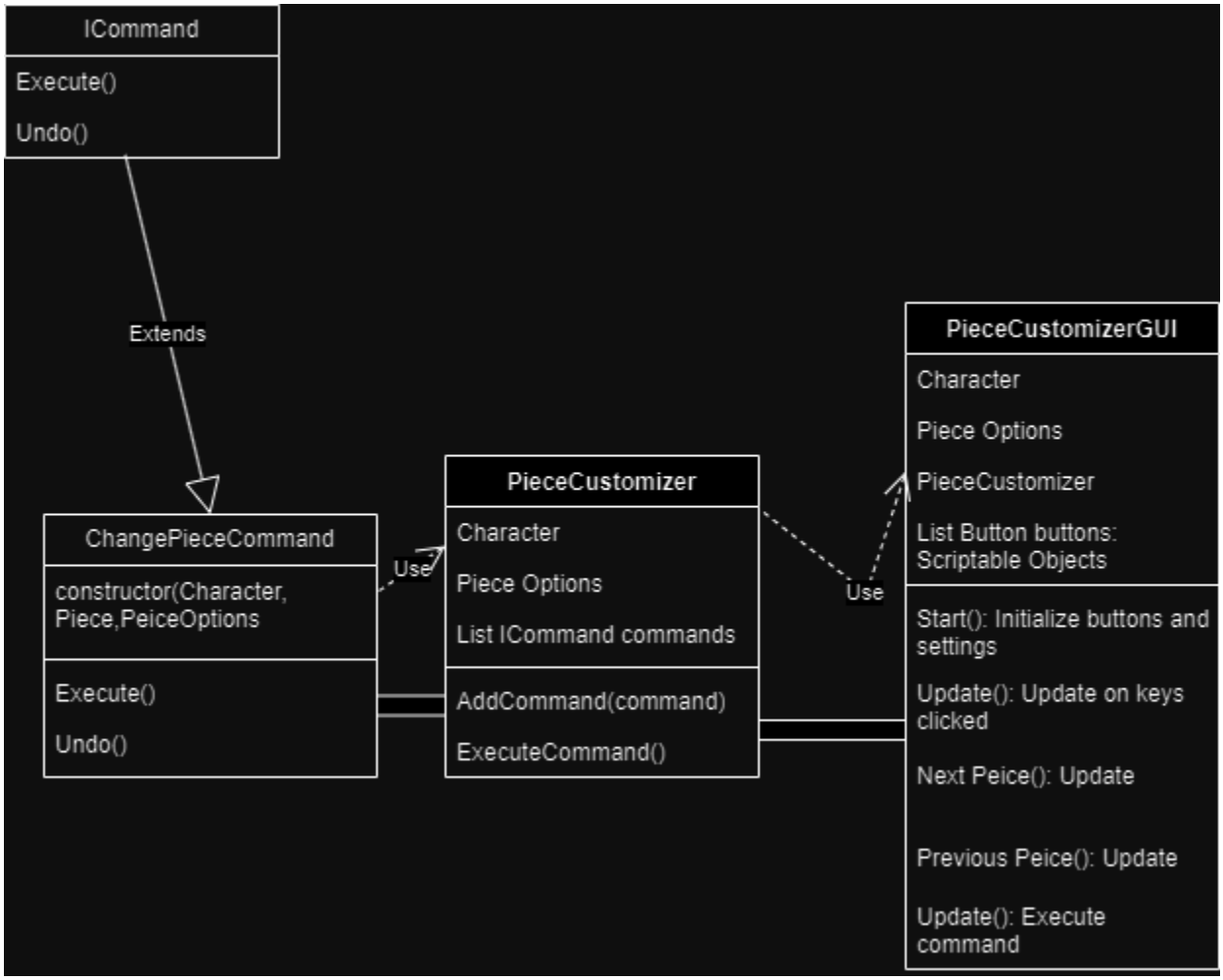
Explanation

The customizer manager essentially handles all possible customization options that could be in the game. As it stands there are only two currently the hats and the boots, but ideally there would be about five. It holds a list of all the piece customizers in the game and initially deactivates all of them except the one at the next index mentioned.I did this by using a base singleton class to define what the singleton class does then I made a list of piece customizers and those piece customizers are put in a list that checks what the current index in the main piece customizer GUI is, and then uses this to activate the main one and deactivate the rest. The piece customizer GUI then uses its own buttons to affect which one is active and which is not. I implemented it this way so that I can easily add and remove piece customizers as I decide what customizations the character may have or may not have.

Command Design Pattern **(30/100)**

Character Customization [Michael]

