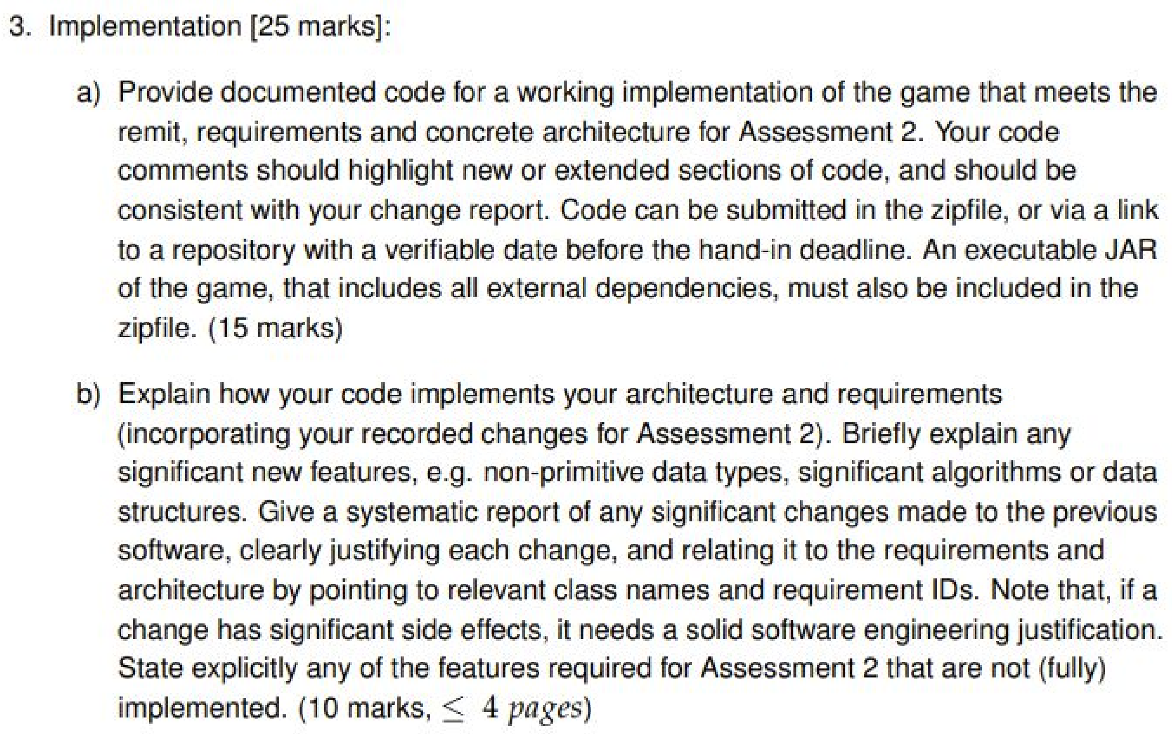
Module	Engineering1 (Eng1) - COM00019
Assessment Title	Assessment 2, Cohort 2
Team	Dragon Boat Z (Team 18)
Members	Robert Dalgleish, Benjamin Jenner, Joseph Lonsdale, Richard Upton, James Wilkinson, Xinyi Zhang
Deliverable	Implementation



3. Implementation [25 marks]:

    a) Provide documented code for a working implementation of the game that meets the remit, requirements and concrete architecture for Assessment 2. Your code comments should highlight new or extended sections of code, and should be consistent with your change report. Code can be submitted in the zipfile, or via a link to a repository with a verifiable date before the hand-in deadline. An executable JAR of the game, that includes all external dependencies, must also be included in the zipfile. (15 marks)

    b) Explain how your code implements your architecture and requirements (incorporating your recorded changes for Assessment 2). Briefly explain any significant new features, e.g. non-primitive data types, significant algorithms or data structures. Give a systematic report of any significant changes made to the previous software, clearly justifying each change, and relating it to the requirements and architecture by pointing to relevant class names and requirement IDs. Note that, if a change has significant side effects, it needs a solid software engineering justification. State explicitly any of the features required for Assessment 2 that are not (fully) implemented. (10 marks, $\leq\ 4\ pages$)

<u>How the code implements Architecture and Requirements</u>

- Requirements are met by the Architecture.
- Architecture uses OOP to…
- Each class designed in the Architecture belongs to its own .java file

<u>Explanation of significant new features</u>

- Data types, algorithms, etc.
- Changing any hard-coding?
- (Brief)

<u>Systematic report of significant changes</u>

- Justify each change and relate it to requirements and architecture by pointing to class names and ID.

<u>Features not fully implemented</u>

- ?

