Architecture Report

The diagrams mentioned in this report include a large amount of detail, so to ensure readability they have been uploaded separately to our website, and are linked where appropriate in this document.

Concrete Architecture

To develop and build our game we used the Unity game development platform, meaning our architecture follows a Unity framework in order to use the Unity engine as effectively and efficiently as possible. This framework handles core low-level functionality for us, such as basic object behaviour and instantiation, and rendering. Our code is written using C#, one of the languages the Unity platform supports, and so the class diagrams show C# classes.

We have produced 3 different diagrams which we believe show the structure of the code to a sufficient level of detail and understandability. To ensure readability in the first two diagrams, we have omitted the relationship of derivation between each class and the Unity class MonoBehaviour, which allows Start, Awake, Update and other Unity functions to run.

To produce the first and second diagrams of our code, we used 'Software Ideas Modeller'[1], and then manually added the necessary UML notation to the second diagram by examining the structure of our code. The third diagram was made in Microsoft Visio [2] and was produced by examining the structure of our scenes.

The first diagram is purely a UML 2.0 class diagram *(see 'Pure UML' diagram)*, showing the classes and actual C# dependencies between classes. We chose this representation as it is an industry standard for modelling class structure, follows on efficiently from the abstract architecture, and clearly shows the content of our classes. However, we feel this diagram notation is unable to represent the full scope of the code, as many scripts work through Unity GameObjects in the scene. These such classes don't directly reference each other, meaning this relationship isn't represented in a pure C# UML diagram such as this.

To overcome this disadvantage, we have produced a second UML diagram *(see 'Full UML' diagram)* that includes the added dependencies between classes that are created through the Unity framework. It is difficult to directly map UML to these relations, but we believe the meaning still holds with our classes. Composition and aggregation associations can be seen between several classes, and usage/weaker dependency is also shown between some. Most dependencies are one to one, as only one class tends to interact with another object at a given time, for example, a player is only in combat one enemy at a time, although there are some instances of multiplicity. It should be recognised that dependencies between scripts and GameObjects and/or their Unity defined components are not shown, to avoid a very complex diagram and deviation from UML notation.

Finally, we have produced a diagram *(see 'Game Scene' diagram)* that provides an overview of the structure of our code and its deployment in the game, which we were unable to achieve with the aforementioned diagrams. This diagram includes custom notation, designed to show the layout of GameObjects in a typical scene and the World Map, and how the scripts interact with them. Class detail has been omitted, as this can be found in the UML diagrams, while UML dependencies between some classes from the *'Pure UML'* can still be seen, denoted in blue. We defined some custom notation in green, to show GameObjects on the scene, and the scripts that are attached and interact with them. We used a UML composition to represent a script being attached to a GameObject since the GameObject depends on the scripts for its intended behaviour. We use a UML use/general dependency arrow to represent a script manipulating or accessing a property of another GameObject, and we attempt to describe the way this takes place if it is not via an explicit reference assigned in the game editor (for instance, scripts often interact with a GameObject on collision with the game object they are attached to). It is worth noting we do not show components other than our scripts (such as transforms and sprite renderers) that are attached to each GameObject, as we felt the complexity this would add to the diagram would make it less readable and useful.

Justification of Architecture

Through using the Unity software we have gained a greater understanding and knowledge about the software, causing our architecture to evolve in order to support our design and development decisions. Some classes in our concrete architecture are similar to those in the previous abstract version, but some have changed to suit our design and the development platform. Below we present a breakdown of our classes and how, if at all, they build upon abstract classes, in addition to what requirements they relate to.

GameProperties
Relates to requirements: 2.08, 2.10
Relates to abstract classes: Player, Player_Worldmap
This class is responsible for tracking progression through the game. It also runs the DontDestroyOnLoad() function, to ensure it persists through each scene, to preserve this information. This deviated from the abstract architecture, asit contains information originally intended for the abstract *Player* and *Player_Worldmap* classes, as we believe having this information stored in one class rather than across multiple facilities access to this information, and also makes saving and loading simpler for future development.

LevelManager
Relates to requirements: 2.06, 2.10, 3.06
Relates to abstract classes: Level
Building upon our abstract design, this loads levels and the world map, but also manages a loading screen to indicate loading, and updates **GameProperties** with the active level.