Diagram



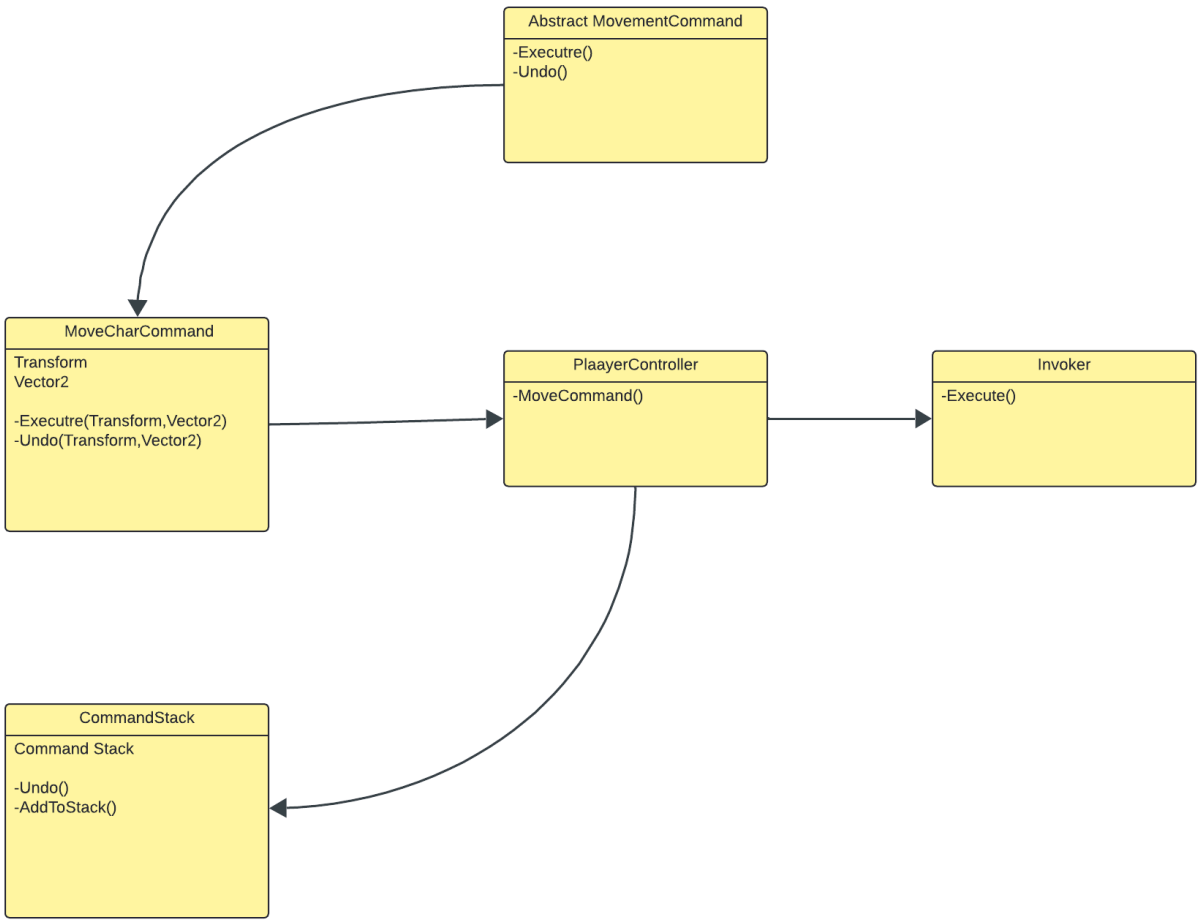
Explanation

This was the main center of the character customization system, it uses the command pattern as the base to control what pieces are set to active or not. The implementation begins with the command base which has methods to execute and undo, this is then used to make a concrete command: change piece command, which specifies how the execute will work and the constructor takes in the parameters that will be used for the execution. The Piece Customizer acts as the invoker, keeping track of the commands, and adding them to a queue to be executed. And then executing them sequentially. The piece customizer GUI serves to initialize the buttons, the list of available customization pieces, and to decide what parameters go into the command. The buttons use scriptable objects to keep track of the names of the buttons so they can be kept unique and accessed and referenced properly. The controls of the piece can be

used with either the keys or the mouse, totally independent of the commands themselves. I implemented it this way to make it simple to add and remove pieces and to choose what pieces are available for the character to choose for customization and what pieces are not available and decouples the decision making process of what piece is placed from the GUI functionality, this helps keep the responsibilities separate, making the system easier to use and extend.
For this pattern I used some AI to help give me an idea of how to arrange the code, and how it should be structured.

