Project: *Soul Keeper*

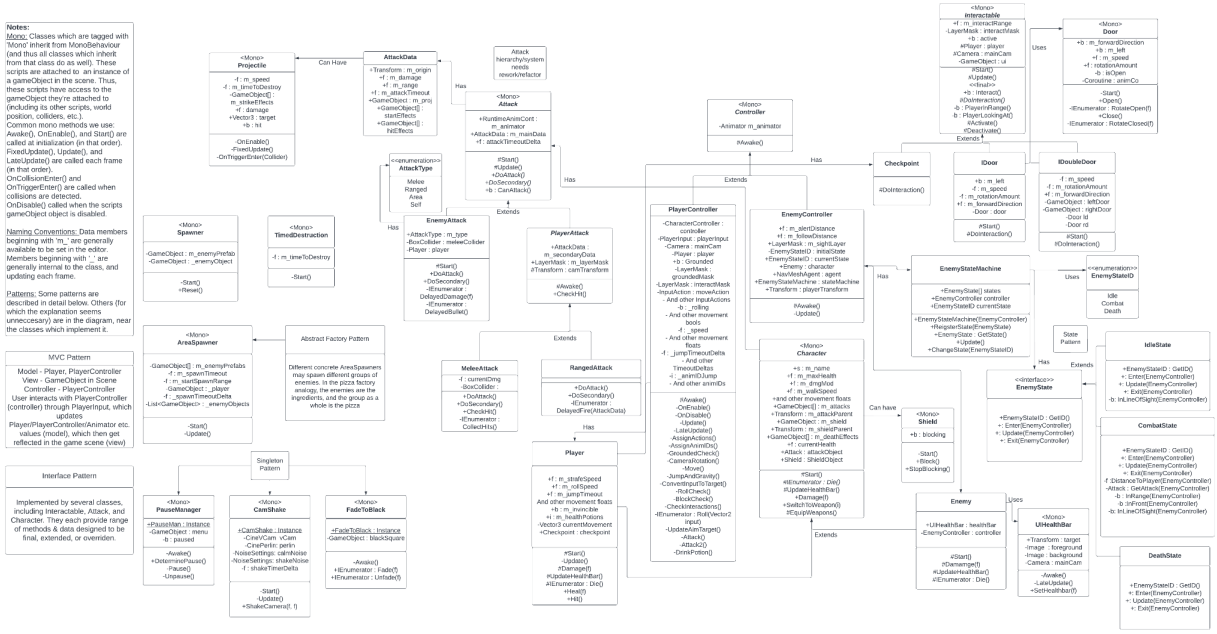
Team Members: Vince Curran, Logan Park, Kevin Vo

Final State of System:

Everything we originally planned to implement has been implemented, including some things we were unsure of getting to. Specifically, the game features one dungeon to fight through with a diversity of enemies as well as a boss, two different classes for the player to play as, the ability to save and load game save files, checkpoints, dialogue, effects, and most everything else you would expect to find in an RPG (besides an inventory system).

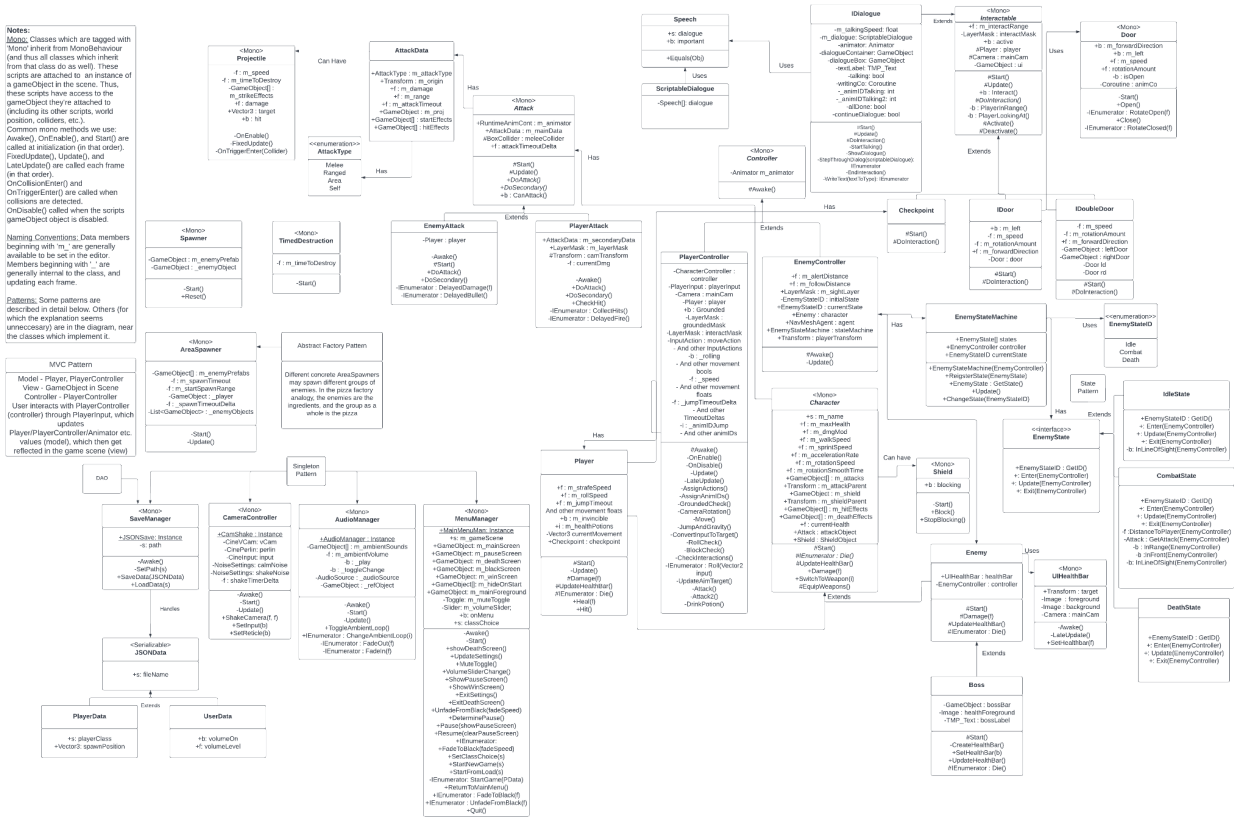
Class Comparison (PDFs also included as standalones in repo):

