Architecture Report

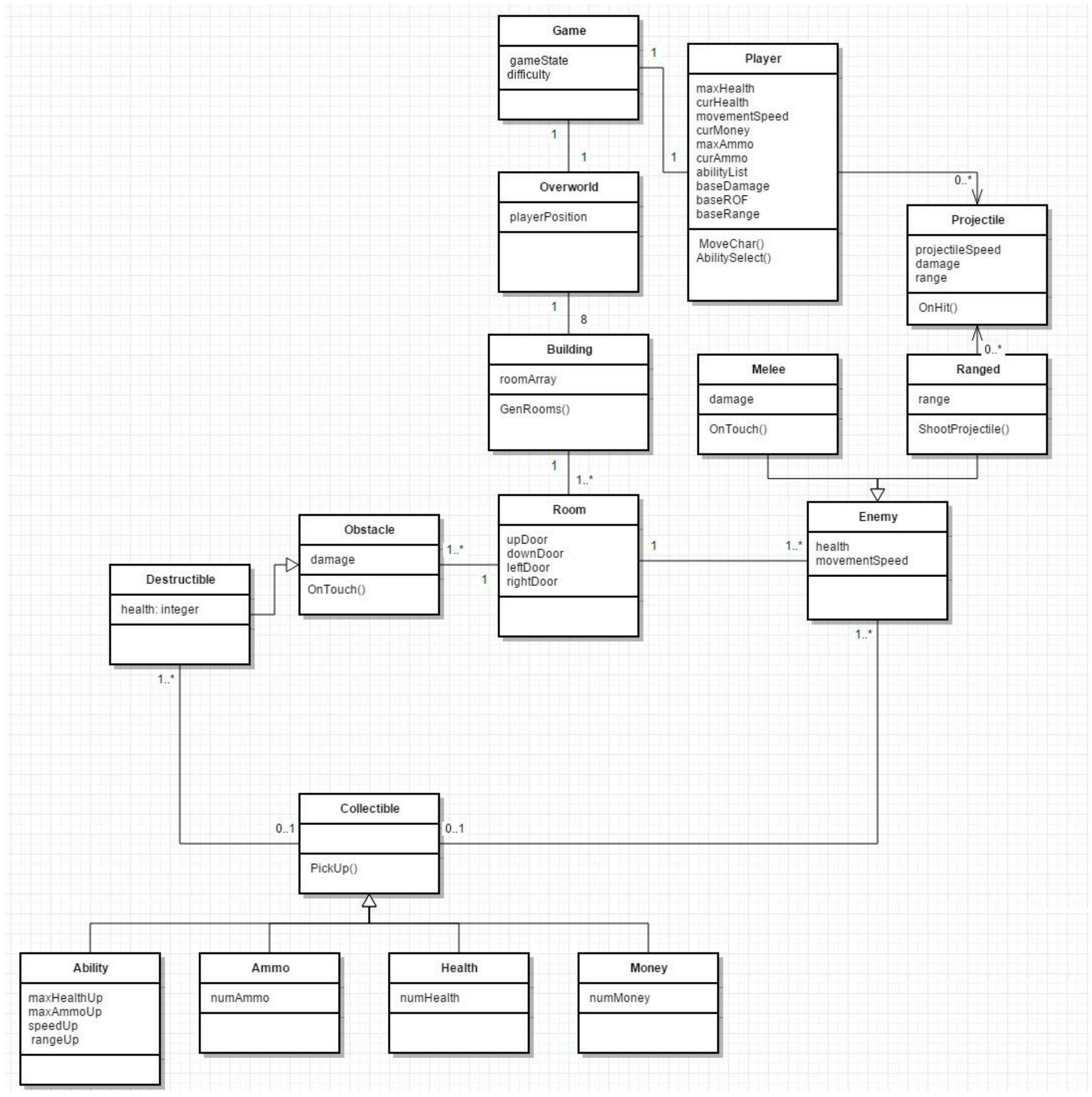
Class Diagram

Leading on from the abstract architecture for the game presented in Assessment 1's architecture report, we have made a number of changes as we progressed with the development of our game. Specifically, this saw additions to our class diagram, with attributes and operations added to the majority of classes as we began designing algorithms and discussing what each class would need to be able to do. It should be noted that this is not an in-depth breakdown of the game- additional attributes and functions may be implemented, as well as additional classes that may be required by our choice of platform in order to run (for example, classes to render sprites on the screen).

As before, we are using UML 2 to describe our class diagram, and the website Gliffy.com (https://www.gliffy.com/) to implement the class diagram.