Task Undo (Movement)**[Aatif]**

Diagram



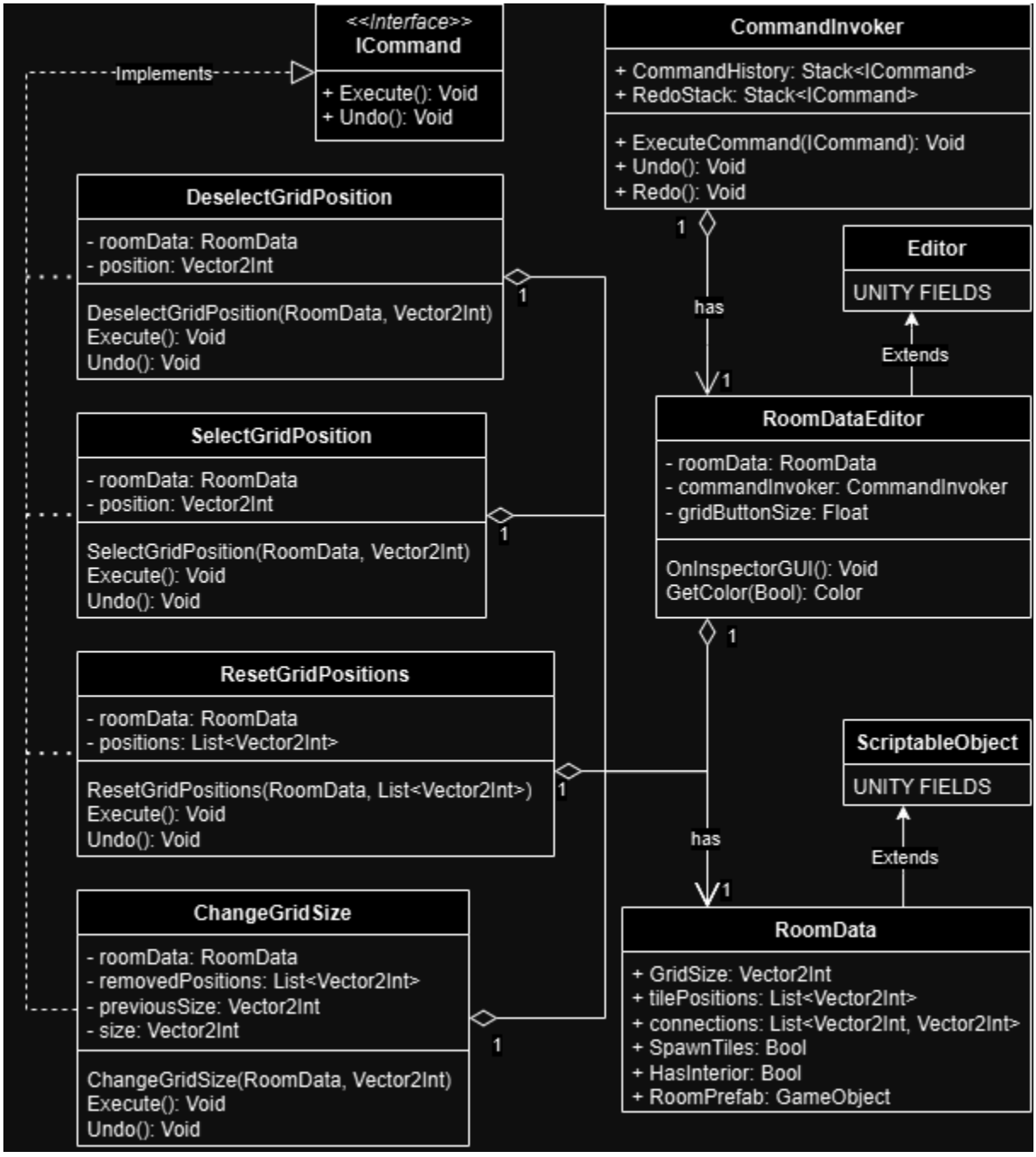
Explanation

My command pattern here works for the movement of a UI element. The goal of this is to move the UI element to certain spots within a limited amount of moves. I added the command pattern to allow for undos of the entire stack of commands. It works by having an abstract class for commands. This class has an execute and undo method, then another class defines what they do, in my case I needed it to move a 2 element. I took the transform and then took the desired

vector2 direction. To clarify the MoveCharCommand script defines the functions and what they do, the execute would move the desired object by the transform imputed and the direction vector, the undo would do the inverse. Next i had the player object create a MoveCharCommand everytime it moved, it then added these new commands to the Command Stack, the command stack would undo by popping from the stack. After that the invoker would execute the function. This is good for our game because it allows us to allow the player to undo instead of completely restarting the task if they were incorrect, because they will be in a stressful environment it would be hard to get it right quickly and accurately and that would lead to frustration which isn't good.

Editing Room Data **[Denzil]**

Diagram



Explanation

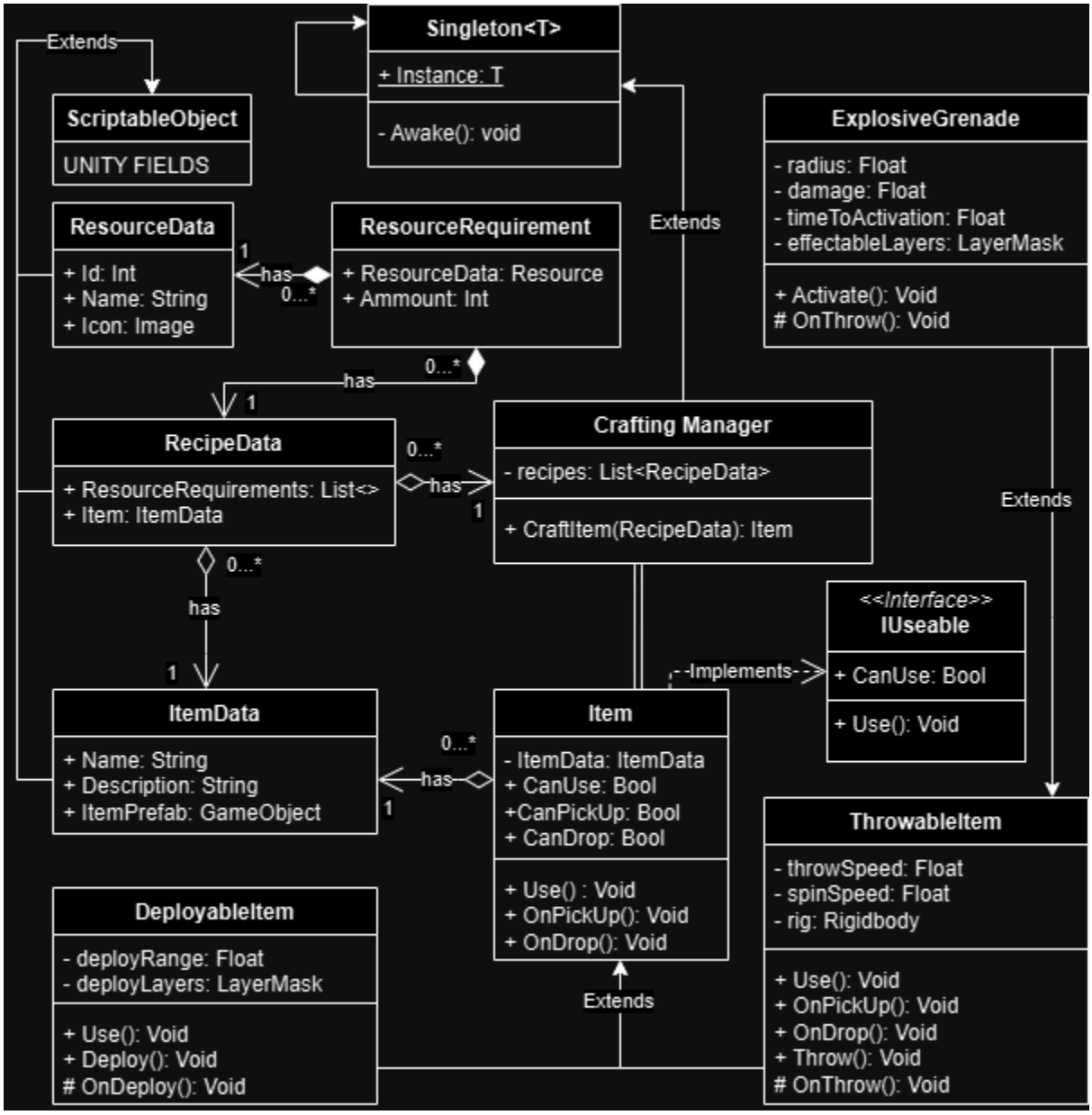
To implement this RoomData editing system, I first had to create the RoomData class. This class is responsible for holding information on which tiles should be occupied when spawning in a predefined room in the procedural map generation process. It also stores the tiles in which it

