Final Architecture and Traceability Report

A UML diagram showing our final architecture design can be found here.

A full traceability report can be found here.

Fortunately, the game we inherited was an extension of our original game from assessment 2. This allowed us to expand on our architecture easily as the previous team had kept the overall architecture design. Due to the somewhat modular nature of our project, there was no issues in implementing additional requirements on top of the inherited code caused by the handover. As such, some of the classes implemented by the previous team (*MinigameManager*, *Bird* and *MovingPillars*) were untouched. Similarly, many of the existing classes were not changed (see traceability report for changes made during ASM4), which was fantastic in speeding up development as even though our UML diagram was growing extremely large, we were able to focus on a small area of it during this task.

Architecture Justification

Our main goal with assessment 4 was to implement the additional requirements quickly and efficiently in order to maximize the time we could spend on improving the already implemented systems. We decided that the best way for us to meet this goal was to extend the existing architecture in the same manner as previous. This meant that the additional requirements would be implemented in their own classes and subsequently linked into the existing systems, usually controlled by the *Game* class. This led to us updating the *Unit* class and implementing a new class *PunishmentCard*.

Our UML diagram was a very useful tool in helping us develop our game quickly and efficiently. Being able to map out relations between classes ahead of time allowed us to stay focused when unexpected coding challenges had occurred.

New Features

As outlined by the assessment 4 brief, the additional features we had to implement were a postgraduate “leveling” system and also a punishment card system. Luckily, as part of our previous implementation, a leveling system was already in place that could easily be modified to meet the postgraduate criteria. We also had previously created a punishment card system during assessment 3, allowing us to very quickly implement a similar system.

Postgraduate System

Previously, we had implemented a system in which a unit can level up, ranging from leveling 1 to 5. Since this system allowed the unit to become stronger over time, we decided that it already accurately reflected what we would want from a postgraduate “unit”. As such, we decided that when a unit reached level 5, it would become a postgraduate, thus being a stronger overall unit to have.

Punishment Card System

The punishment card system presented some challenges, but as we already had previous experience with a similar system, we were prepared for many of the issues that may have arisen. We decided that punishment cards should appear on the map and be collectable if a

unit enters the corresponding sector. This would add a punishment card to the collecting players “hand”. The player would be able to activate the punishment card during their turn, with a different effect depending on the card they picked up. However, some of our punishment cards effects required the player to be able to select a unit or player for the effect to apply to. This led to a new challenge in that our game had previously had no game state to accommodate such a selection, we had to rework some of the previous game state code in order to add an additional card selection state.

Extra Features

As well as the required new features, we set ourselves an overall task of improving the look of the game, the “pseudo-requirements” we set were unit names, a complete overhaul of unit appearance, a fight animation and we also decided to add a sound system to our game.

For the sounds, a new class *SoundManager* was created. Sounds had to be added during key interactions, so *SoundManager* contains methods to play specific sounds which are then called by other object interactions, such a button presses or dialog events.

Unit appearance was done by adding an additional component to each *Unit*, a *UnitSprite*, that held all of the data referring to the unit’s appearance and the unit’s name. This system pulled from a pool of forenames and surnames as well as a pool of doodle styled model parts (hat, head and body), it then assembles and outputs the unit from the respective parts alongside the full name.

Lastly, the fight animation was implemented as a series of videos attached to a gameObject that were called within the *Sector* Conflict method.

We decided to make changes to the units appearance as we wanted to more accurately capture the overall “doodle” style. We also wanted to try and create a uniqueness to each unit, thus we decided to create a system that would randomly generate both the unit’s name and appearance. Despite not being a core requirement, we wanted to include this aspect of the game as it would not only further solidify upon our theme, but it would also bring a sense of personalisation and attachment to the game and your units/team. Building upon this, the fight animation was implemented as we wanted our game to be more interesting than what the basic requirements detailed and also because conflicts previously had felt quick and anti-climatic.

Our overall goal with the extra changes was to create an attachment between the users and their “player” identity. We wanted to make the user more immersed and invested in the gameplay and hopefully by extension, increase the amount of fun they were having.