WorldMap
Relates to requirements: 2.01, 2.06
Relates to abstract classes: Player_Worldmap, World Map Location
Builds upon *Player_Worldmap*, where levels are entered through calling to **LevelManagement** if unlocked. Nodes are setup during Start(). *World Map Location* was not developed as its own class, as it is simpler to associate nodes on the scene with levels and manage them in one class.

CharSelectManager, PauseManager
Relates to requirements: 2.02, 2.04, 2.05, 3.06.I , 3.07.I
Relates to abstract classes: None
These classes manage the menus for character switching and pause/quit respectively. This did not feature in our abstract design, so these are new additions to the architecture, but we felt they were essential to implement our desired functionality, so it would be remiss to not include them.

PlayerScript, CombatTip
Relates to requirements: 2.03, 2.04, 2.05, 2.07, 3.07.I

<u>Relates to abstract classes: Player, Character Attributes</u>
This class implements the playable character. This includes attacks, movement and character switching. The *Character Attributes* was not implemented as a child class, rather those features were simply included in the **PlayerScript**, a it was more intuitive to implement the player as one entity, and then change player stats and sprites as the character was switched. **CombatTip** does not implement anything in the abstract architecture, but it was needed to handle combat interaction between the player and enemies.

EnemyCombat, EnemyHealth, EnemyMovement, AttackIndicator
<u>Relates to requirements: 2.02. 2.03, 2.07, 3.07.I</u>
<u>Relates to abstract classes: Characters Mov, Enemy</u>
These classes implement the behaviour and components that form an enemy. In combination these classes implement the *Enemy* class from the abstract. The **EnemyMovement** is an implementation of abstract *Characters Mov*, but it is not a parent class, as it useful to separate movement from combat. These are separated to give more flexibility when designing enemies (i.e: a box has with health, but no combat or movement).

BulletController
<u>Relates to requirements: 2.02, 2.03, 2.04, 2.07,</u>
<u>Relates to abstract classes: Bullet</u>
This class handles the movement of projectiles and their damage on collision with an entity. This implemented our abstract *Bullet* class.

MovingPlatform, PassThroughPlatform, HazardScript
<u>Relates to requirements: 2.01, 3.08</u>
<u>Relates to abstract classes: Environment</u>
These classes define behaviours of our environmental objects to enable platforming. This was quite a specific feature, so it didn't feature in the abstract, but it loosely builds on the abstract *Environment* class in that environment objects are implemented.

CauseScript, EffectorScript
<u>Relates to requirements: 2.01, 3.08</u>
<u>Relates to abstract classes: None</u>
These classes implement cause and effect objects in the scene, such as using a switch to toggle on or off an event or hazard. Does not relate to a specific abstract class, but necessary for some of the interactions we wanted in the game.

CameraFollow, CameraMovement
<u>Relates to requirements: 2.01, 3.06.I, 3.07.I, 3.08.Q</u>
<u>Relates to abstract classes: Camera</u>
These classes implement different camera behaviours based on whether the player is in the world map or a combat level. These classes implement the abstract class *Camera,* in that they define its movement through the scene, though the camera itself is a pre-built Unity object.

Additional notes on the architecture:
Singleton classes: GameProperties, CharSelectManager, PauseManager and LevelManager are all singleton classes. This is to simplify access to these back-end managers, of which there is only ever one instance.

Not implemented abstract classes: Some classes in the abstract were not implemented. *Player Minigame, GameOptions* and *Shop Keep* were not considered because it was not within the scope of

Assessment 2. *Background* and *Foreground* are properties of the Unity scene, and so were not something we had to implement as a class, rather they were manipulated in the editor. *Game* functionality is handled by unity also, so we have not implemented our own rendering or game engine function. *Object* is representative of a Unity Object, which MonoBehaviour (and thus our classes) and GameObject (which our scripts are attached to) derive from, so this is accurate to our architecture, but does not need implementation by us. Lastly, *Control* and *Spawner* were not needed, as we have small levels where we manually place enemies for greater design control.

Bibliography

[1] D. Rodrina (2018, Jan, 2) *Agile CASE tool for software design & analysis* [Online] Available: https://www.softwareideas.net/

[2] Microsoft, *Miscrosoft Visio*, 2016 [Online] Available: https://products.office.com/en-gb/visio/flowchart-software?tab=tabs-1