connects to other rooms or hallways, a boolean for if the GridMapManager class that handles procedural generation should spawn the occupied tiles or if they are created in the room prefab, and a bool for if the interior is already included in the room prefab. This RoomData class inherits from the ScriptableObject class to allow for easy creation, storage, and shared use. After creating the RoomData class, I realized that manually entering Vector2Int's into a list for the occupied tiles was not fun or efficient, so I decided to create an editor script that would visually display a grid of buttons that represent each tile position. Once I had the RoomData and RoomDataEditor script, one problem I noticed was that Unity doesn't automatically support undoing button presses. This is the place where I realized I could use the Command design pattern. By converting the logic normally handled by the RoomDataEditor class into commands, ie. classes that implement the ICommand interface, I could then use a CommandInvoker class to call those commands and store them inside a stack that I could go backwards through at any time. This allowed me to create an undo/redo system for 4 different actions in my RoomData editing system. This implementation of the command design pattern is valuable for the project because it allows me to easily modify any instances of the RoomData class without worry. If I accidentally changed the size of one RoomData's grid of positions to 1 by 1 from 20 by 20, normally it would erase the cut off data, however, with this command design pattern implementation, I can undo that action and save lots of pain and time.

Factory Design Pattern **(25/100)**

Crafting Items **[Denzil]**

Diagram



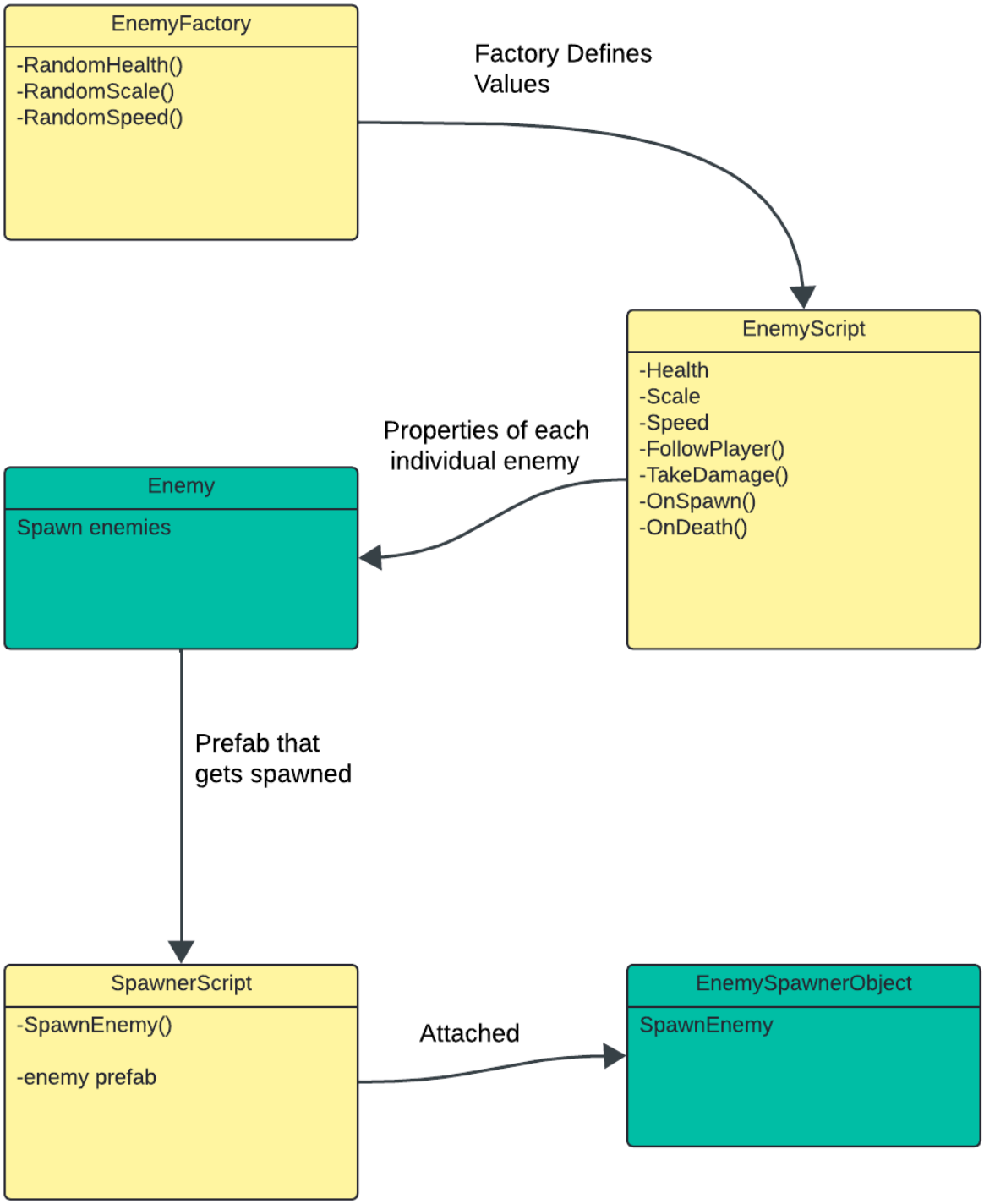
Explanation

To implement a crafting system, I first needed to create items to craft, and, to craft items, I needed stuff to craft those items with. To accomplish this, I first created the RecipeData class which inherits from the ScriptableObject class and holds a list of resource requirements as well as the item it crafts. Currently these resource requirements are unused, however they will

eventually be used to verify a player has enough resources to craft an item, and how many resources of each type to remove from their inventory. Once I had the RecipeData class created, I could move onto the things that they craft. I first created the ItemData class that would hold all data shared between all items. Instances of this ItemData class are held by the Item class which serves as the base class for more complex items. The Item class specifies shared methods between items, and it implements the IUseable interface which defines a Use() method. This Use() method will be used to facilitate the factory design pattern, so that the Item base class can be returned when crafting an item of any type. The CraftingManager singleton handles the crafting requests and stores a list of all recipe instances. The crafting mechanic was implemented using the factory design pattern so that items of any type could be crafted, added to the player's hotbar, and then used regardless of the item's type. Using the factory design pattern allows us to keep the item crafting system scalable and organized, as well as support complex item creation. To showcase this functionality, I created two child classes of the Item class, DeployableItem and ThrowableItem. As of now, only ThrowableItem has a child, which is called Explosive grenade, however more items will be added as the project progresses. The explosive grenade class is able to define properties specific to it, such as its damage and radius, as well as have complex behavior like an Activate() method which causes it to explode.