How the code implements Architecture and Requirements

New Requirements were established from the Assessment 2 brief. These then flowed into amendments to the Abstract and Concrete architectures. Changes to the Abstract Activity Diagram were implemented in the code as new algorithms or game states, such as pauseState.

Changes to the Abstract and Concrete UML Class Diagrams were implemented as new classes, eg, the PowerUp class. Every new class designed in the Architecture has its own .java file in the implementation. These new files, their corresponding features and the requirements behind each new feature are explained below.

Explanation of significant new features and systematic report of significant changes

1) POWER-UPS

Classes created for this feature: PowerUp, PowerUpBomb, PowerUpHealth, PowerUpInvulnerability, PowerUpSpeed, PowerUpStamina.

Classes updated for this feature: Game, Boat, Obstacle.

Justification for change: to meet the following Requirements: UR_POWER_UPS 1.3.3, FR_POWER_UPS 2.0.15.

We settled on having 5 power-ups in the game:

- **Bomb power-up**: destroys the next two upcoming Obstacles in your lane.
    - As Obstacles stored in a Lane object are not necessarily ordered by y-value, the first step in determining which Obstacles need clearing is to sort the Obstacles in the Lane. We used Insertion Sort to do this, using an ArrayList and inserting each Obstacle in the Lane into it at the right index. We also skip Obstacles that are actually PowerUps, by checking against the method "isPowerUp()".
    - With Obstacles sorted by their y-value, we can determine which ones are the two closest to the Boat and still ahead of it, by comparing the y-value to the Boat's. The algorithm selects the first Obstacle to exceed the Boat's y-value, and the next Obstacle in the ordered ArrayList.
    - Finally, to actually meet the requirement and destroy the Obstacles, we need to remove them from the screen. The algorithm gets the Bodies of these two Obstacles and adds them to an ArrayList called "toBeRemovedBodies" which can be found in the "Game" class. In "Game", there is a step in the main loop in which all Obstacles with a corresponding Body in "toBeRemovedBodies" are removed from the screen.
- **Health power-up**: increases the robustness of your Boat.
    - As the method "applyPowerUp()" takes a Boat as a parameter, updating the robustness of the Boat is not an issue.
    - However, we didn't want this power up to make your robustness exceed your max stats, so to do that we need to compare the boat to "boatStats" in the "GameData" class. We can only do this if we get the Boat's type and use this as an index to compare the robustness stat in "boatStats" to the updated value, and reduce it if it exceeds the maximum.
- **Invulnerability power-up:** makes your Boat immune to damage and speed reduction when hitting Obstacles, for 7 seconds.

    - To make the Boat invulnerable, we simply set the Boat to be invulnerable with the method “setInvulnerability(true)”. The algorithm for damaging the boat needed updating to check whether the Boat was invulnerable at the time of the collision, and not damaging it or slowing it down if it was.
    - The trickier problem was getting the Boat to only be invulnerable temporarily. The method “applyPowerUp()” is resolved instantly when it gets called in the main game loop, and so we needed Boat objects to have a timer in them for any power ups that were based on a timer.
    - In the “Game” class, wherever the variable “currentTimer” is updated, we also update the “powerupTimer” attribute of all the Boats in the game. Whenever one of these timers hits 0, we set that Boat’s invulnerability back to false.
- **Speed power-up:** makes your Boat move considerably faster for 2 seconds.
    - This power-up works in a similar way to Invulnerability. Increasing the current and max speed of the Boat was simple as the Boat gets passed as a parameter, however, a timer needed setting, and the current and max speed values needed resetting when it hits 0.
    - So whenever a Boat’s “powerupTimer” attribute hits 0, it now also sets its max speed back to its default values, using the Boat’s type as an index in “GameData.boatStats”. If the current speed now exceeds the default max speed, we need to slow the Boat back down.
- **Stamina power-up:** replenishes part of your stamina.
    - This power-up takes the Boat as a parameter and increases its stamina by 1.5 times. Boats do not have a maximum stamina, so this isn’t considered in the algorithm.

2) DIFFICULTY

Classes created for this feature: None.

Classes updated for this feature: ChoosingUI, GameData, Game.

Justification for change: to meet the following Requirements: UR_SELECT_DIFFICULTY 1.2.7, UR_DIFFICULTY_LEVELS 1.3.2, FR_DIFFICULTY_LEVELS 2.0.14, FR_MAIN_MENU 2.0.19.

We decided upon 3 levels of difficulty: Easy, Normal, and Hard, which is stored as a single integer “difficultySelected” in “GameData”, ranging from 0 to 2, from easiest to hardest.

We chose an integer to represent Difficulty so its value could be used in calculations, rather than have lots of branching “if” statements based on the difficulty the player had chosen. For example, we use this value as a multiplier when determining the number of Obstacles that should appear in a Lane in a given leg. It is also used in setting the probability of Power Ups being dropped: 20% drop rate on Easy, and 10% drop rate on Hard.

