Implementation Documentation - Team 14

Babar Khan
George Rogers
Jacob Turner
James Crump
Chloe Remmer
Shijie Lin

1 Relation to Architecture and Requirements

Like the previous team, we used Object Oriented Programming to implement the Concrete Architecture, as it relies heavily on the principle of inheritance. Whilst a lot of the initial classes were already implemented from Assessment 1, we added in a few of our own classes that inherited from the current ones in order to implement new features, such as power-ups.

As the Architecture Diagrams were built from the Requirements, we were capable of implementing all of the new Requirements. The new features we created for these new requirements will be explained below and evidence for completion can be found in the Testing documentation.

2 Significant New Features

As a helper class, we implemented the new class ‘Tuple’ in the Concrete Architecture diagram that is designed to store two different objects. This allows us to create basic pairs of data, which was initially used to save the players boat type and the round they were on in a single JSON class dump. However, as the saving algorithm became more complex, the Tuple class became unused in favor of constructing the JSON string manually.

Below we will break down the new features needed to implement each of the new requirements:

2.1 Power-Ups - Requirement: UR_POWERUPS

Following our Concrete Architecture diagram, we first created a ‘Powerup’ class that inherits from the original Entity class. This class was intended to store the main attributes of a power-up object and the creation of a power-up object easier and less error prone. As we have decided upon a set amount and variation of power-ups (including: ‘speed boost’ and ‘health regen’), we also created a ‘PowerupType’ class that inherits from the Enum class. This enabled us to set predefined power-up types that we can then reuse throughout the game.

As part of the functional requirement, FR_OBSTACLE_POWERUP_RATE, we modified the original ‘Lane’ class to include a field on ‘powerUps’ as well as methods, such as ‘getPowerUps()’. This allowed us to alter the spawn rate for obstacles so 20% of the time a power up is spawned. This is intended to allow for obstacles and powerups to spawn proportionately to each other as difficulty in the race legs increases as well as the difficulty selected by the user. In order to implement the functional requirement FR_POWERUP_EFFECTS, we adapted the code for obstacle collision (because of the similarities in functionality between power ups and obstacles) which allowed us to reuse methods, leading to more concise and modular code. If the user's boat collides with a powerup then the specific powerup is detected using the ‘powerUp.GetType()’ method in a switch statement, which in turn applies the appropriate effect. This change is implemented in the ‘Boats’ class so that it can be used for both players and computer computer boats.

2.2 Save-Reload - Requirement: UR_SAVE_RELOAD

In order to implement the user requirement UR_SAVE_RELOAD, we adapted the previous ‘Settings’ class (now renamed ‘Config’ in the Concrete Architecture diagram) to include a new method, ‘getSaveLocation()’, and field ‘SAVE_FILE_LOCATION’. We included this method in the ‘Config’ class as it is used to implement the initial settings of the gameplay, therefore, it is easier to load the saved settings of a previous game into there if they are a part of the same class, which helped us fulfill the functional requirement: FR_SAVE_RELOAD.

The save file structure is implemented as a JSON file, containing all the necessary attributes required to reconstruct the game in the exact state it was in when the user saved. This allowed us to reduce the problem to adding saving, one class at a time, and then include the generated JSON string in any classes that had an instance of said class. For this same reason, reloading the save is reduced in a similar fashion, with the necessary data being removed in a constructor and the rest being passed down to the constructor of any sub-classes.

2.3 Difficulty - Requirement: UR_DIFFICULTY_BEFORE_GAME

Similarly to ‘UR_SAVE_RELOAD’, the user requirement ‘UR_DIFFICULTY_BEFORE_GAME’ was implemented by adapting the ‘Config’ class to include a new field, ‘GAME_DIFFICULTY’ and method, ‘setGameDifficulty(int)’. As before, by including these new requirements in the ‘Config’ class it is easier to initialise the settings of the game with the difficulty pre-selected by the user. This also helps us to fulfil the functional requirement, ‘FR_DIFFICULTY_SELECTION’.