Implementation

Team 7:
Bermuda Digital Entertainment

Prajwal Binnamangala
Joseph Sisson
Sebastian Sobczyk
Aymeric Goransson
David Ademola
CJ Donoghue

As can be seen from our Architecture document, the game had a solid foundation to build on with the class ***GameObject*** as the building block of all objects in the game. We did not change the class itself significantly, however it has provided us with great tools to expand the game further. The ***draw()*** function is a perfect example of this, as it supplies a lot of functionality that can be cherry-picked for different purposes.

An example of this is the difference between the unchanged draw function in the ***Player*** class and the draw functions of the ***Obstacle, PowerUp*** and ***Boat*** classes. These were created to satisfy the requirements **UR.OBSTACLES, UR.POWER_UP** and **UR.ENEMY_SHIP** respectively. Since these objects are simpler than the ***Player*** object, their drawing functions can be stripped of excess detail.

We found this starting Architecture very clean, intuitive, and practical so we decided to build the software using this approach. As can be seen from the Architecture diagram, a total of 8 classes extend the ***GameObject*** class (***Boat, College, HealthBar, Indicator, Obstacle, Player, PowerUp and Projectile***).

We decided to do this kind of approach for all the different screens that are part of the software. This has helped us to keep the code modular in line with our requirement **NFR.MODULARISE**. As further seen from our Abstract and Concrete architecture diagrams, all 8 screen classes (***DifficultyScreen, EndScreen, GameScreen, PauseScreen, SaveScreen, ShopScreen, TipsScreen and TitleScreen***) extend the libGdx's ***ScreenAdapter*** class. Almost all also use ***Stage*** object class as their main building block apart from ***GameScreen*** which heavily relies on the ***SpriteBatch*** class to display all the gameplay graphics.

We took care not to include any function definitions within the screen building section of the classes and keep them all separate at the bottom of the class file. An example of this can be the ***DifficultyScreen*** class that deals with changing various variables depending on the chosen difficulty. The functionality of the code is kept separate from the building of the stage and the screen and the buttons are used to call on those separate functions. This keeps the code modular and makes it possible to test the functions.

We decided to leave the ***HUD*** and ***YorkPirates*** classes almost untouched as we felt their functionality was brilliant and didn't need changing aside from adding the shop menu ***HUD*** and changing minor camera settings in the ***YorkPirates***.

Undoubtedly though, the prime example of our modular coding approach is the ***GameScreen*** class. We chose to initialise almost all game graphics in the initialisation function of this class as this is a watertight approach to minimise the risk of any memory leaks occurring during the rendering of the game's frames.

The only part of the code that does not conform to the structural template we have decided on is the code designed to satisfy the requirement **UR.BAD_WEATHER** and rendering of the fog graphic. Since the fog is a dynamic event within the game with a unique effect and functionality, we decided on a dynamic implementation in the code itself. Similarly to other assets, the ***Sprite*** for the fog is initialised in the ***GameScreen*** class, however the fog itself does not belong to a specific class of the ***YorkPirates*** package. Instead, the fog's dynamic functionality is entirely enclosed within the ***render()*** method of the ***GameScreen*** *(figure 1)*.

The code snippet works with a set of conditions that change during gameplay to spawn the fog effect at pseudo-random intervals for a pseudo-random amount of time (random within limits)

```
fog.setX(player.x-(fog.getWidth()/2));
fog.setY(player.y-(fog.getHeight()/2));

if(fogDecider == 0){
    fogDecider = MathUtils.random( start: 30, end: 60);
}
else{assert true;}

if(Instant.now().getEpochSecond() - startTimeStamp > fogDecider){
    fog.draw(game.batch, alphaModulation: 0.8f);
    if(counterStarted==false){
        fogCounter = Instant.now().getEpochSecond();
        counterStarted = true;
        fogLengthDecider = MathUtils.random( start: 15, end: 30);
    }
    else{
        if( Instant.now().getEpochSecond() - fogCounter > fogLengthDecider){
            startTimeStamp = Instant.now().getEpochSecond();
            fog.setFlip( x: true, y: false);
            counterStarted = false;
            fogDecider = 0;
            fogLengthDecider = 0;
        }
        else{assert true;}
    }
}
else{assert true;}
```

Figure 1

The non-trivial new features that we have added to the software are:

- ***Boat*** class. Entirely new class that extends the ***GameObject*** class that is responsible for creating enemy ships and controlling their behaviour (movement, projectiles, collisions etc.) (Requirement: **UR.ENEMY_SHIP)**

