Architecture: Update

Architectural Model

We found UML 2.0 was adequate at expressing the structure of our architecture without any need to define additional notation so we will continue to use the language along with *Lucidchart*, the modelling tool we used in the first deliverable. We will continue with *Lucidchart* as we have seen no evidence to suggest any other tool would be more appropriate. To aid the creation of our UML diagrams and to visualise the structure of the current state of the code we have used a plugin for the IntelliJ IDE called Code IRIS which generates a UML diagram based on references and usages in the source code. We found this tool useful when comparing our current source code to our abstract UML diagram. Labels such as "<<*abstract*>>" will be used any situation where standard UML 2.0 can't express an important feature of the architecture. There will be further discussion over the exact architecture of the *ItemManager* and *SkillManager* class which we could not express without adding the attributes of the classes.

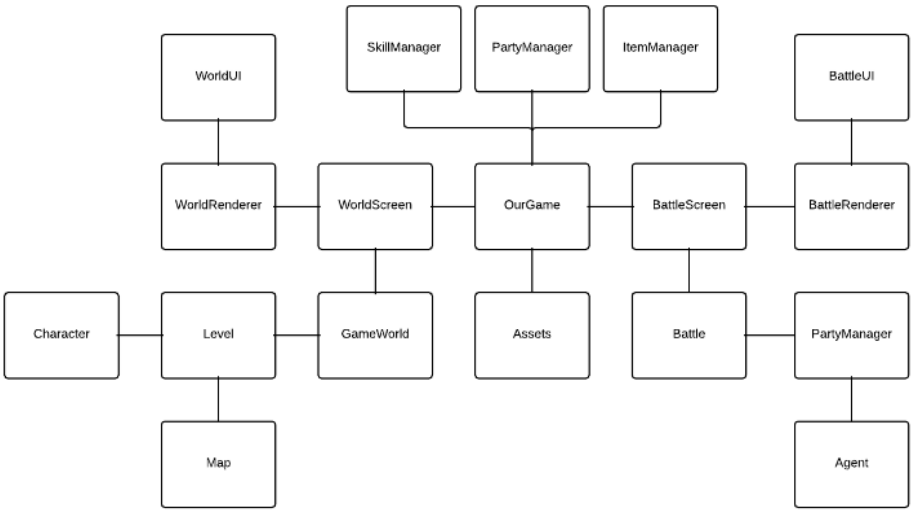

Figure 1. The abstract class diagram which was built on top of.

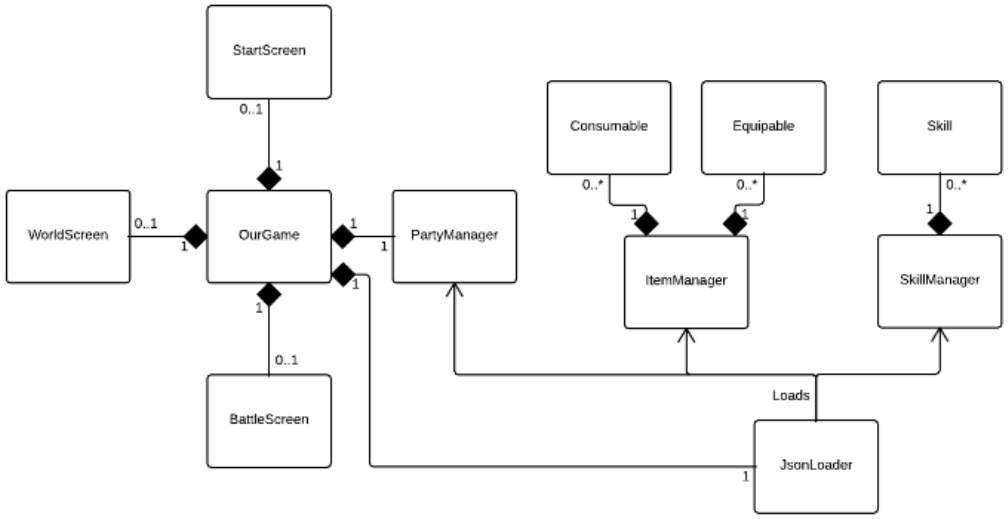

Figure 2. An overview of the general structure of our architecture.

