Assessment: 1

Deliverable: Architecture

Team Name: Team 8

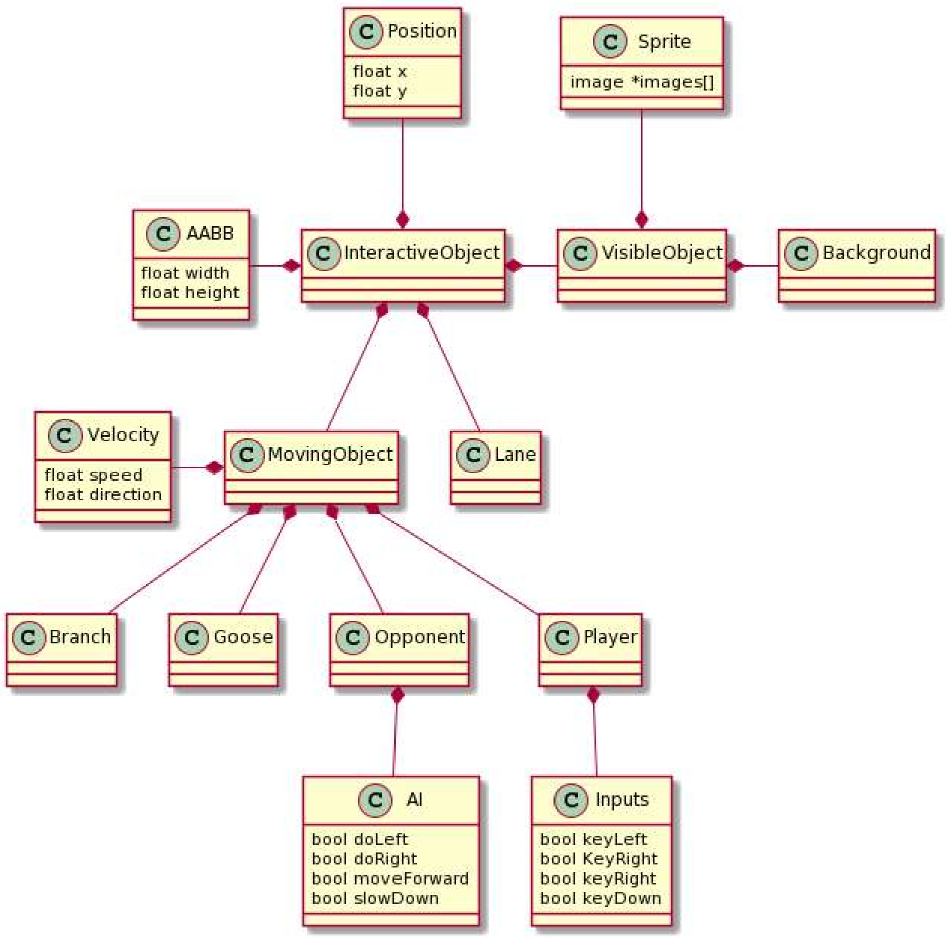
Team members: Charlie Hayes, Matilda Garcia, Joshua Stafford, David Kayode, Ionut Manasia, Matthew Tomlinson

Architecture (a)

Introduction

Both the abstract and concrete architecture were created using the Unified Modeling Language (UML). UML, is a standardized modeling language consisting of an integrated set of diagrams. It “was created to forge a common, semantically and syntactically rich visual modeling language for the architecture, design, and implementation of complex software systems both structurally and behaviorally” [1]. Describing the architecture using UML allowed our team to have consistent and readable diagrams to refer back to when designing, creating and maintaining the system. It also means that future developers or maintainers of our system will be able to recognise the standardised notation of our architecture and understand the system.

Abstract Architecture



The abstract diagram was modeled using the Entity-component-system (ECS) architectural pattern. The entity-component-system favors composition over inheritance, breaking down each entity into common components allowing for greater flexibility while trying to implement and delete different conceptual parts of our game. We have also chosen to group entities into super classes, allowing for enhanced readability, as we avoid having too many relations in the actual diagram. The diagram was created using plantUML, an open source software for modeling UML diagrams from plain text, and its main parts are:

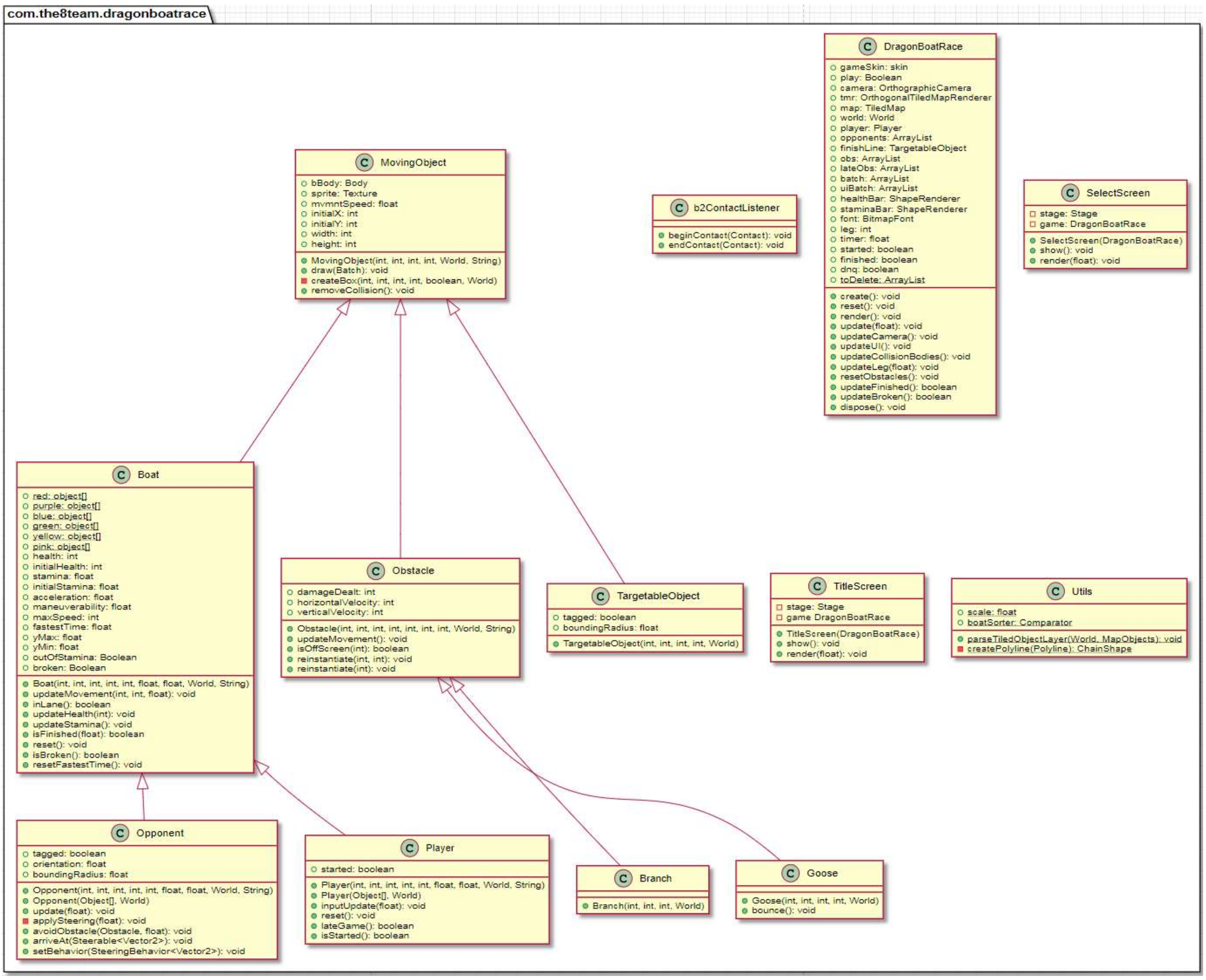
- **VisibleObject**, an interface for entities that will be rendered in the screen using:

    - **Sprite** class, a basic visual representation of the object on the screen, it is used by every entity
- **Background**, as the background of our game
- **InteractiveObject**, an interface for the entities that will be able to interact with each other in the game, using:
    - **AABB**, a class for a collision component, setting the boundary of a given object
    - **Position**, a class holding the coordinates of an object in our 2d world
- **Lane**, an entity used to represent the delimitations between the lanes were the race would take place
- **MovingObject**, another interface for an object that would be able to move on the screen relative to the world, using
    - **Velocity**, a class holding the necessities for perpetually changing an entity's position
- **Branch**, a class for a simple obstacle, slowing down and damaging the player, it can be dynamic or static
- **Goose**, a class for another obstacle, that would move erratically, also slowing and damaging the player
- **Opponent**, a class for the competitor boats that would race against the player, implementing
    - **AI**, a class for systematically moving an entity on the screen with a set behavior
- **Player**, a class for the players boat, needing
    - **Inputs**, as a way for the object to react the key presses of the player

