Project 7

1)

Project Name: Finding Felix

Team Members: Kyle Ma and Owen Smith

2)

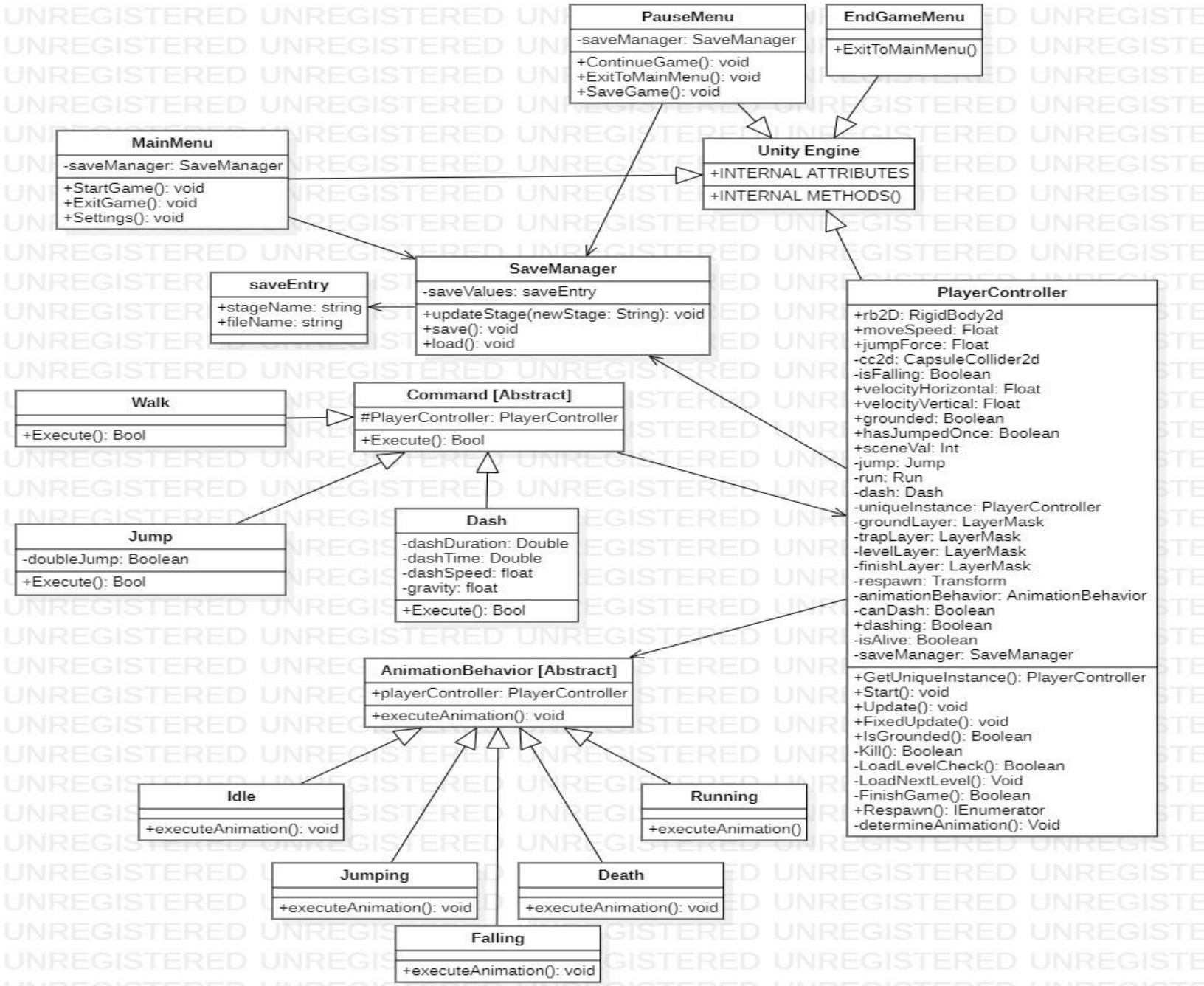
Final State of System Statement:

As discussed in our project 6 submission, the structure of our code has changed quite drastically from our original plans in project 5. At the time of our project 5 submission, we were both very new to Unity, and as we developed a better understanding of the engine's tools and structure, we refactored our ideas quite a bit. Nonetheless, our original feature list from the project 5 submission has been completed entirely. There is a functional save system, solid player movement, original pixel artwork, thoughtful level design, and a simple UI in the menus. The largest change in design approach between project 6 and now was a ground-up animation manager that employs the Strategy pattern. Unity has a built-in graph-based animation manager, but the state-transition model used by default was awkward, and so we opted to write our own. Additionally, we decided to scrap the idea of a three-save load menu and instead only allow one persistent save file at a time given the games short length. Although we implemented the features we wanted, the game could definitely be expanded with more levels, music, and interactive

event-based audio, but time-constraints forced us to limit some of these “bell and whistle” ambitions.