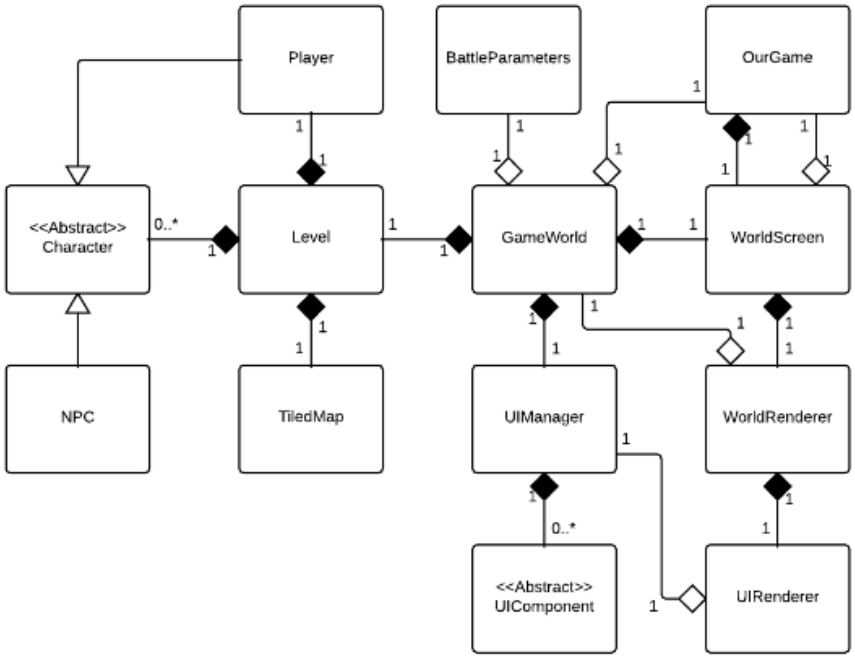

Figure 3. The WorldScreen expanded.

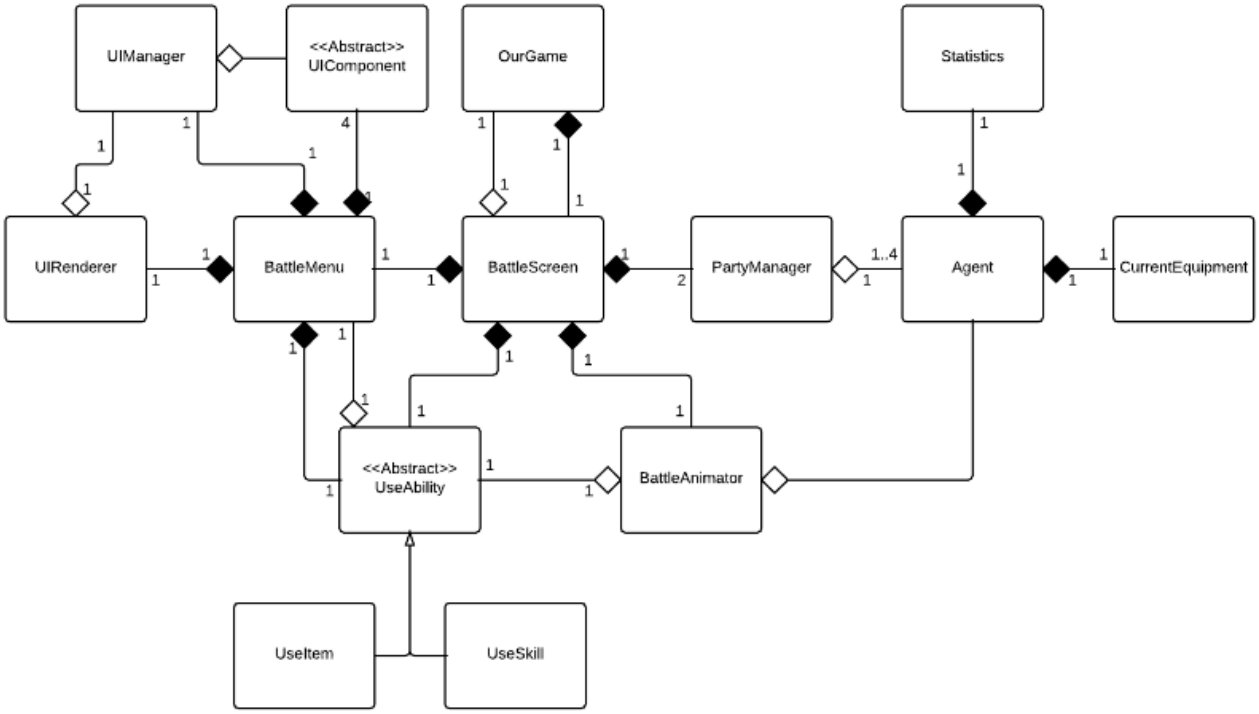

Figure 4. The BattleScreen expanded including the PartyManager structure.

Abstract Architecture

This abstract architecture is a revision on the architecture submitted for Assessment 1, removing the detail regarding the types of relationships between classes and the multiplicities after receiving feedback. This is where we have built our concrete architecture on top of. The general structure of our architecture comes from splitting the *OurGame* into the two separate sections of the game, exploring a game world (*WorldScreen*) and battling with enemies (*BattleScreen*). We tried to carry this principle of loose coupling throughout our concrete architecture as it is one of our modularity requirements (25) [1].

