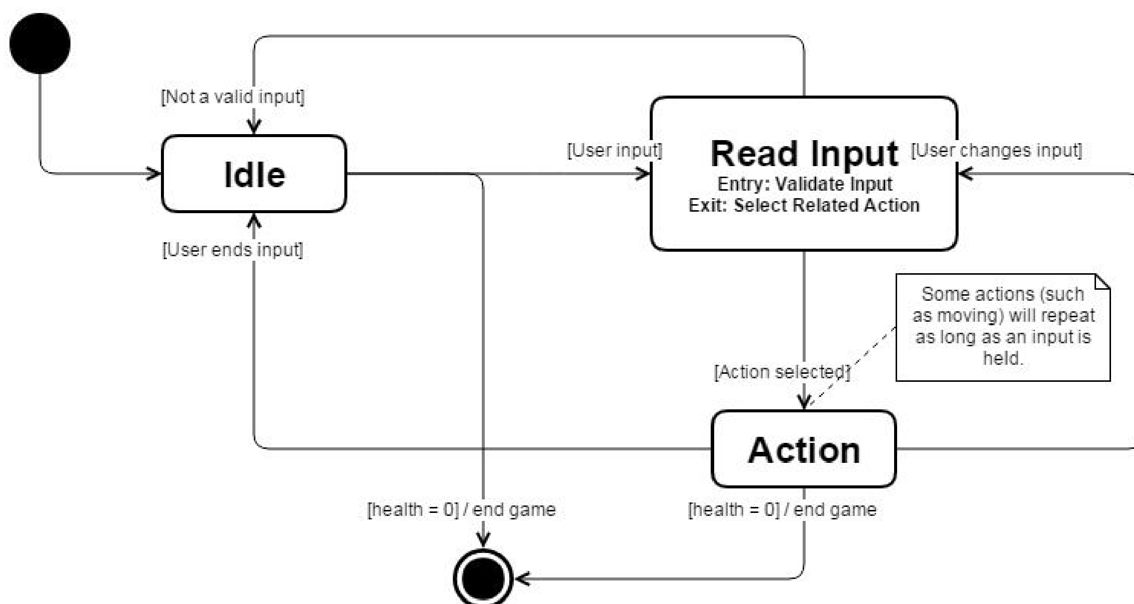
Architecture Report

Modelling

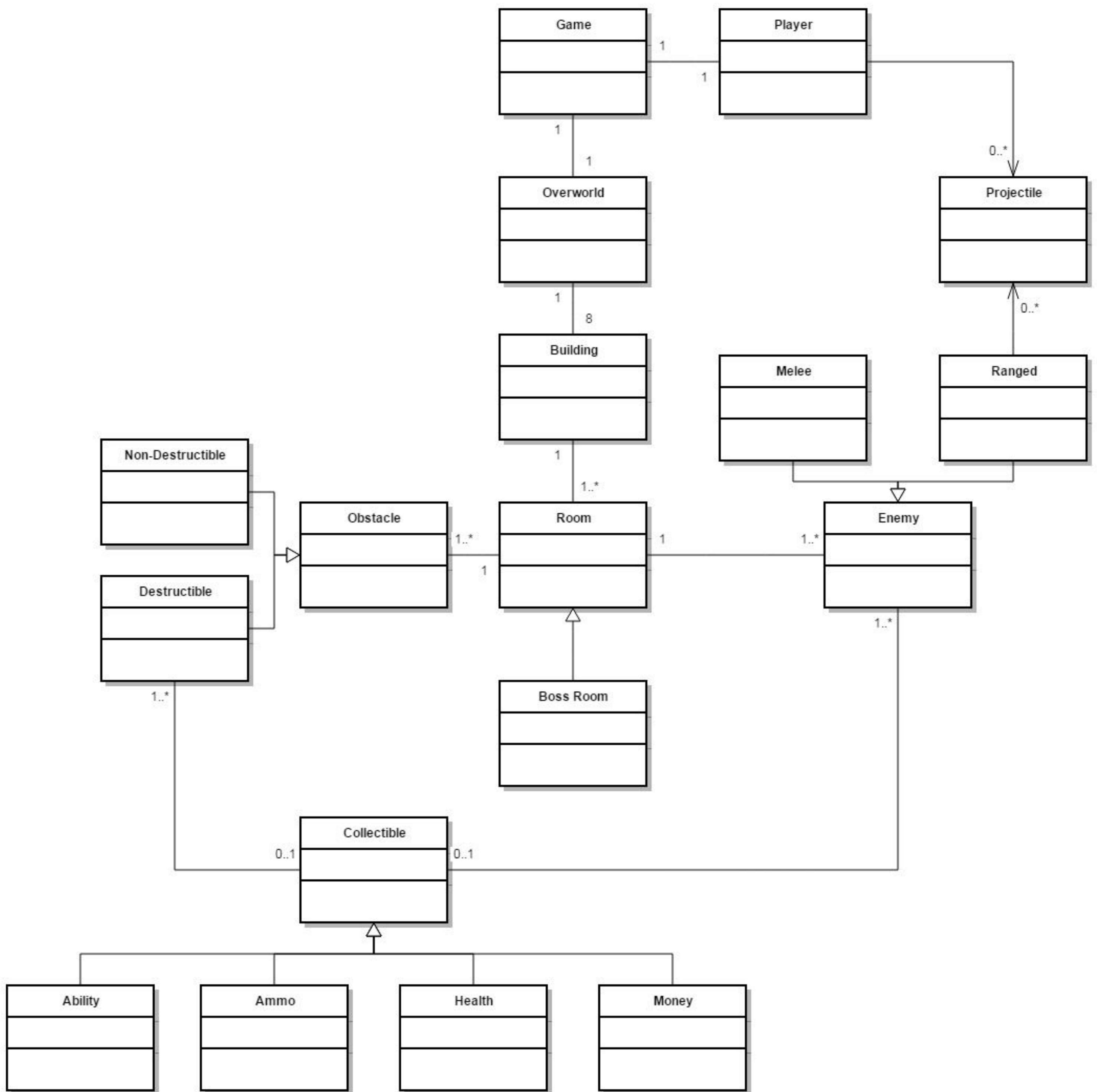
In order to obtain a clear image of the structure of our game elements, we will be using UML 2 to create class diagrams of each entity in our game. The use of UML 2 will help us to organise our thoughts, and clarify how each entity relates to each other, as well as any attributes and methods these entities will require.