Project 5 Class Diagram:



Project 5 Class Diagram.pdf

Project 7 Class Diagram



Notable Changes:
Some notable differences between the two diagrams include the differences in the Attack hierarchy, the addition of more ‘Managers’, the Dialogue system, and creating a seperate class for Bosses. Much of this is discussed later.

Pattern Use:
MVC:
Model - Player, PlayerController
View - GameObject in Scene
Controller - PlayerController
User interacts with PlayerController (controller) through PlayerInput, which updates Player/PlayerController/Animator etc. values (model), which then get reflected in the game scene (view).

State:
A state machine (EnemyStateMachine, EnemyState) is being used to control the behavior of enemies.

Singleton:
Used by several different classes which we want to ensure only one of throughout the lifetime of the application. Most notably these are the GameManagers (PauseManager, MainMenuManager, FadeToBlack, etc.).

Abstract Factory:
Different concrete ‘AreaSpawners’ may spawn different groups of enemies. In the pizza factory analogy, the enemies are the ingredients, and the group as a whole is the pizza.

DAO:
The ‘JSONSave’ class provides an interface for interacting with the game's persistent data.It provides simple ‘save’ and ‘load’ operations so other classes do not need to know the details.

Builder:
The Builder pattern is used by the ‘IDoubleDoor’ class to set up the two ‘Doors’ which make up the ‘DoubleDoor’.

Third-Party vs Original Code:
Monobehaviour: Classes which are tagged with 'Mono' inherit from MonoBehaviour (and thus all classes which inherit from that class do as well). These scripts are attached to an instance of a gameObject in the scene. Thus, these scripts have access to the gameObject they're attached to (including its other scripts, world position, colliders, etc.).
Common mono methods:
Awake(), OnEnable(), and Start() are called at initialization (in that order).
FixedUpdate(), Update(), and LateUpdate() are called each frame (in that order).
OnCollisionEnter() and OnTriggerEnter() are called when collisions are detected.
OnEnable() and OnDisable() are called when the script's gameObject is enabled/disabled.
Note: These are all virtual so do nothing if not implemented

Pathfinding:
To simplify writing the AI for our enemies, we use Unity’s NavMesh and NavMeshAgent to handle the ‘heavy lifting’ of pathfinding. Our EnemyStateMachine utilizes this ‘Agent’ by updating its target destination, speed, and stopping distance to create different behaviors (get close to player, search for player, idle, etc.).
https://docs.unity3d.com/ScriptReference/UnityEngine.AIModule.html

Camera:
To simplify the control of the main game camera, we use Unity’s Cinemachine to handle the ‘heavy lifting’ of tracking the player and handling collisions. We simply set a target (the player) for the camera to follow as well as some parameters and get a pretty good camera implementation. We extend the functionality some by providing a CameraController which interfaces with the Cinemachine components. The Controller allows us to shake the screen

when the Players hit by applying different “NoiseSettings”, disable the camera input while in game, and some other things.
https://docs.unity3d.com/Packages/com.unity.cinemachine@2.2/manual/index.html

Animation:
Animation is handled by Unity’s Animation Controller. This is basically a state machine where you define all the states, transitions, and parameters in the editor. The transitions are governed by the parameters, which we set in our code (mostly in Player/Enemy Controller).
https://docs.unity3d.com/Manual/class-AnimatorController.html

UI:
Unity UI handles connecting UI components (like buttons) to methods which we write, we can just “add methods” to button clicks. Sliders automatically call OnValueChanged() when modified, which we overwrite to give functionality.
https://docs.unity3d.com/Packages/com.unity.ugui@1.0/api/UnityEngine.UI.html

JSON:
We use Unity’s JSONUtility to convert Serializable objects into JSON strings. We then write the JSON strings to files using System.IO’s StreamWriter.
https://docs.unity3d.com/ScriptReference/JsonUtility.html

Player Input:
We use Unity’s PlayerInput system to capture the user's input. This makes it easy to assign ‘actions’ to button presses. We then use delegates as well as some of the provided methods (such as WasPressedThisFrame()) to see when input happens, then we handle it with appropriate response.