Enemy Creation **[Aatif]**

Diagram

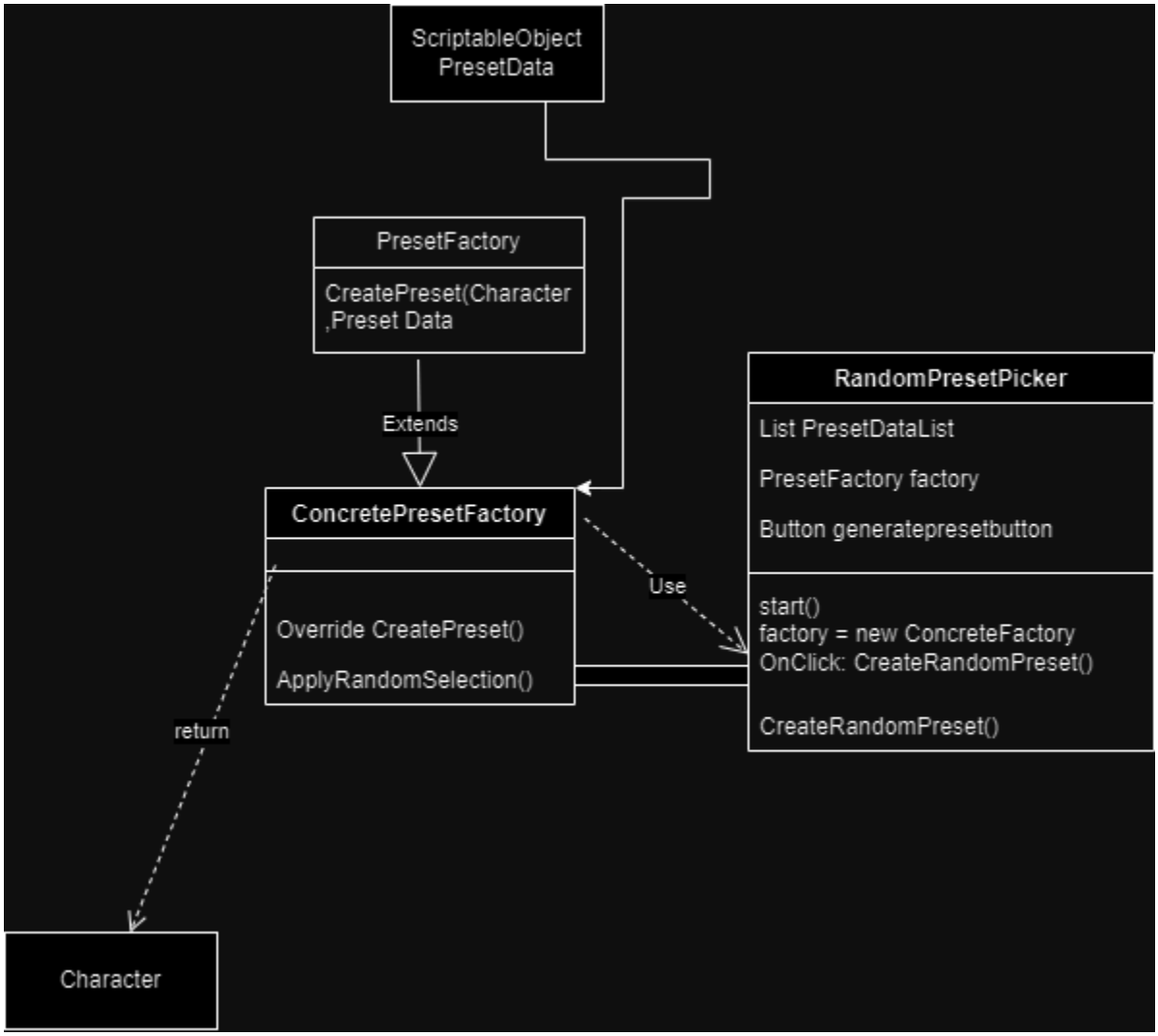


Explanation

Here the factory pattern used to define concrete values for the enemy class. We have our enemy factory that defines values for the enemy script, the enemy script is the attached to an enemy prefabs and defines its values, the enemy prefab is then spawned by the spawner script, from then it is then used in the enemy spawner object. Now without this factory object defining the enemy values the enemy script will either have all static non changing values or it would need to be changed within its own script. The pros of the factory pattern in this use case is we can now have our enemy script hold all of its important information, information such as how its future AI will act. Without the factory pattern script the enemy will need to define its values, making everything uniform and not modular. Having the factory pattern also helps spawn different types of enemies and set values, for our game we only planned on having 1 simple “mini” enemy so we don't plan on using the factory pattern for that but by using it the way we have implemented. To summarize the factory pattern is used to create different enemies each time it spawns, this helps us if we want to add