After exploring a number of options, we decided to use Gliffy.com to build our UML class diagrams. We chose this particular editor as it has an intuitive and easy to read user interface, and options to export our class diagrams as JPEG images that we can add directly into our documentation. The grid layout made it easy to align elements of the diagram.

In addition, we decided to build a small state diagram representing the interaction between the user and their character in-game. This will help us at a later stage when we design the operations that the 'Player' class will have in a more concrete form. As shown below, the character will default to an 'Idle' state when the user does not give any input (such as 'wasd' movement or mouse clicks). When the program detects an input, the class will check if the input corresponds to an action, and perform said action if it finds a match, otherwise returning to the 'Idle' state. Included in the state diagram is functionality to change action immediately without returning to the 'Idle' state (which may cause the player to experience some lag when they change input), and a terminating state when their character's health is reduced to 0.



Class Diagrams



Class Breakdown

Before we constructed the class diagram, we began by identifying all of the potential classes that we would require in order to implement the game. The 'Game' class is the 'master class' at the top of our architecture- it handles the game state (whether the game is currently running, or if it's a game over) and what difficulty is the game set to. Requirement 10, which states that *'the game must end when every area of the university has been conquered, or when the user's health and lives are both 0'*, can be met by implementing this functionality into the 'Game' class, and we also satisfy requirement 11 with the implementation of the difficulty setting. This class will also be responsible for controlling the GUI that the player sees, complying with requirement 4.

As we chose to implement a design similar in style to the original Legend of Zelda game, we have an 'Overworld' class which represents the world map that will be used to traverse between buildings. This will be a map of the University campus, with each campus location that the player can move through having its entrance on the overworld map, as per requirement 3.

Each building itself (which is analogous to the Legend of Zelda's dungeons) will be represented by its own class, which stores the building's room layout. Each building (8 in total) will be a representation of an actual campus building which was a necessity in our game, showed by requirement 3. Requirement 6 and 7 states that '*The game must contain at least 8 different objectives for the player to complete*' and '*The game must have rounds'* respectively. We have chosen to represent each 'round' with the implementation of individual buildings on campus as its own section of the game to be completed. Each building will have at least one objective that is needed to be completed before you can progress. We will adhere to requirement 6 by representing at least 8 buildings. The game will contain rounds and each different building entered will be a new round in the game, this satisfies requirement 7. Finally, as each building represents a different round in the game, this means the game is compartmentalised and therefore will be easy for the university to use on UCAS and Open Days which was specified in requirement 12.

The room layout will consist of a number of 'Room' instances. A subclass of 'Room', 'Boss Room', will also be implemented, allowing us to generate a specific type of room designed to hold a boss enemy inside rather than the usual mobs and obstacles. [Each room will have an objective (for example, 'defeat all of the enemies') that must be completed in order to progress to the next room. Due to the fact that there will be one or more rooms in a building and there will be an objective for each room this means at an absolute minimum there will be 8 objectives before the game is over, hence, satisfying objective 6.

The player will also need their own class to represent their in-game character. Requirement 1 states that '*The game must feature the player character, which will be a duck*'. This will be implemented using a 'Player' class. This class will be used to store relevant data attributed to the player and their character, such as the character's health and how much ammunition they have. the 'Player' class will also satisfy requirements 8.1 and 15, which will allow the player to control the duck's movements, such as walking, flying, and swimming, all with the mouse and keyboard.

Enemies will be represented by the ‘Enemy’ class. This class will have two subclasses; The ‘Melee’ class to represent enemies that need to be in close proximity to the player to cause damage, and the ‘Range’ class which will represent enemies that will launch projectiles to damage the player. This satisfies requirement 5.

There will be duck-related terrain present in the rooms, which will be randomly generated. This terrain will be implemented using the ‘Obstacle’ class, which will have subclasses representing whether the terrain is destructible or indestructible. This will partially fulfill requirement 5, but will not be the only type of duck-relevant obstacle.

There will be collectible items which benefit the player, and will assist them in completing objectives, which is requirement 8. There will be 4 different types of collectible: Ammunition, Health, Money, and Ability. These will all be subclasses of the class ‘Collectible’. The subclass ‘Ability’ will also satisfy requirement 8.2 - these collectible abilities will be rarer than the ammo, health and money.

Finally, an instance of the ‘Projectile’ class will represent a projectile fired by either the player or a ranged enemy. Note that the ‘Projectile’ class has unidirectional associations leading to it from ‘Player’ and ‘Ranged’ classes. This is so that we eliminate the inference that a single projectile would be associated with the player AND a ranged enemy, which is not what we want - each projectile should only be associated with the sprite that instantiated it. We have chosen to represent ranged attack in this way as the projectile will have its own sprite in the game, and will need its own hitbox, and attributes to hold its speed, damage and so forth, fulfilling Requirements 5.4 and 5.6.