Implementation Summary

Expansion of the Architecture

Generally speaking, our implementation for the project follows the ideas laid out in the updated class diagram in the architecture report. Included below is a table, which lists the classes we have used in our code, the class in the class diagram that our implementation is representative of (where relevant), and comments about why this choice was made (where relevant).

Class in Implementation	Class in Architecture	Comments
Collidable		Separate class which handles all collisions in-game i.e. collision detection.
DungeonRoom	Room	The class keeps a track of all of the entities that are inside the room (obstacles, enemies etc.).
Enemy	Enemy	The enemy subtypes have been collapsed into the same class- we have represented them having an integer value mapped to the type of attack that an enemy has.
EntityManager		While this does not represent any classes in our architecture, this made it easy for us to manage our sprites in the game.
GUI		This was also included for ease of use, by collecting any GUI elements that our game would use.
Item	Collectible	Represents permanent ability pickups rather than things like health packs. Currently does nothing.
Level	Building	Contains the room layout (generated by the LevelGenerator) and the objective of the room.
LevelGenerator		The building generation algorithm is placed in its own class here.
MuscovyGame	Game	The ‘master’ class of the game.
Obstacle	Obstacle	This represents indestructible obstacles. The class can be used to represent those that damage the player on contact and those that do not.
OnScreenDrawable		Wrapper class for sprites.

Pickup		Represents small pickups (like health packs) rather than permanent abilities. Currently does nothing.
PlayerCharacter	Player	Handles the character response to the user input. Also handles animations for the player sprite, but is not currently implemented.
Projectile	Projectile	Represents the player's and the enemies' projectiles. This class allows us to control the projectile's damage, range and speed.

Algorithm Design

With regards to our 'Level' and 'DungeonRoom' classes, we will need to consider:

a) The data structure of the 'Level'/'DungeonRoom' that will hold its contents
b) How the entity's contents will be generated

With the 'Level', we will need to store the layout of the rooms that compose the building. A 2D array is the intuitive choice for this, where each location in the array will be either null or point to an instance of a 'DungeonRoom'.

When considering the generation algorithm, we decided on certain criteria that we wanted. The main idea was that we did not want any loops appearing in a 2x2 square. After considering the problem, we chose to use the following algorithm to generate our rooms:

1) Initialise a 7x7 array and place the starting room in the centre, 3x3.
2) Loop over the array until we have the desired number of rooms.
    a) For each non-empty position (meaning a room exists there), randomly decide whether or not to place a room in each adjacent position (50% chance).
        i) We will only place a room if the new room's position will have 1 adjacent room (i.e. the pre-existing room).
3) Place the boss room, item room and shop room.

To make checking the number of adjacent rooms easier, we implemented a separate method which when passed a room position, calculated the number of adjacent rooms (allowing for multiple parts of the code to run the same code easily). The algorithm was first prototyped in Python in order to verify its feasibility, after which the algorithm was then implemented in the context of the project. The algorithm can be seen in the 'LevelGenerator.java' file.

For our DungeonRoom layout, we have mapped a 18 x 10 grid onto the floor space for the room. The code will pseudo-randomly generate the room contents by selecting a random predetermined layout to use, after which any random entities inside the room layout will be resolved. The reason for implementing the generation of rooms this way was so that was

could ensure that obstacles would never spawn in such a way that a door would be blocked off completely from access. Furthermore, should someone else wish to create a new room layout, they can easily do so by adding a new case to the switch/case statement and the 2D array of the room layout.