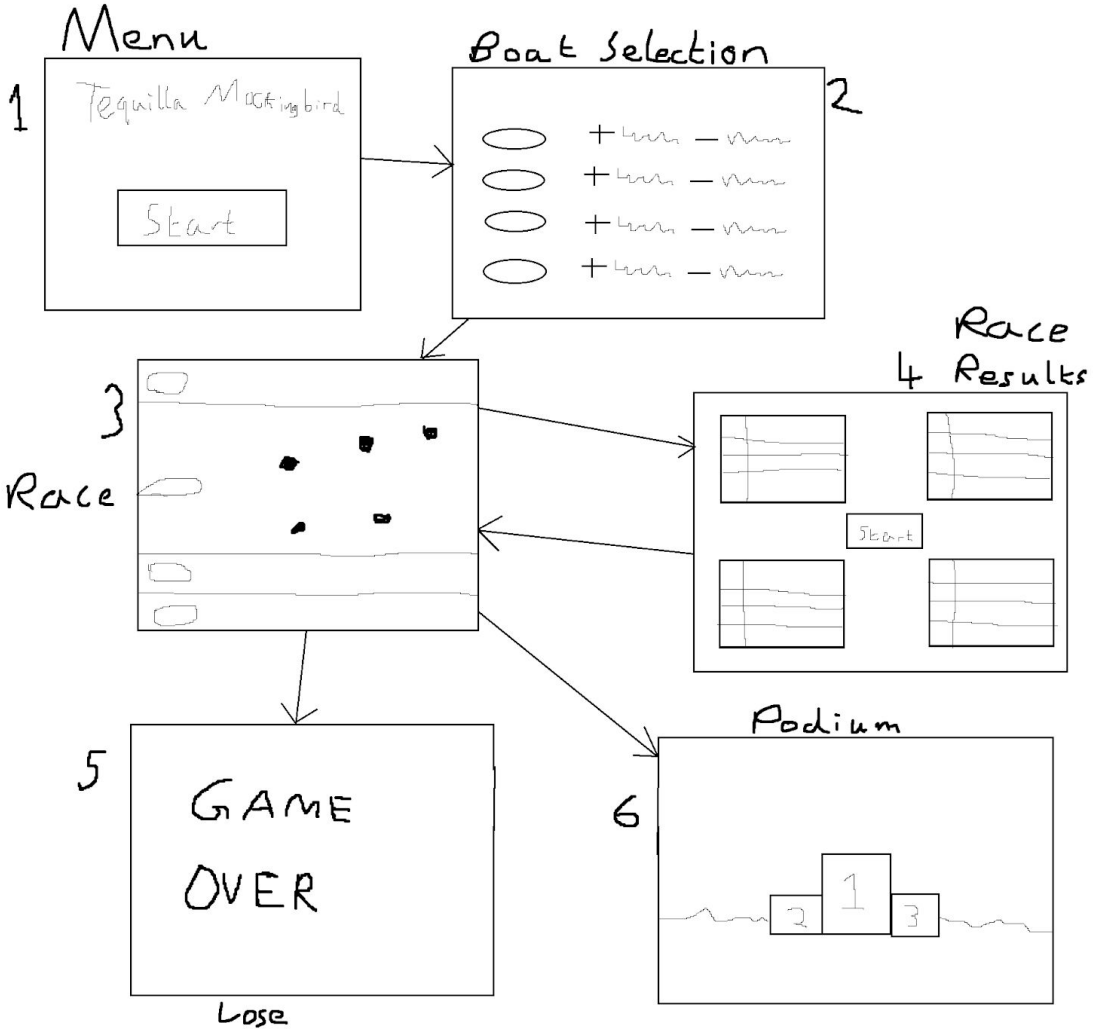
