As we took control over *Team Mallard's* project we implemented and extended their codebase to include new features and to perfect and correct features that we felt were not implemented well. During the course of the document I will refer to the requirements document as well as the architecture document.

One of the first classes created was the "floaty numbers" class. The floaty numbers class gives a visual feedback to the player when they cause damage. When the player damages an enemy or receives damage a number will appear and act according to a function(e.g bob according to a sine wave), when the player kills an enemy the score they receive will also appear on screen and act accordingly. By doing this the player receives a greater amount of feedback in an intuitive fashion. We were inspired by games with similar mechanics such as Borderlands)[1]

A second addition that we have made to the codebase is the addition of a "Round" system. The round system allows us to implement levels with multiple types of objective By creating this abstraction between maps, objectives and rounds we can create new rounds(with new maps), without having to create new objectives tailor made for that specific map. An example of this could be a map of the "Computer Science" building, with one round having an objective of killing five enemies, and a second round having the player traverse the level to reach a flag. You could then carry over these objectives to a new map of the "Ron Cooke Hub" or to a map of "Heslington West".

A further feature we implemented to improve our game was to improve the AI. Previously all enemy AI would attempt to intercept the player and cause by damage by colliding with them. We implemented an AI system that also allowed enemies to attack the player from range. We also improved the AI system(A*) that "Team Mallard" implemented previously. Previously the AI would be called to be updated every frame, we then changed this to update once a second. We felt this was a necessary improvement, it allowed us to be able to spawn more enemies on screen at a time, in addition we felt it was unnecessary to check the AI frame as the player would not change position a large amount, and the map is static so there is no need for it calculate an updated route.

We also implemented was a eight directional aiming and walking system. When we first played the game we noticed that the player's sprite was locked to facing in the four cardinal direction however they could use ranged attacks in 360°. This looked odd when the player was facing left but would be shooting their gun almost vertically. This included movement, the player could be moving in a non-cardinal direction (e.g north-west) but the animation for moving north would play. The team implemented aiming and shooting in eight directions (North, Northeast etc.) by calculating the angle of where the player was looking and rendering a different player sprite depending on this angle. This made the shooting seem much smoother and feel much tighter. To continue we then also added the player being able to walk in the four diagonals, by taking the player vector and calculating it we were able to render the player moving to the Northeast,Southeast,Southwest and Southeast. Finally to continue with this theme, we implemented the player always facing in the direction of attack.This meant that the player would no longer shooting behind them and walk forward but would look and act correctly. Thus combined made the player feel much smoother and well responsive.

The following section will analyse how our codebase implements the requirements passed to us from the previous groups documentation as well as any changes made by our team. I will begin by looking at the "Gameplay systems" requirements.

To implement the requirement ***G1*** we will be utilising the *round*, *gamescreen* and *objective* classes of our game. Each round will contain a map and a related objective. The player will play a "round" on a map (e.g

Computer Science and Law and Management) to achieve an objective. Each map may have different types of objectives - meaning a player may play the same map twice but with two different objectives; one say to 'kill ten enemies' and the other to 'survive for an amount of time'. This extends into the requirement ***G2***. Which states that "The game will include at least eight different **'objectives'** (goals), of which there are at least two distinct types. Objectives can be either be failed or succeeded by the player". To achieve this we have created the two types of objective within the "objective" package. There is an abstract class "objective" which is extended by the classes "killObjective", "bossObjective" and "collectObjective".The abstract relationship allows us to quickly implement new types of objectives. Requirement ***I5*** also states that the "The current objective will be displayed at all times on screen." While the previous team had implemented this we then extended this to work with all types of objective, one example of this is in our kill objectives

The requirement ***G4*** was already implemented by the previous group. However to give more feedback we implemented the class "FloatyNumbers" to spawn numbers, which is used to represent the score given when an enemy is killed. The "FloatyNumberManagers" is used to manage the groups of numbers while the class "FloatyNumbers" is used to update the number itself such as its sprite and position.

Requirement ***G7*** and ***G8*** states that we need "obstacles" and that some of these obstacles could be implemented as enemies. We created new subset of enemies that utilise ranged combat as one of these obstacles. It also stated that obstacles could be collision obstacles that impeded the player's progress. To achieve this there are two(at least) layers within the map, "Collision" and "obstacles"(obstacles could be 0-any number, and a layer is randomly selected from these layers). Any tile within these layers the player can not move through.

The initial implementation we received from the previous group already had an implementation of powerups. However they implemented it so that the most recent powerup collected would override the previous powerup. However we felt that this punished the player as they may have a powerful powerup that is then overwritten by a less powerful one. We then changed their requirement so that they could stack, allowing the player to have multiple power ups (***G11***). We implemented a powerup manager class which manages the active power ups with methods for updating and rendering the power up bars and maintains an instance for each powerup. Each powerup is also an extension of the item class, as we have changed the game so that the player always has a gun and a melee attack, the item class is a left over from the previous codebase, but is useful as it can be used for extensibility if necessary.

Requirement ***I6*** requires that when a player or enemy is damaged or score is gained this will be displayed as a number on screen. As discussed above this was implemented using the floatyNumbers class and the associated manager.Initially the player did not receive these numbers however we felt damage numbers were necessary to affirm whether a player or enemy was being damaged or not.

To implement swimming for requirement ***C4.*** For all characters(players and enemies) there is a method isOnWater this method reads the current tile that the player is on. This method is called whenever movement is performed, it gets the current location, and checks this against the base layer off the map, if the base layer has the property "water" the animation will swap to a swimming animation.

From the previous group we had a rudimentary implementation of flying. We took this implementation and changed it according to our requirements. We felt the player should be able to fly over obstacles and enemies as a means of escape, not just as a means of a quick dash. To ensure that the player does not fly and get stuck on top of a obstacle (rendering them unable to move) the player class reads the tile

beneath the player and checks if it is collidable, if the tile has the collision property the player will continue moving until the tile does not have the collision property.

The following table demonstrates the features that we have and have not implemented by referring to the requirements document:

Gameplay Systems	Implemented(Y/N)	Interface / Visuals	Implemented(Y/N)
G1	Y	I1	Y
G2	Y	I2	Y
G3	Y	I3	Y
G4	Y	I5	Y
G5	Y	I6	Y
G6	Y	Control/Movement	Implemented(Y/N)
G7	Y	C1	Y
G8	Y	C2	Y
G9	N	C3	Y
G10	Y	C4	Y
G11	Y	C5	Y
G12	Y		
G13	Y		

Bibliography

[1]Gearbox Software.Borderlands.borderlandsthegame.com,2009