Name: Zachary Haney

Project Description:

The project is a short, single-level platform game based on the metroidvania subgenre of video games. The player moves a character through a small labyrinth of platforms, collecting items to aid in progression and ultimately trying to reach an exit. Along the way they find their progress impeded by breakable walls, nasty enemies, and dangerous spikes.

Implemented Features:

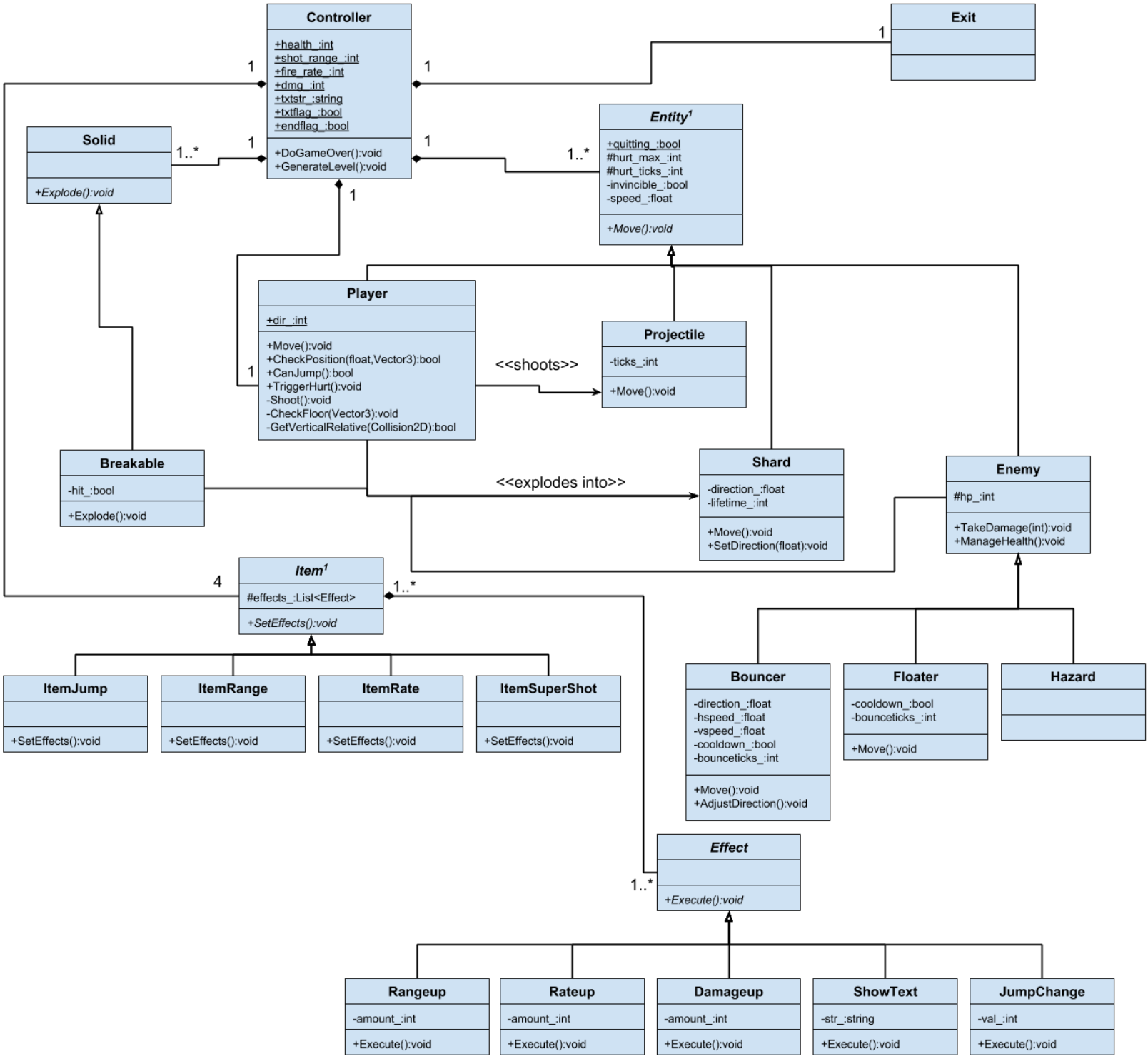
ID	Feature
1	The user must be able to move their character around
2	The user must be able to encounter a variety of enemies/hazards
3	The user must be able to interact with enemies, through means such as attacking them and taking damage from them.
4	The user should be able to find items that grant their character new abilities
6	The user should have the screen scroll to keep their character visible when nearing an edge in a large room.
7	The user should have a way to win the game.
8	The user should have a way to lose the game.
9	The user should be able to have their character collide with objects.

Unimplemented Features:

ID	Feature
5	The user should be able to switch screens when entering a door.
10	The user should be prompted with an easy to navigate title screen before starting the game.
11	(Stretch) The user should be able to save and load their game
12	(Stretch) The user should be able to encounter one or more boss monsters in order to have a challenging experience to overcome.

Final Class Diagram:

(Due to the amount of methods and parameters some of these classes hold, scaffolding code has been omitted for those particular classes to conserve space.)



