Implementation

We have implemented all the new feature requirements. Most of the new requirements are feature additions, and therefore do not require any significant/major architectural changes.

General

The codebase is modified stylistically (whitespaces, new lines) to conform with the style checks we have employed in CI.

Power-up pick-ups

In order to fulfill the requirement UR9 and as according to the planned architecture, we've made alterations to the ObstacleType and Boat classes. In the software given by the previous group, obstacles could be one of three different types, each doing different damage amounts to the player and having a unique speed.

In our new program, we have added five new options to the ObstacleType enum. The difference is that these new "obstacles" do zero damage to the boats and can be easily identified as power-ups by their different textures. We then altered the "checkCollisions" function in the class Boat. Now, when a boat collides with an obstacle, the game checks what kind it is, obstacle or power-up with a switch statement. Then in the case that it is not a proper obstacle, and the player has no shield, it will apply the damage and speed decrease. Otherwise the program will apply the corresponding effect.

Difficulty Selection

As was planned out in the concrete architecture, a new class named DifficultySelectScreen has been added which implements the libGDX screen class. This class acts in a similar manner to the main menu and boat selection screens. It contains a set of buttons, each of which takes the user to a different screen.

In the previous software, the BoatSelectScreen passed an enum "BoatType" that would be then used in the following screen (MainGameScreen), but with the new requirement UR11 to allow the user to select a difficulty, the BoatType enum will reflect both the type of boat and the difficulty chosen. Therefore, we don't want to pass an enum after the BoatSelectScreen. The subsequent code we implemented instead passes a string called "type" with values such as "fast", "endurance", "strong" and "agile" through to the DifficultySelectScreen.

It is then in this class that the difficulty is chosen and the combination of that with the type can be used to pass a BoatType enum to the MainGameScreen just as in the previous implementation. The difference being that the BoatType enum now features 12 possible enumerations which are combinations of boat type and difficulty (i.e. FASTEASY, AGILEHARD).

Choosing easy difficulty will result in the player's boat having increased speed, stamina, health and agility. The values for each boat type is increased by 20%, whereas on hard difficulty, each type of boat has it's values decreased by 20% instead in order to make the game harder for the player to finish in a high position and subsequently to qualify for the final round.

Save/Restore

With the save/restore feature as laid out in UR14 and UR15, we have decided to use JSON as the game state file format. It is human-readable, and easy to manipulate, which made debugging easier.

By using Gson, we are able to serialise the MainGameScreen class, which is the state of the game. We have modified most of the classes, with an extra addition of annotation @Expose on most of the properties. With the annotation, it allows us to only record the essential state properties, and keep the file size small.
In addition to the extra annotation, most serialisable classes now implement the PostProcessable interface, which has a function called postProcess. This function is called when objects are deserialized.

Most of the Save/Restore logic resides in the tools.state package, specifically the SaveRestore class that was designed in the architecture. Only three methods are of public API: Save, Restore, and slotIsAvailable. (Other methods are of internal organisation, including file flush, getting a Gson instance, etc.) These public methods are used in the main game. No save/restore logic resides into the game itself.

There are a couple LibGdx classes that require custom serialisation, namely Vector2 and Hitbox. Adapters are created for auxilarating the gson library.

One of the changes we made to the previous codebase is the way the leaderboard is calculated. Previously, the leaderboard relied on using the class property boat, which has the type Boat.
The class Boat is an abstract class, therefore it cannot be instantiated. When restoring (deserializing) the Race class, the boat objects in it were abstract, and the JSON library had to instantiate/restore all the boats, which was not possible, and crashed the game.
To fix this problem, we investigated and modified the way the leaderboard rankings are calculated, and modified the class of the boat property such that it is now of ComputerBoat type (computerBoat). We suspect that this is an oversight of the previous team as the javadocs comments do not indicate clearly what the variable actually was.

In the aspect of the user, we have added the ability to pause the game, and let the user save/overwrite the game state (via keyboard keys). Also, in the main menu screen, we have added a new button for the user to access the state restoration screen.