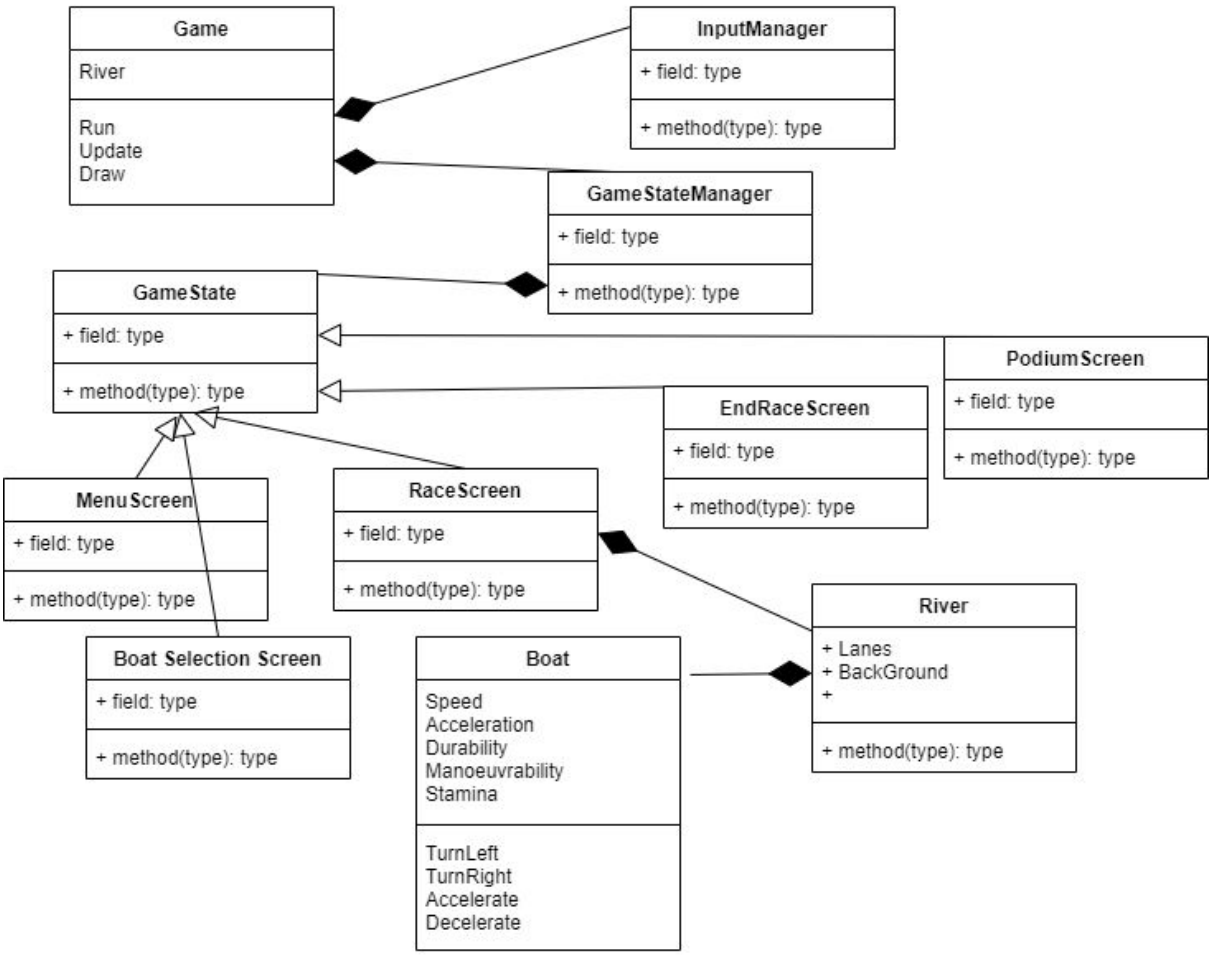
3. Architecture of the program