1. Entity and Item aren't true abstract classes due to how Unity handles scripts that are directly attached to game objects. However, there are treated as such for the purposes of this class diagram.

A lot has changed since the previous version of the class diagram. Several proposed methods and fields were either replaced with ones with similar functions, removed entirely, or, in the case of some methods, were found to have similar methods already defined in the MonoBehavior class (the parent class that all scripts in Unity's framework must inherit from if they are to be directly attached to game objects.) and needed only to be implemented. Most of these minor changes are due to both estimating and being undecided on a framework to use at the time of the original diagram was created.

However, a major change is that the proposed Room class from the original diagram wasn't implemented due to time constraints and its role of handling level generation was given to the game controller. As the game currently only generates one level, this is adequate for now. Had this had more time to develop into a larger game, the controller's level generating code would've been split off into its own object.

Another significant difference is that the Solid class was split from the Entity class, made into a standalone object, and given a subclass of its own. The reason for this is that the Entity class was redefined as encompassing only objects that are capable of movement (with the exception of Hazards.). Solid objects do not move in-game. Therefore, it would be pointless to have them inherit behaviors only meant for objects with the capacity to move. In a similar vein, the Door class was renamed to Exit and was also split off into its own object, however, it is not a child of Solid due to being a background object.

Additionally, the Item class (formerly known as Collectible) was also split from the Entity family, but for an additional reason: Despite also being a type of immobile object, the Item also had to be separated in order to implement the design pattern the Item object and its children use for instantiation. In order to implement this design pattern, several features intended for derived classes of item were split into their own family of classes, all derived from an abstract class known as Effect.

Design Pattern:

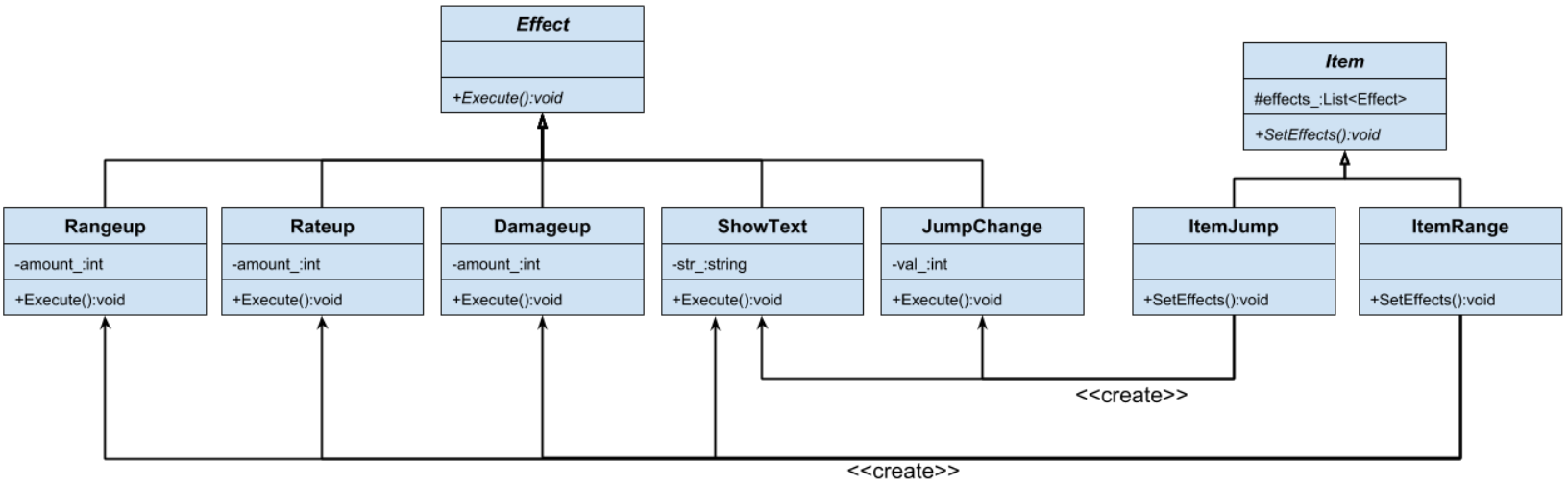
The game uses a Factory design pattern to implement the four collectible items found in the game world. Factory was chosen due to the ease of which it can be used to define new Item variants, and for its ability to allow an Item to instantiate its behaviors and effects without ever needing to define any new methods to achieve a similar effect.

The products in this Factory pattern are children of the abstract class Effect. All Effect derived classes have a shared method called Execute() which executes any behavior defined in their implementations.

The Item class defines the SetEffects() method, which acts as the factory method for the pattern. The four Item variants derived from this class implement this method by adding instances of classes derived from the Effect class to a protected list field. The Item class also

implements a collision function defined in MonoBehavior, where upon collision with the player it calls the Execute() function of every Effect in that instance's Effect list.

A diagram of the design pattern is illustrated below, using two of the four Item subclasses as an example.



What I learned:

I've learned that analysis and design is a challenging yet rewarding and fun experience. I faced difficulties at several points during this project, such as deciding on a framework to use, what design pattern to implement, and what design patterns can be implemented within the limitations of my chosen framework.