- ***DifficultyScreen*** class. Deals with creating the screen for choosing the difficulty for the playthrough. Holds functions that change variables used for game mechanics according to chosen difficulty. (Requirement: **FR.DIFFICULTY_LV**)

- ***Obstacle*** class**.** A simple extension of ***GameObject*** that creates obstacle objects and holds information regarding their effect on the player. (Requirement: **UR.OBSTACLES**)

- ***PowerUp*** class. A very similar class to ***Obstacle*** responsible for the functionality of power ups that have been added to the game. (Requirement: **UR.POWER_UP**)

- ***SaveLoad*** class. Holds the code responsible for creating the save file JSON and functions that allow saving ***GameObject*** variables etc. (Requirement: **UR.SAVE_GAME**)

- ***SaveScreem*** class. Provides the GUI for the game saving and loading functionality. (Requirement: **UR.SAVE_GAME**

- ***ShopScreen*** class. Builds the GUI for the game's shop and adds the shop's back-end functionality that builds on functions from other classes, i.e. boosting player's health with its own function that calls a function from the ***Player*** class.(Requirement: **UR.SHOP**)

- ***TipsScreen*** class. A very simple screen class that displays necessary information for the player about the game's environment.(Requirement: **UR.HELP**)

Where any graphical assets were involved and/or created, we have kept true to the initial look and feel of the game as we deemed it to be very clear and satisfying for the user. The screens were made mostly using existing skin and sprites so that they do not feel out of place when transitioning between one screen and another.

The functionality of ***PowerUp*** and ***Obstacle*** objects has been designed similarly to that of the fog graphic in terms of the effects they have on the player. All necessary information is held within the object itself for the purpose of the game save, however the effect on the player is decided at the time of impact in-game through the render method and the ***entity.power*** variable.

We have also added some new functionalities to previously existing classes, such as setting the health of the player, healing the player, altering the player's speed etc. all done through the ***Player*** class.

To meet the **UR.ENEMY_SHIP** requirement, we have changed the functionality of the ***College*** class as previously the class spawned idle ships that did not interact with the player or player's projectiles. Any code related to spawning idle ships has been removed from the ***College*** class and the new class ***Boat*** was added instead. The constructor of the class is very similar to the ***College*** class but takes a different approach for health tracking and health bar drawing as well as calculates collisions differently. The enemy ships are designed to follow the player once in range and act as additional obstacles even when destroyed.

Another systemic change that we decided on was changing the zoom of the main game camera to give the player a wider field of view. This was also done for the broader aim of unlocking the game screen size and being able to work with the resizing of the window. The project we took over did not have any provisions for keeping the game functional when the game window was resized. This resulted in graphics stretching, on-screen buttons not working and the camera de-syncing every time the window was resized. We have tackled this issue by changing the Viewport of the ***HUD*** class to a ***FitViewport*** and adding overriden ***resize()*** functions that update the camera and the ***HUD*** Viewport.

We have also removed all code associated with playing music in the game as we have been instructed by the customer that the game should have no music due to the nature of the setting it will be showcased in.

Another feature that has been removed were the shaders responsible for hit animations on colleges. While we understand that this may be considered to have hindered the overall quality of the game, this change was necessary as it interfered with the implementation of the unit tests and meeting the **NFR.UNITTEST** requirement. This is further justified by the fact that hit registering on the player, colleges and enemy ships is made obvious to the user by visual representation of taking damage (decreasing health bars on all entities) and the projectiles disappearing upon impact. We also felt that the shaders and flickering did not go in line with the simple visuals of the game.

All the features required for the purpose of the Assessment 2 have been added in our implementation. To summarise:

- Ways to spend plunder (**UR.SHOP –** (mainly) ***ShopScreen*** class)
- Combat with enemy ships (**UR.ENEMY_SHIP –** (mainly) ***Boat*** class)
- Addition of bad weather (**UR.BAD_WEATHER** – code in Figure 1 of this document)
- Addition of obstacles (**UR.OBSTACLES** – ***Obstacle*** and code in ***GameScreen render()***)
- Five special power ups (**UR.POWER_UP** – ***PowerUp*** and code in ***GameScreen render()***)
- Support for different levels of difficulty (**UR.DIFFICULTY_LV** – ***DifficultyScreen***)
- Save and Load facilities (**UR.SAVE_GAME** – ***SaveLoad*** and ***SaveScreen***)