Before we discuss the attributes and operations, we should note the changes that have been made to the structure of the class diagram. The 'Boss Room' class, which was previously a subclass of 'Room', has been removed. We decided that there was no real need for a specific class for boss rooms, as it would still be fundamentally the same as a standard room- it would have one enemy and a number of obstacles associated with it, similar to a normal room (which would likely have more than one enemy instead of just a single one).

Similarly, the 'Non-Destructible' sub-class of 'obstacle' was removed as was not going to add any additional functionality to the 'Obstacle' class. Furthermore, we have removed the 'Overworld' class, as a result of design changes moving from a 'Legend of Zelda' style overworld to one similar to the 'Super Mario Bros.' series (see GUI report for more information about the change).



Class Breakdown

We have introduced a single variable to the ‘Game’ class, gameState, which will act as a global variable denoting the state of the game (e.g. on world map, playing etc.). This will allow us to control the execution of functions in the game (we could place guards at the start of the function that stop it from executing in certain game states), which will help us to implement requirement 10 (“The game must end when every area of the university has been conquered, or, when the user dies.”). In addition, representing the game state in this way will make it easy to save the current state of the game, satisfying requirement 2.

To be able to represent buildings in the desired way, we will need an attribute for ‘Building’ that will store the layout of rooms. This will likely be a 2D array of ‘Room’s. We will also need to provide a function that will generate the room layout (randomly) for the building- this functionality is represented on the class diagram by the ‘GenRooms’ operation for ‘Building’.

Building on from above, the ‘Room’ class will need some attributes to allow the player to navigate between them- this is in the form of upDoor, rightDoor, downDoor and leftDoor, which will represent where doors exist in the room (each attribute corresponding to one of the room walls). We will only draw (and allow players through) doors where we have specified the location of one when we generated the building.

The ‘Player’ class representing the user’s in-game character will require a number of attributes in order for the game to be able to monitor various things such as health and what abilities the player has access to. Health will be determined by ‘maxHealth’, holding the maximum amount of health the player can have, and ‘curHealth’, representing the player’s current health (as they will take damage over the course of gameplay). Similarly, ‘maxAmmo’ and ‘curAmmo’ allow for the same idea to be applied to the amount of ammunition the player has when they use a ranged attack.

To represent money, we will use ‘curMoney’ to hold the current amount of money that the player has collected. With regards to abilities, ‘abilityList’ will allow the game to decide if the player has collected certain abilities and hence allow/deny access to said abilities, allowing us to satisfy requirement 8.

Four attributes, ‘movementSpeed’, ‘baseROF’, ‘baseDamage’ and ‘baseRange’ will be used internally to determine how quickly the player moves, as well as parameters for their attacks. The reason for implementing attacks this way is to make abilities and pickups affecting the player’s attacks much easier to implement (i.e. we can just add a modifier to the relevant attribute), and thus assisting with completing requirement 1.1.

We will also need to provide functions that will handle what happens to the user’s character when they give an input (e.g. wasd keys to move). Therefore, we have included MoveChar(), which will handle player movement, and AbilitySelect() to allow the player to swap between their different abilities that they have collected.

The ‘Enemy’ class, as the parent of both ‘Melee’ and ‘Ranged’ subclasses of enemies, will have shared attributes between the two. Thus, we have added a ‘health’ and ‘movementSpeed’ attribute, as these will be used by both types of enemy. As melee enemies don’t have a range, and only damage the player on contact, we only need a ‘damage’ attribute to implement this. Ranged enemies, on the other hand, will shoot projectiles at the player. the ‘range’ attribute will govern how far the projectile can travel before it is removed from the game.

Each projectile, represented by the ‘projectile’ class will need to have its own attributes in order to damage targets properly. Therefore, its speed, damage and range will be represented by the attributes ‘projectileSpeed’, ‘damage’ and ‘range’. The OnHit() function will trigger when it makes contact with a damageable object.

Generally, obstacles (represented by the ‘Obstacle’ class) won’t need many attributes or functions, as they are static. the ‘damage’ attribute will be used in a similar manner as melee enemies, and inflict that damage on contact, which will be handled by the OnTouch() function. Objects that don’t damage the player will simply have a damage of 0. Destructible obstacles will also need a health attribute, so that the game can decide when it ‘dies’.

Finally, the ‘Collectible’ class and its subclasses have had attributes and operations added. the ‘PickUp()’ function will trigger when the player touches the collectible.Depending on what the effect of the collectible is, the function will use one of the attributes of whichever subclass the collectible is of to change the player's stats. For example, if the player picks up a health pack, the function would use the health pack’s ‘numHealth’ attribute to increase the player’s ‘curHealth’ by its value (of course, remembering to cap the increase to the maxHealth).