a. Abstract and concrete architecture representation
(Image 1)

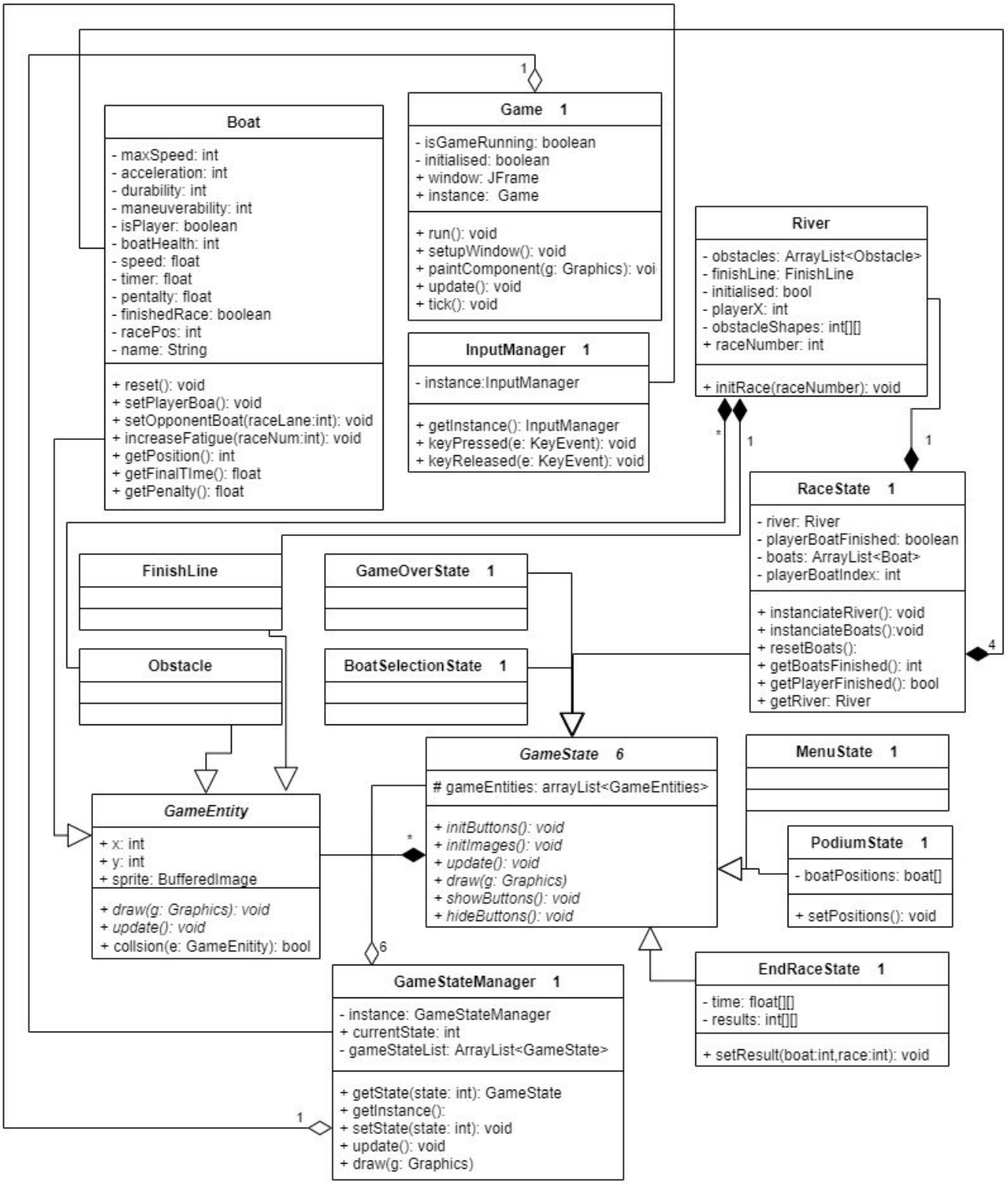


1. Game opens on a menu screen with a start button. Clicking the button takes the player to the boat selection screen.
2. The boat selection screen shows the stats of all the possible boats the player can pick from highlighting the positives and negatives of each boat they can use. Clicking on a boat takes the player to the race.
3. On the race screen the player can either succeed and go to the race results screen or hit too many objects and break their boat, taking them to the game over screen.
4. From the results screen the player can view the results of all the races so far and click a button that will take them to the next race.
5. Repeat step 3 and 4 until the player has completed a total of 4 races. If the player makes it to the end of the 4th and final race successfully then do not go to the results screen but instead go to the podium screen and display the results of the final race.