Preset Character Spawner

Diagram



Explanation

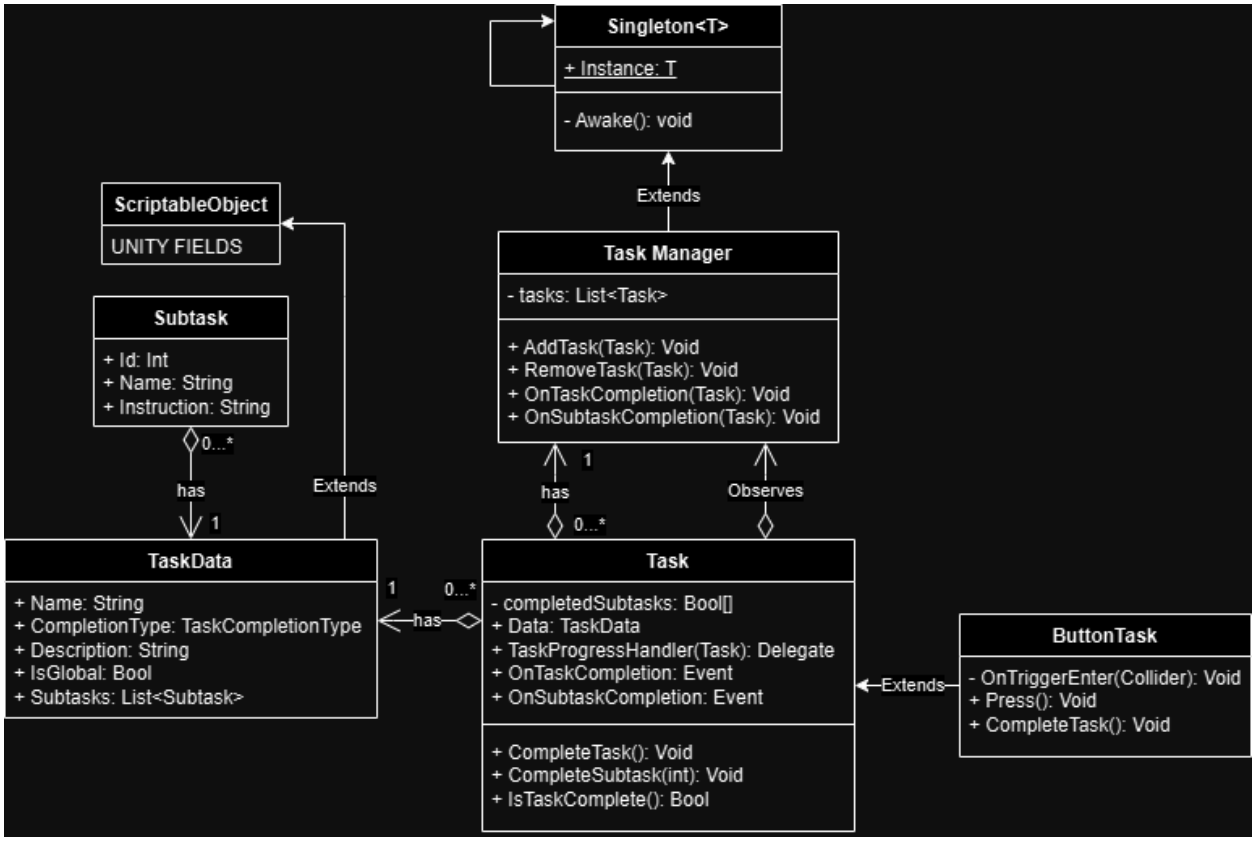
This system was implemented using a scriptable object to hold information about pre-created character options, and randomizes features and the options to create a new preset character. I implemented this by creating an abstract preset factory, and this extends into the concrete preset factory that uses the create preset method to create the preset using the data from the scriptable objects, and randomizes the parameters around to return the character. The interaction of this is done with the random preset picker script which has been used to create a random preset character for the player to use rather than creating their own. I implemented it this way to make it easier to manage and develop this feature so that it can be used for character creation in the place of total customization. This allows for simple and efficient but also an interesting way for the creation system to work. For this pattern, I used Artificial

Intelligence to help guide me through how to create a proper factory and to give me an idea of how to use it. I built upon this system to create my final code.