Overview

The basic structure of having the *OurGame* class containing a group of managers, the assets and a Screen for each distinct part of the game has not changed much. *ItemManager* and *SkillManager* now use the Singleton design pattern. Although the two classes don't have a private constructor we use the principle of having one instance of the class which contains static methods to get instances of the Skill and item (Equipable and Consumable) classes. Singletons are often criticised for keeping a global state making it hard to test reliably however we keep the state of our attributes constant so the order of testing will not have an effect on the result [2]. The advantage to using this pattern was readability and avoiding a scenario where multiple instances of the exact same skill would belong to different Agent instances, wasting memory. This relates to the performance of the game seen in requirement 16. We introduced the *JsonLoader* class which allows us to load in the state for the *ItemManager*, *SkillManager* and *PartyManager*. We used *Json* files to store the Items and Skills since it allowed us to quickly change attributes of certain skills or items without hard coding each one into the source code. We also load in the current party which serves as a form of save state which contains the party members (Agent class) statistics. Other classes added include *Consumable*, *Equipable* and *Skill*. *Consumable* and *Equipable* are the two types of items currently implemented in the game, each having a different behaviour when used in battle. The *StartScreen* class is a new class that renders the starting splash screen and changes to the *WorldScreen* when an input is given.

PartyManager

The *PartyManager* would appear to be largely the same as the abstract diagram however an important part of the architecture is actually going on in this class. The *PartyManager* contains two lists of integers IDs representing the Consumable and Equipable items in the party's inventory, we didn't represent this on the UML diagram because this would require attributes being added. Similarly the Agent class has Skills stored in this exact same manner. The reasoning behind this decision was to keep only a single instance of each object in a central and easily accessible place instead of having the scattered across many instances of the Agent class. No friendly or enemy class as their behaviour did not differ from each other and the standard agent class.

WorldScreen

A change compared to the UML diagrams is the addition of Character class that serves a similar purpose as the Agent class in the *BattleScreen* section of the architecture. One consideration we thought of was to use the Agent class to represent the game characters in both the *BattleScreen* and *WorldScreen* as the information for each character could be easily shared however this would have led to tight coupling which can cause poor maintainability

and readability of code [3]. The *WorldUI* class no longer exists but has been replaced by a *UIManager* class. They serve a similar role keeping track of many *UIComponents* which a *UIRenderer* can then access. We added the *UIRenderer* to try and keep closer to the principle of single responsibility and not letting the *WorldRenderer* deal with the map, characters and UI [4]. We decided to make a separate class for battle parameters to help the readability of the code. In a language like C++ we could create a Struct and have no need to create a new class however this is a limitation of using Java as our programming language and an example of how our language has influenced our architecture. We are storing multiple characters in the level including the player and NPCs as well as having the player also stored in a separate variable. We did this so we could iterate through each of the characters in the scene and call some function for each character as well as not having to search through a potentially large list of characters for the player. For very little extra memory this structure is much more efficient. Both Renderer classes access the assets needed to draw the scene directly through static fields / getters and setters in the Assets class. We felt our solution worked best for a large system where multiple renderers would need access to this class and made writing the code slightly quicker and more readable in the end. The *UIComponent*, *UIManager* and *UIRenderer* are all new classes we introduced to help render and traverse through the UI. These three classes will be discussed in the UI section in further detail.

BattleScreen

We have removed the battle renderer class as we didn't need the extra functionality in a mostly static battle scene. The reason we required the *WorldRenderer* class over the camera class provided by the *libGDX* library was mainly due to needing to render the tiled map along with the UI and list of characters. The new class *BattleMenu* is essentially serving the same purpose as the *BattleUI* class in the abstract diagram. Two new classes are called *UseAbility* and *BattleAnimator*: the role of a child of the *UseAbility* class is to pass information about the move each agent makes to the BattleMenu, BattleAnimator and BattleScreen. *UseAbility* itself is an abstract class which is inherited from *UseItem* and *UseSkill*. These are the two types of actions an agent can do in its turn. This enables the functionality specified in requirement 7.5 and is the core mechanic in our battle system. The class contains information about the user, the target and the skill or item id. The BattleAnimator class manages the positions of each of the agents and animates them while they are using an ability (skill/item).

UI

The architecture of the UI in both the WorldScreen and BattleScreen was not considered in the abstract diagram due to avoiding excessive detail. I have excluded the diagram of all the classes that inherit from the UIComponent class due to page restrictions and the simplicity of the relationships. The UIComponent class is the class which all other UIComponents inherit from. A child of UIComponent has the job of drawing a certain kind of UI to the screen. The reasoning behind this structure is allowing one data structure to hold many UIComponents and being able to render them by going through the data structure and calling the draw function for each component. This makes the code more maintainable and allows the addition of different UIComponents with greater ease. The UIManager class has the job of containing all the different UIComponents and allowing the UIRenderer to access them and call their draw methods. This decision was influenced by the principle of having one class have one purpose.

Bibliography

[1] S. McConnell, Code Complete, 2nd ed. New York: O'Reilly Media, Inc., 2004, p. 80.
[2] J. Rainsberger, "Use your singletons wisely", Ibm.com, 2016. [Online]. Available: http://www.ibm.com/developerworks/library/co-single/. [Accessed: 19- Jan- 2016].
[3] R. Pressman, Software engineering, 5th ed. Boston, Mass.: McGraw-Hill, 2000, pp. 352-355.
[4] R. Martin, Agile software development. Upper Saddle River, N.J.: Prentice Hall, 2003, p. 95.