(image 2)


Game
River
Run
Update
Draw
InputManager
+ field: type
+ method(type): type
GameStateManager
+ field: type
+ method(type): type
GameState
+ field: type
+ method(type): type
PodiumScreen
+ field: type
+ method(type): type
EndRaceScreen
+ field: type
+ method(type): type
MenuScreen
+ field: type
+ method(type): type
RaceScreen
+ field: type
+ method(type): type
River
+ Lanes
+ BackGround
+
+ method(type): type
Boat Selection Screen
+ field: type
+ method(type): type
Boat
Speed
Acceleration
Durability
Manoeuvrability
Stamina
TurnLeft
TurnRight
Accelerate
Decelerate

(image 3)


Boat
- maxSpeed: int
- acceleration: int
- durability: int
- maneuverability: int
- isPlayer: boolean
- boatHealth: int
- speed: float
- timer: float
- pentalty: float
- finishedRace: boolean
- racePos: int
- name: String
+ reset(): void
+ setPlayerBoa(): void
+ setOpponentBoat(raceLane:int): void
+ increaseFatigue(raceNum:int): void
+ getPosition(): int
+ getFinalTIme(): float
+ getPenalty(): float
Game 1
- isGameRunning: boolean
- initialised: boolean
+ window: JFrame
+ instance: Game
+ run(): void
+ setupWindow(): void
+ paintComponent(g: Graphics): voi
+ update(): void
+ tick(): void
InputManager 1
- instance:InputManager
+ getInstance(): InputManager
+ keyPressed(e: KeyEvent): void
+ keyReleased(e: KeyEvent): void
River
- obstacles: ArrayList<Obstacle>
- finishLine: FinishLine
- initialised: bool
- playerX: int
- obstacleShapes: int[][]
+ raceNumber: int
+ initRace(raceNumber): void
RaceState 1
- river: River
- playerBoatFinished: boolean
- boats: ArrayList<Boat>
- playerBoatIndex: int
+ instanciateRiver(): void
+ instanciateBoats():void
+ resetBoats():
+ getBoatsFinished(): int
+ getPlayerFinished(): bool
+ getRiver: River
FinishLine
GameOverState 1
Obstacle
BoatSelectionState 1
GameState 6
gameEntities: arrayList<GameEntities>
+ initButtons(): void
+ initImages(): void
+ update(): void
+ draw(g: Graphics)
+ showButtons(): void
+ hideButtons(): void
MenuState 1
PodiumState 1
- boatPositions: boat[]
+ setPositions(): void
GameEntity
+ x: int
+ y: int
+ sprite: BufferedImage
+ draw(g: Graphics): void
+ update(): void
+ collsion(e: GameEnitity): bool
EndRaceState 1
- time: float[][]
- results: int[][]
+ setResult(boat:int,race:int): void
GameStateManager 1
- instance: GameStateManager
+ currentState: int
- gameStateList: ArrayList<GameState>
+ getState(state: int): GameState
+ getInstance():
+ setState(state: int): void
+ update(): void
+ draw(g: Graphics)
1
*
1
1
4
*
6
1

b. justification for the architectures and how they relate to the requirements

We created an abstract architecture through storyboarding in an early team meeting. We drew a simple mockup of the flow of the program by using a flowchart between each “screen” that the player will see and interact with, using the requirements as a guide to ensure the architecture that we were aiming to implement was capable of delivering the product that client was expecting. The basic principles we decided on in this abstract stage focused the team on a single construct that was a cumulative work and collection of ideas from all the members. See the design we came up with in image 1.