Observer Design Pattern **(15/100)**

Task Completions **[Denzil]**

Diagram



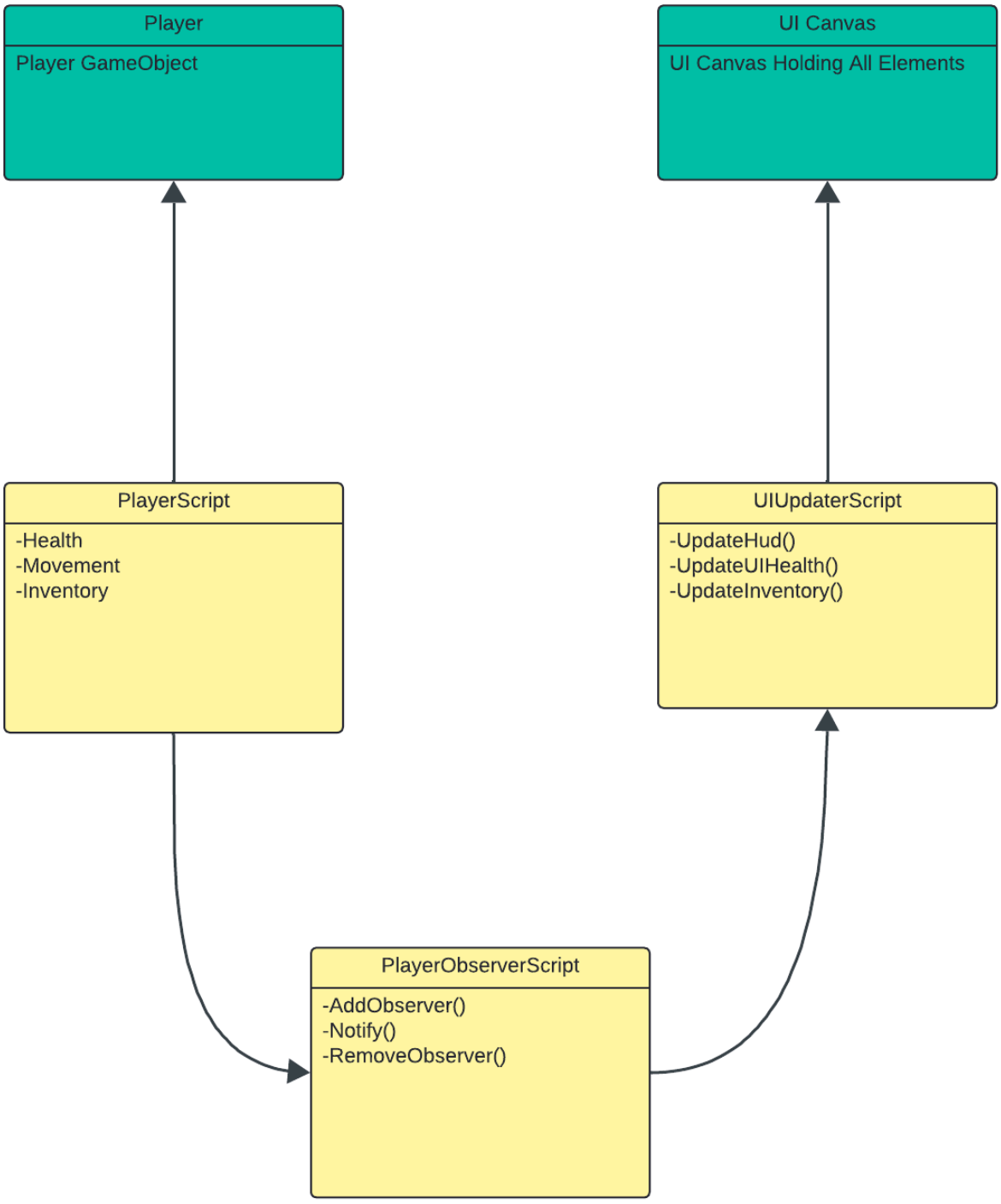
Explanation

To implement task completions using the observer design pattern, I first decided to create a TaskManager singleton. This task manager is responsible for holding a list of all tasks in the scene, and running any shared logic that happens when tasks are completed. After setting up the TaskManager, I created the Task and TaskData classes. The TaskData class holds information related to the Task class instance it is contained in, such as subtasks, the name of the task, and how the completion should be measured (Percentage, Amount, or Binary). The Task class on the other hand is the base class for more complex tasks. Inside this Task class, there are two events and a delegate. I decided to use Unity's built in observer pattern through the use of events and delegates instead of implementing an IObserver/ISubject interface myself.

I did this to give me more customization over which parameters are passed, take advantage of Unity's optimized code, and maintain the same format over other instances of the observer pattern that I will have to implement. The OnSubtaskCompletion event is invoked when any of the task's subtasks are completed and the OnTaskCompletion event is invoked when all of the subtasks have been completed. Moving on from the Task class, the actual observer implementation is done through the TaskManager and Task classes. When a task is added to the list of tasks in the TaskManager, it attaches itself to the added task and listens for the two events I mentioned earlier. The Task class does not call the OnTaskCompletion and OnSubtaskCompletion methods inside of the TaskManager, instead, those are called through the use of the observer pattern. I created an example task named ButtonTask, which showcases the working observer when the player walks into its trigger collider. Once triggered, the button task completes its only subtask, and in turn invoking the OnTaskCompletion event, which then triggers a sound to be played inside of the TaskManager's OnTaskCompletion method.