The player chooses the difficulty level in the “ChoosingUI” class, after the player has chosen which Boat type they want to play as. This was done so that we wouldn’t need to make a new UI class for difficulty selection, given this decision is still within the remit of “ChoosingUI”.

3) PAUSING

Classes created for this feature: PauseUI.

Classes updated for this feature: GameData, Game.

Justification for change: to meet the following Requirements: UR_PAUSE_GAME 1.2.4, FR_PAUSE 2.0.12.

The game needed to be pausable from any game state. As such, a new “pauseState” needed declaring to tell the game that it’s paused; this was put in GameData alongside all the other game state variables.

To ensure the game could be paused from any game state, it needed to remember which state it was in previously, so the variable “previousState” was created to tell the game to which state it should return when the player exits the Pause Screen.

In the “Game” class, the program listens for the user inputting the ESC key in order to set “currentUI” to a new instance of PauseUI. The newly created “PauseUI” class offers the player the option to resume, save the game, or exit to the main menu.

Resuming (or pressing ESC again from the pause state), will set the current UI back to the game state stored in “previousState”. Saving will set the current UI to SaveUI, and exiting will set it to MenuUI.

4) SAVING & LOADING

Classes created for this feature: SaveUI.

Classes updated for this feature: GameData, Boat, Lane, Map, MenuUI.

Justification for change: to meet the following Requirements: UR_SAVE_GAME 1.2.5, FR_SAVE_SLOTS 2.0.11, FR_MAIN_MENU 2.0.19.

In order to save and load a game, the methods “saveGame(slot, map)” and “loadGame(slot)” were defined in “GameData”.

The “saveGame” method uses Google’s GSON library to serialise Java objects into JSON objects. Serialising primitive variables such as “currentLeg”, “currentTimer” and “difficultySelected” can already be done using GSON, but for our custom data types such as Boat and Map, custom serializers needed defining.

For the Boat serializer, GSON serialises the Lane number, robustness, maneuverability, max speed, acceleration, stamina, current speed, turning speed, angle at which it is turning, Boat type, X position and Y position into JsonPrimitives. For the Map serializer, GSON serialises (for every Lane) the number of Obstacles in it, and for every Obstacle, its X position, Y position, and Obstacle type.

Using GSON, we then write the serialised Boats, game data, and Map to a file “save_data/save_state_*x*.json” where *x* is the save slot selected.

The “loadGame” method reads the “save_data/save_state_*x*.json” made by “saveGame" and gets the Boats, game data, and Map as either JsonArrays or JsonObjects. Game data then gets split down into currentTimer, currentLeg and difficultySelected. For every Boat in the JsonArray, a Player or AI Boat is initialised.

5) OTHER SIGNIFICANT CHANGES

Undoing hard-coding

- In the original codebase, lots of important values, such as the number of boats, obstacles, legs, etc. were all hard-coded into wherever they were used. Specifically, there were 4 boats, 3 legs, and 30~35 obstacles per Lane.
- So, whenever there was a loop regarding the number of legs, the value 3 was hard-coded into the “for” loop. The value 3 was also hard-coded into any “for” loops looping over the opponents.
- Considering that our requirements for the game necessitated all of these values being changed, we decided to store the number of boats, finalists, legs, etc. in GameData and refer to that class from the loops they were needed in.
- This allows for any changes to these values to not be as tedious to implement as they were the first time around.

Features required that are not fully implemented

Power-Ups

- After trying to debug “PowerUpBomb”, we ran out of time trying to understand what was going wrong in the removal of Obstacles.
- The overridden method “applyPowerUp” does in fact result in a list of Obstacle bodies that should be being added to “toBeRemovedBodies” in the “Game” class, however when we play test it, these Obstacle bodies don’t seem to be being cleared.

Saving/Loading

- Saving (or loading) is not possible from the “ResultsUI” state.
- Loading doesn’t always work properly when loading a save in a newly opened game. For example, loading sometimes won’t load the Boats in with the correct rotations.

Balancing

- The different boat stats have not been thoroughly balanced.
- Also, the increase in stats of the AI boats is not enough to counteract the increase in difficulty from extra Obstacles, so AI boats perform worse on higher difficulties, which is counterintuitive.

Animations

- Only the Boats are animated. Geese were also supposed to be animated, but we never got around to implementing this.

Game Over screen

- The Game Over screen doesn’t work perfectly. When you get a Game Over after not qualifying for the final, the game closes.
- Also, the output of the Game Over screen (ie 1st, 2nd, 3rd, or Game Over) depends on when you crossed the finish line, regardless of any penalties you accumulated that might reduce your standings.