3)

Final UML Diagram:



Pattern Documentation:

Strategy:

- The Strategy pattern is used in AnimationBehavior, which is the class that acts as a replacement for Unity's built-in animation manager. The PlayerController has an AnimationBehavior object, which is updated to one of its inherited classes as game conditions prompt different animations (negative velocity and being off the ground would call for an instantiation of Falling, for instance).

Singleton:

- Singleton is employed in the PlayerController. Because PlayerController inherits MonoBehavior from the Unity engine, the "new" keyword is generally prohibited. Despite this, we can use component creation to instantiate the script as a Singleton in a given scene and have any other script used in that scene access it. Moreover, GetUniqueInstance() prevents duplication of player-controlled objects. It is somewhat unconventional when compared to simpler implementations with Java objects, but is functionally equivalent.

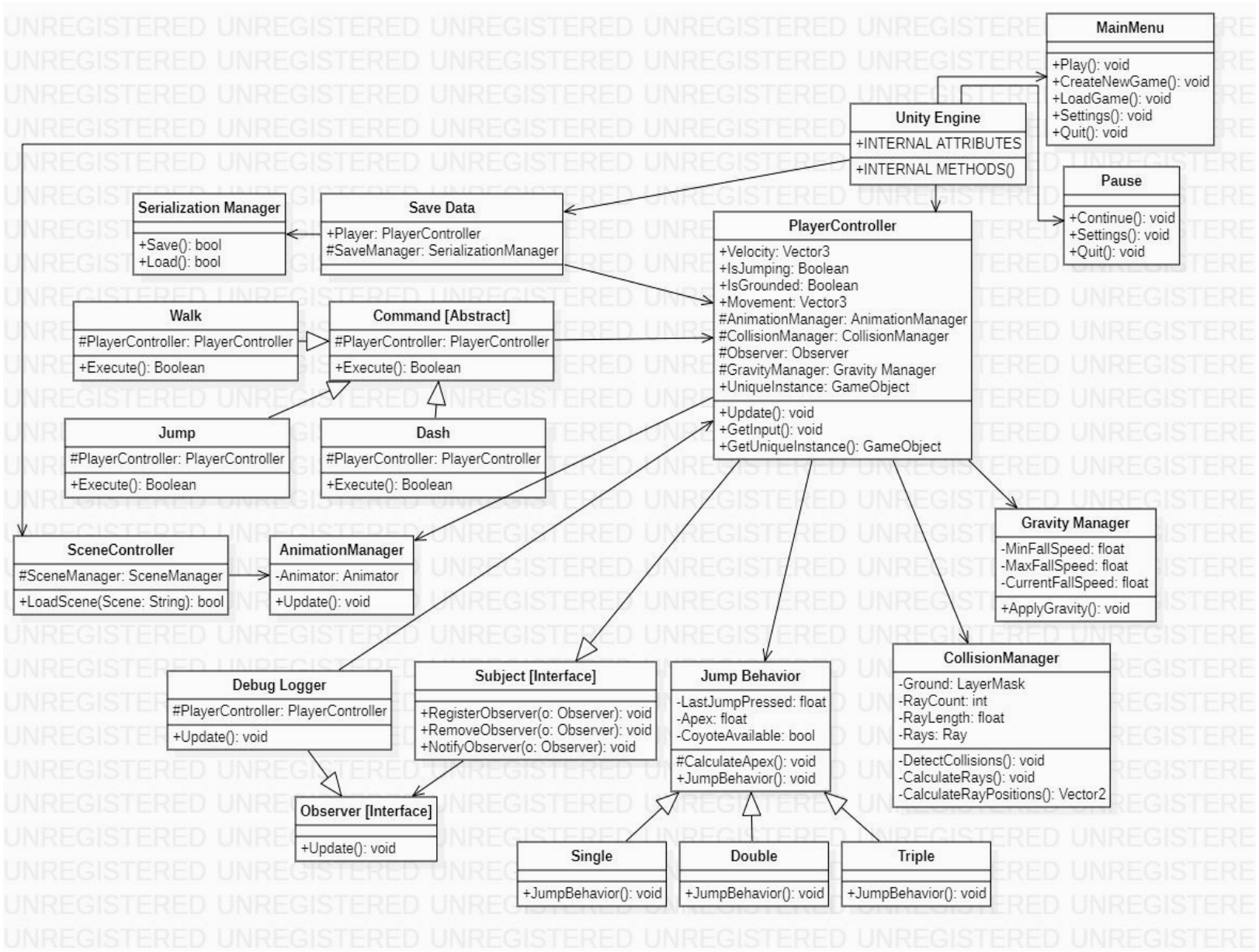
Memento:

- The Memento Pattern is employed for our save and load system. In this case our Save Manager acts as the memento, the PlayerController acts as the Originator, and the user acts through the pause menu and main menu as the Caretakers. The player is able to pause the game and save their current level, which then goes to the SaveManager which saves to an XML File. Upon load, where the user chooses to load the game, the MainMenu class will go the SaveManager which will load from the same xml file in order to bring the user back to the stage they were previously on.

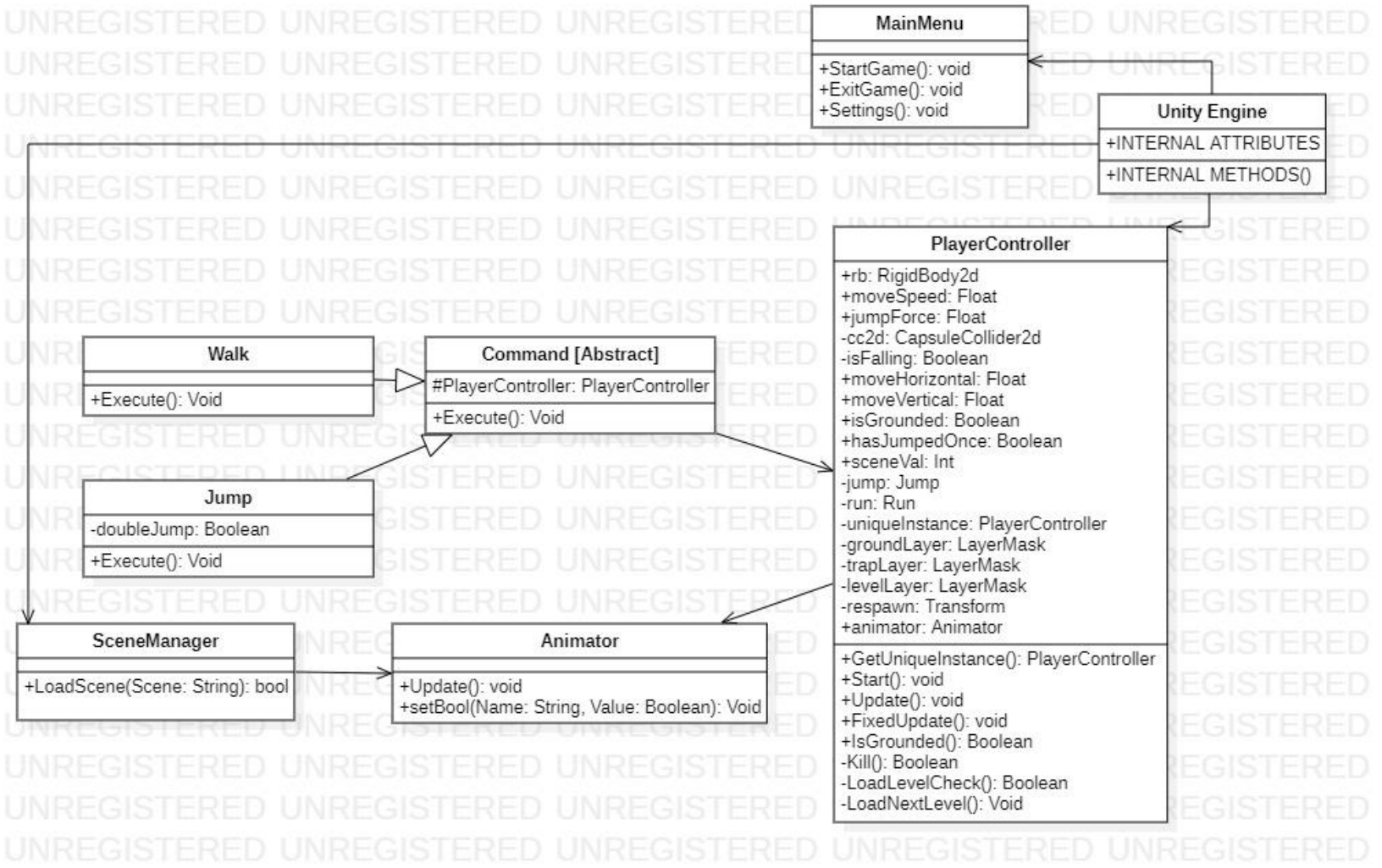
Command:

- The command pattern is used for all of the project's movement mechanics. We have a command abstract class, with 3 different concrete inherited classes: Run, Jump, and Dash. Each of these classes have a player controller as an attribute. And each of these inherit the execute method of the parent class in order to deal with the functionality. Run's execute makes the player move in whichever direction they choose while flipping the character. Jump deals with the player's ability to jump and more importantly their ability to double jump. Dash deals with the player's ability to dash, or move quickly in a short burst of speed in one direction.

Project 5 UML Diagram:



Project 6 UML Diagram:



Summary of Changes:

As far as design goes, there were really two major pattern changes between project 5 and our current version. Because we felt that the simplicity of the jump behavior did not warrant multiple algorithms, we opted to replace the Strategy pattern for JumpBehavior in project 5 with a memento pattern in the final product. Additionally, as we debugged our project, we found that an Observer pattern would have been an unnecessary overhead for this particular project (would have been a glorified logging tool), so we opted to employ a Strategy pattern in our own animation manager system. This seemed to be a better allocation of our time while still meeting

our pattern requirements for the project. Other changes in the system can largely be attributed to our improved understanding of Unity. As evident in the project 5 UML diagram, we originally anticipated having to write much of our own collision and physics management from scratch by applying conditional vectors and raycasting. Because Unity has a great modular 2D physics engine, we ultimately did not have to make these systems from scratch, but instead adjusted the proprietary Unity system to our liking. The absence of a scene manager in the final product generates a similar explanation. The result is a much simpler class structure with all the same functionality.

4)

Since both of us are new to unity and C#, we both needed to follow some tutorials for the things we did. A lot of the base work that we laid down at the start was inspired by some tutorials that we found. Much of the beginner movement mechanics were inspired from: https://www.youtube.com/watch?v=TcranVQUQ5U&t=1s.

However, as the project moved forward, we had to modify the code completely in order to fit our specific needs. Some of the base structure is the same, but most of the code is now completely different. However, the mechanisms for serialization were taken directly from a tutorial, as there are only a few ways to save data in Unity - https://www.youtube.com/watch?v=6vl1IYMpwVQ.

The raycasting for collision detection in the player controller was also taken from a youtube tutorial - https://www.youtube.com/watch?v=c3iEl5AwUF8.

Nearly everything else we made was mostly done entirely from scratch. The animation manager, player movement mechanics, level system, menus, etc. are original.

5)

1)

Our primary issue with this project was learning to use unity. Neither of us had any prior experience with game development in general and especially unity. Learning how to create a project, set it up, and lay down some basic functional elements was a difficult task but ultimately rewarding. Because of this, while we were planning the design for our project, we were not sure what was feasible and what was not. We did not know what we could incorporate with Unity's built in functions and what we would have to create on our own.

2)

We also had issues with the code volume of this project. Originally in project 5, we had planned to create our own custom collision manager, animation manager, and gravity/physics engine. We quickly realized that this was not feasible and instead opted to use much of Unity's built in functions. We used many of unity's built in animation tools and physics engine as this was much simpler, and getting a grasp on Unity and c# did not allow us to do this.

3)

During the design process, we had to decide where to incorporate our 4 OO Designs. Since we decided early on that we wanted to create a 2D platformer, we decided immediately on command and strategy for the movements. We also decided to incorporate a singleton and observer for debugging purposes as these both seemed useful. As the project progressed, we would often switch to different design patterns that seemed more feasible. Here at the end of our journey, we ultimately decided on command, strategy, memento, and singleton.