Using this abstract plan we created a sketch UML diagram of the classes that we intended to create. For this process we used a free online drawing tool called draw.io facilitating the creation of tables, arrows and text boxes. This diagram had many iterations throughout the engineering process that we used and altered as the design of the product progressed. The first iteration of our UML diagram omitted many details that were required for a complete and thorough design, yet gave us a starting point to begin conceptualising the basis of the program and allowed the team to begin working on building a code base. Because of the omitted details this was still a relatively abstract architecture, but as we were still deciding how we wanted to make most of the program work we were more than okay with working with it. Some details of the architecture were decided during this process of transferring from the abstract design to a UML sketch are:

- The program's function would be split onto a list of states. Each state will have its own draw and update functions. All the different states will be held in a game state manager that will control the changing between the states.
- An input manager using key listeners will take all the input from the board for the game.
- Each game state will have a list of JButtons that use action listeners to program the onclick events in the game.
- All pictures to be drawn to the screen will be stored as a BufferedImage and all images for the JButtons will be stored as an ImageIcon.
- The computation for the race, including obstacles and finish line, will be stored in a river class that will be reset at the start of every race.

See image 2

After beginning to create the game we came across many fundamental concepts that we realised needed to be a part of the programs architecture for the game to function as intended such as:

- We wanted the GameStateManager to be a singleton class, allowing us to call functions from it from whatever state the game was actually in, without having to directly pass an instance of the class into the states themselves at any point.
- We wanted the GameState class to be abstract.
- We wanted an abstract class “Entity” that will be able to handle the position, size, sprite and collision of anything that was involved with the actual game. This would simplify the collision handling of the obstacles, finish line and boats.

Etc. This led us to the final, concrete architecture being decided upon that we would work off of for the remainder of the project. We only deviated slightly from this in occasions where

there was an oversight in how we needed to program the implementation of the architecture. See image 3.

The architecture was planned around meeting all the requirements planned out it Req1:

ID	Description	How Requirement is managed in architecture
Boat_Select	The user must be able to pick a boat from a selection at the start of the competition	Handled by BoatSelectionState
Boat_ Control	The user must be able to control the boat during the race	Handled using KeyListeners in the InputManager, information on KeyEvents passed to game states
Unique_ Stats	Each boat must have unique stats, including speed, durability, acceleration, and maneuverability.	Boats have a set of stats that can be changed in them, boat selection state will allow the user to pick the set of stats they want.
River_ Obstacles	The river that the boat drives in must contain obstacles for the user to avoid.	River will generate an ArrayList of obstacles that can be fetched in the RaceState
Time_ Penalty	The user must get a time penalty for hitting obstacles or going into other lanes	Collision control has been minimised into one function by having an abstract class for all entities. Time penalties for being out of the lane will be decided by the boat's position.
Obstacle_ Collision	Hitting an obstacle will reduce the boat’s robustness	All collision control has been minimised into one function by having an abstract class for all entities.
Rower_ Fatigue	Over the course of the race the boats will lose stats to simulate fatigue	When the River class is reset, it will be reassigned a new race number that will degrade the boat's stats.
2D	The game must be a 2D graphical game	2D images will be handled as BufferedImages and stored in a GameState that will draw them to the screen.
Java	The game must be written in Java	-
End_ Podium	The end screen must have a podium showcasing the top 3 boats	The podium screen will handle the collection of the winners results and the drawing of the winners on the podium.
Music	The game can contain music	Music has been omitted from the design as it not a necessary requirement and will require extra work
Time_Frame	The game must last around 5 minutes total, with each leg being around 1 minute	Length of the race will be decided by the position of the FinishLine in the River class
Difficulty_ Increase	Each leg will increase in difficulty	By decreasing the maneuverability stat and increasing the number of obstacles each race the game will get harder as the user progresses