Math/Physics:
Unity provides a bunch of math/physics related functionality which we use extensively. This mostly includes Vector/Quaternion math and Colliders. Vectors/Quaternions are used for finding positions, distances, rotations, etc, and all the common operations are given (such as Vector.Dot). Physics.Raycasts are used to find the nearest Collider of a certain type/object, and Colliders can call our methods when interacting with other Colliders (which they do, such as the case for when a projectile hits something and gets deleted due to it).
https://docs.unity3d.com/ScriptReference/UnityEngine.CoreModule.html
https://docs.unity3d.com/ScriptReference/UnityEngine.PhysicsModule.html

Tutorials:
Including channel names in lieu of video names due to sheer amount of videos watched:
https://www.youtube.com/c/samyam (mostly getting input & player controller, some UI)
https://www.youtube.com/c/SpeedTutor (mostly UI/menu stuff)
https://www.youtube.com/c/TKaupenjoe (save & load)
https://www.youtube.com/c/MetalStormGames (UI, animation events)
https://www.youtube.com/c/Brackeys (world space UI, combat)

https://www.youtube.com/c/unity (many basic Unity stuffs)
https://www.youtube.com/user/ICampEasts (mostly attack related)
https://www.youtube.com/c/CodeMonkeyUnity (UI, Cinemachine)
https://www.youtube.com/c/WorldofzeroDevelopment (math, physics)
https://www.youtube.com/c/TheKiwiCoder (mostly enemy AI and attacks, some player control)
https://www.youtube.com/c/Roundbeargames (attacks)

Videos not from above channels:
https://www.youtube.com/watch?v=cPltQK5LlGE&t=395s (doors)
https://www.youtube.com/watch?v=3ff_EwE1IDo&t=863s (interactions)
https://www.youtube.com/watch?v=Os7uf-wiU8o (interactions)
https://www.youtube.com/watch?v=ez0ofMVUXYQ&t=340s (rolling)
https://www.youtube.com/watch?v=YE8w-Xycvhw&t=366s (projectiles)
https://www.youtube.com/watch?v=46hSbkTuHmc (collisions)
https://www.youtube.com/watch?v=uZlSOAP76-A (movement)

Assets:
All assets in the game were acquired for free from various websites such as:
https://assetstore.unity.com/
https://www.mixamo.com/#/
https://www.zapsplat.com/
https://www.kenney.nl/

This includes all art, ranging from models, textures, animations, particles, etc. We still had to convert everything to our format, create the controllers for the animations, and plenty of other stuff, as none of it is really ‘game-ready’.

Misc:
The source code for Valheim was used as inspiration/guidance for some things. Examples include the improvements to the Attack hierarchy (mentioned in detail later), and how to use Unity’s Animation Event system.
https://www.valheimgame.com/

OOAD Process:
Attacks: Originally, our hierarchy leading down from ‘Attack’ was very messy, including ‘EnemyAttack’, ‘PlayerAttack’, then such things as ‘RangedPlayerAttack’, ‘MeleePlayerAttack’, etc. leading from those. This made creating actual Attack objects complicated because you would have to attach the specific subclass to your object. We remedied this by having the AttackType (Melee, Ranged, etc) be part of an AttackData object. Then deciding what attack algorithm to use based on that. This allows us to just have EnemyAttack and PlayerAttack as subclasses of Attack. This idea is inspired from Valheims code, where they have a similar AttackData class to hold data for Attacks. The difference is, all their attacks use the same algorithm whether made by the player or an enemy, so they need no subclasses. This would be

ideal for us, so the same attack objects could be used for Enemies as the Player, but we haven't been able to write a general enough algorithm (that led to a decent user experience) to handle both cases with the same code.

Managers: Originally, our ‘MenuManager’ class was split into several classes, including PauseManager, MainMenuManager, SettingsManager, etc; which each handled a specific menu. This led to some inconsistencies in class design (like whose responsibility is it to switch between 2 of those screens, or to open one from ‘any’ place?), as well as challenges when developing, because you would have to think about exactly which one to reference when you needed something done. To remedy this, we combined them all into one ‘MenuManager’ class, which handles the functionality for everything that can happen from Menu UIs. This allowed us to have one central interface for calling the methods, be able to reuse code duplicated between classes, and combine methods that were always called in succession.

Character Controllers: Originally, we were unsure of how to handle all of the input from the Player. We were thinking of doing something similar to the Command pattern, which we think would have worked, but not been as clean. Eventually we settled on MVC, because it would allow us to reuse the same base Model for enemies as the player character, we just needed to switch out the Controller.