UI/HUD (Player) Updates **[Aatif]**

Diagram

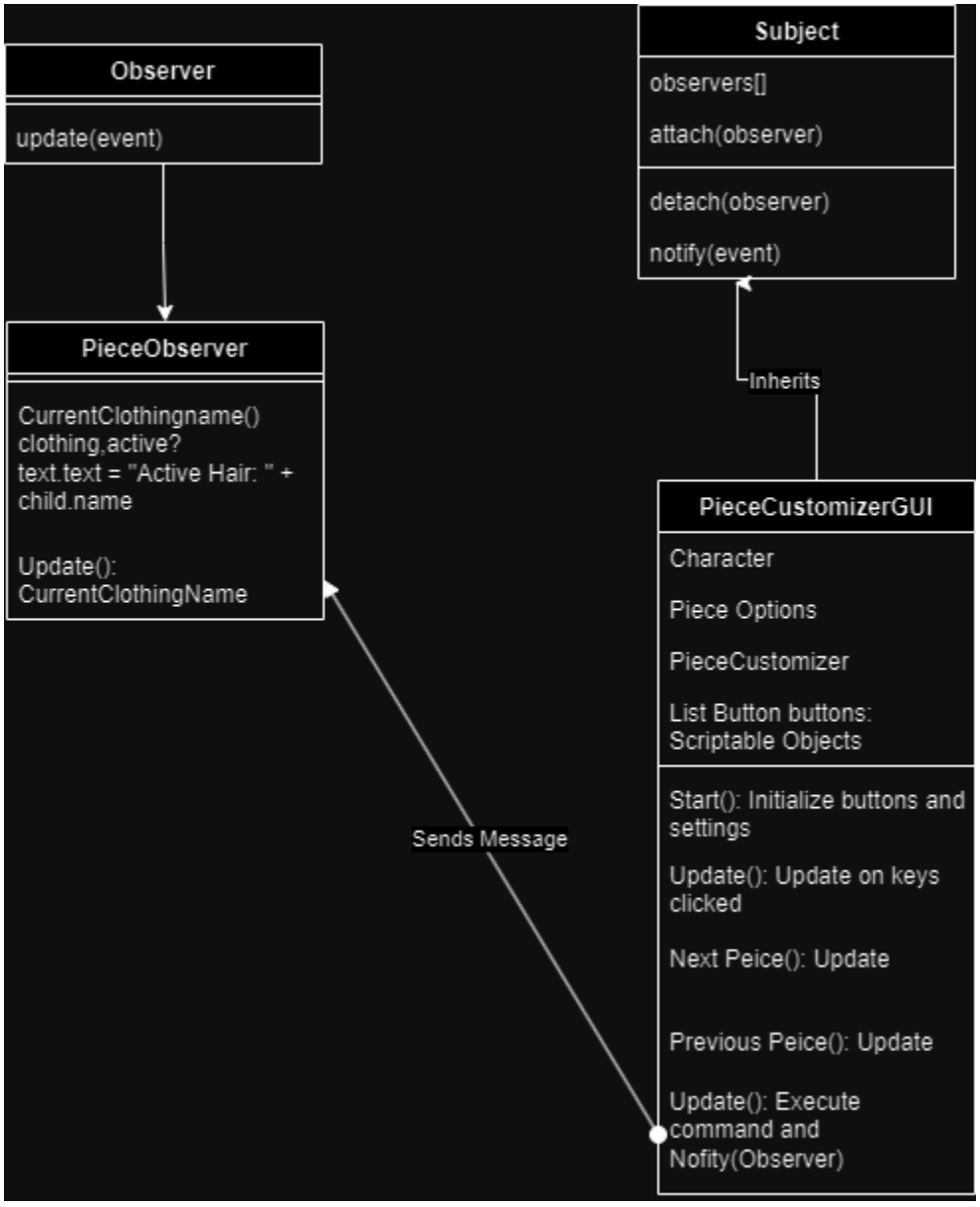


Explanation

Here I have an observer that observes the player, it takes in the relevant information relevant to the player, health and the player's inventory. It then conveys this information to the UI and HUD. If there was no observer the player would need to speak to the UI directly, this would end up creating too many circular dependencies. The observer pattern helps keep the player separate from the UI and removes circular dependencies. This also helps with extension of both of these classes, if the player had some elements or variables that did not affect the UI, damage per bullet for example, the UI wont need to know, by adding this observer the UI knows only what it needs to and the player does not need to know about this at all. In the future of our game we also plan on using the observer to give information to our server (HostPC) to handle the calculations. This is a great implementation for the observer because each PC will have an observer for each individual player, this means all the observers are the same and all talking to the server. With one script used for communication we save our player from having dependencies with the UI and the server and the observer is extremely reusable.

UI Updater[Michael]

Diagram



Explanation

The observer pattern I implemented simply uses unity's built in delegation to help me identify and change the display of what feature is currently active on the character customization page. It was implemented by first defining a delegation and telling it to invoke this delegation upon the piece being updated the PieceObserver then receives this message and then updates what the current clothing name is being displayed on the UI. For the sake of explanation I used the subject and observers to explain the inheritance, but Implementing it in the code did not work as planned so I simply used unity's delegation system to enact it in the code. I used the observer

system to separate the functionality of the UI display from the customization itself and to pick and choose what gets used and notified. This helps make the system easier to use and extend when needed, making it less difficult for the player to know which option is selected.

Video Report **(10/100)**

https://youtu.be/W4lFgs5P2B8
Video report with each members contribution and their explanation of each of their design patterns

---