Implementation Summary

The following section will discuss the design decisions relating to the architecture of the game. Within the document diagrams will be provided; such as inheritance or state diagrams. In the final segment of this document the updated requirements are analysed and requirements that have been implemented / not implemented have been noted.

Our UI (user interface) system was tailor made for our project - this was implemented using nine objects organised into one package. The three main objects are the UIManager, UIComponent and UIRenderer objects. The UIManager stores a list of all the current UIComponents and provides methods for adding, removing and getting UIComponents. The UIRenderer provides methods for rendering, resizing and disposing of the components within the specified UIManager. The UIComponent is an abstract class that other components inherit from these include components such as, UIAgentList, UIDoubleList, UIList and UIMessageBox. These components implement more complex UI actions such as lists and message boxes

Another important characteristic of our project is the relationship between the “character” object, the “NPC” object and the “player” object. The overarching class “character” is abstract containing methods that are useful both to the “player” object and to the “NPC” object. This includes methods for updating movement for characters, their current position in relation to the map and methods to activate the character. In addition the character object contains the states which inform the characters' movement behaviour: these are “Stationary, Waiting, Transitioning” and “Up, Down, Left, Right” respectively. The Player and NPC objects then inherit from the character class, providing methods for updating their movement while in a transitioning state or in a stationary state.

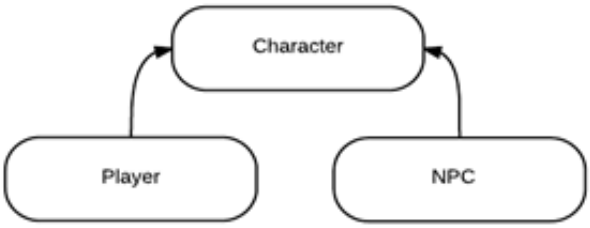

FIGURE 1: Inheritance for character, player and NPC.

As the primary component of our game involves the battling system (as well as the world map), this needed to be implemented alongside the world map so that play flows seamlessly between the two systems. These are the two objects that extend the ScreenAdapter class of LibGDX, these are the “WorldScreen” and “BattleScreen” objects. These objects manage the functions of a window (such as the height and width) as well as calling the other objects necessary for the game to function such as the renderers (“WorldRenderer” and “BattleMenu”) as well as the other objects necessary for the game to function. This includes (for the BattleScreen) initialising characters (as well as equipping them with items and skills), initialising the UI, and assigning the position of the characters in battle. To continue the “WorldScreen” initialises the Json loader (discussed in more detail below) as well as the UI for the world map. By utilising the screens function of LibGDX it allows us to transition between the two aspects of the game retaining information easier than using our own system.

Within our game we will be needing to store a large number of skills, items and party members. Rather than creating them all within the IDE itself we will be using the data format Json [2] to store a large amount of information and loading the information via an object "JsonLoader. The three objects loaded using the loader are the "ItemManager", the "SkillManager" and the "PartyManager". Respectively, "Agents" are loaded into the "PartyManager", "Skills" into the "SkillManager" and "Equipables" and "Consumables" into the "ItemManager".

One algorithm that I will be looking at is the algorithm used for moving agents (characters/ player/NPC, refer to figure 1) around the world map. We were influenced by the *Pokémon* [1] style of movement. This style of movement is where the "character" transitions fixed from one tile to another. The "character" will always move an integer number of tiles in a direction, a movement key can also be pressed to change the direction the agent is currently facing. As movement is a key component of the game and if the "feeling" is off it can throw off the player it had to be correct therefore our movement system had to be implemented perfectly. Our implementation relied on three states "stationary", "waiting" and "transitioning", and movement is based off a given direction and a current state. The state diagram is given below of how we went out implementing it. The only difference between a "NPC" and a "Player", is that a NPC's next direction is a random function, while the player's movement is based on an input. [Figure 2]

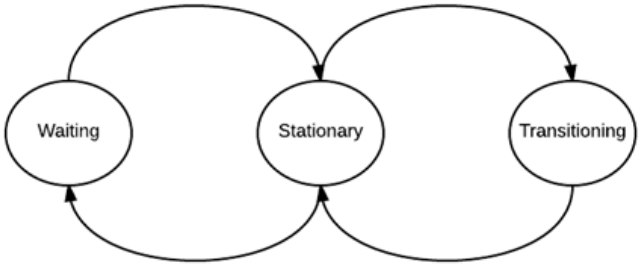

FIGURE 2: State diagram for character movement

In the theme of movement to implement tiles that cannot be moved through within our game we will be using a collision map generated from an imported tile map. To store the tiles marked as "blocked" we will be using a two dimensional array. As the map is loaded in game, every tile within the map is scanned row by row and column and column, if a tile contains the key "blocked" they are then stored within the two dimensional array. When a character moves, its target position is checked against the corresponding position in the array, and if it is marked as blocked then the target cannot be moved to and the player stays still.

Below is a table, following the same notation as in the requirements document, outlining which requirements were and were not implemented.

All features that were required to be complete by the Assessment 2 deadline have been fully implemented.

WORLD MAP		**BATTLES**		**USER INTERFACE**		19	Y
1.1	Y	7.1	Y	11.1	N	**USER**	
1.2	Y	7.2	Y	11.2	N	20	Y
1.3	Y	7.3	Y	11.3	N	20.1	Y
2	N	7.4	Y	12	N	21	Y
3.1	N	7.5	Y	13.1	N	22	Y
3.2	N	7.6	Y	13.2	N	23	Y
4.1	Y	7.7	Y	**TESTING**		**MODULARITY**	
4.2	N	**ITEMS**		14	Y	24	Y
4.3	N	8.1	N	14.1	Y	24.1	Y
5.1	Y	8.2	N	14.2	Y	24.2	Y
5.2	N	8.3	N	14.3	Y	25	Y
PARTY		8.4	Y	15	Y	25.1	Y
6.1	Y	**OBJECTIVES**		15.1	Y	25.2	Y
6.2	Y	9	Y	**NON-FUNCTIONAL**		26	Y
6.3	N	9.1	N	16	Y	26.1	Y
6.4	Y	9.2	N	17	Y	27	Y
		10	N	18	Y		

Bibliography:
[1] Nintendo(1999).Pokemon Red/Blue.http://www.pokemon.com/uk/pokemon-video-games/pokemon-red-version-and-pokemon-blue-version/ [Accessed 12-Jan-2016]
[2] JSON. http://www.json.org/