Concrete Architecture

The concrete diagram was generated from our final code, using umldoclet (https://github.com/talsma-ict/umldoclet) a free api for translating Java code into UML class diagrams. Basic architecture has been kept fairly consistent to the ideas presented in the abstract diagram, but they have been altered and developed to better meet the requirements presented by our framework of choice, LibGDX. Our main components being:

- **DragonBoatRace**, the game class were the game's logic is being computed, also holding instances of every entity
- **B2ContactListener**, a class containing logic for the collision system
- **SelectScreen**, a class for drawing the boat selection screen, where the players are able to choose from a selection of boats with different stats
- **TitleScreen**, a class for drawing the screen at the start of the game, greeting the player
- **Utils**, a class used for utilities in relation to our way of rendering the world using tiles and Box2D
- **MovingObject**, a class similar to the one on the abstract representation, holding methods to draw an entity, create its boundary box and move it on the screen
    - **Boat**, a class holding methods for moving at accelerating speed with different stats tracking its top speed, maneuverability, robustness and acceleration
        - **Opponent**, a boat that is being controlled by an AI, holding all the necessary methods
        - **Player**, a boat that is being controlled by a player, holding methods to translate their key inputs
    - **Obstacle**, a class for an object meant to disturb the competitors, by dealing damage and slowing the boat
        - **Branch**, an obstacle dealing high damage, moving in a straight line or being static
        - **Goose**, an obstacle that moves in a random path and deals a small amount of damage
    - **TargetableObject**, a class for an object that can be set as a target for an AI's behavior



Class Hierarchy:

Note how the concrete architecture is framework dependent and implements and extends from LibGDX extensions, packages and classes.

- java.lang.Object
  - com.the8team.dragonboatrace.**b2ContactListener** (implements com.badlogic.gdx.physics.box2d.ContactListener)
  - com.badlogic.gdx.Game (implements com.badlogic.gdx.ApplicationListener)
    - com.the8team.dragonboatrace.**DragonBoatRace**
  - com.the8team.dragonboatrace.**MovingObject**
    - com.the8team.dragonboatrace.**Boat**
      - com.the8team.dragonboatrace.**Opponent** (implements com.badlogic.gdx.ai.steer.Steerable<T>)
      - com.the8team.dragonboatrace.**Player**
    - com.the8team.dragonboatrace.**Obstacle**
      - com.the8team.dragonboatrace.**Branch**
      - com.the8team.dragonboatrace.**Goose**
    - com.the8team.dragonboatrace.**TargetableObject** (implements com.badlogic.gdx.ai.steer.Steerable<T>)
  - com.the8team.dragonboatrace.**SelectScreen** (implements com.badlogic.gdx.Screen)
  - com.the8team.dragonboatrace.**TitleScreen** (implements com.badlogic.gdx.Screen)
  - com.the8team.dragonboatrace.**Utils**

Architecture (b)

MovingObject

This object is used as a superclass to all objects that will be moving around the screen. In the abstract architecture, MovingObject inherits from the entity InteractiveObject which inherits from VisibleObject. This provided it with the components of Position, AABB and Sprite. In the concrete architecture, MovingObject has no parent classes and is the super class to basically all other entities in the game. This superclass provides the collider body, sprite, width, height, movement speed and initial coordinates for child objects.
Requirements met:
UR_WIN: Boats must be able to move in order to reach the finish line and win the game.

Boat

This class inherits from the MovingObject class and is the basis for the player and all of the AI enemies in the game bar the obstacles. This class did not exist in the abstract architecture and player and opponent were completely separate objects that directly inherited from MovingObject but it was decided in the concrete architecture that it was more efficient to have a class for the boat and inherit both the player and the opponents from this as many of the components of both objects were the same. The collisions for the boat are controlled by the b2ContactListener class that listens for collisions between bodies.
Requirements met:
UR_BOATS: Provides the ability to create different boat objects with different qualities such as varying stamina, acceleration, max speed and maneuverability
FR_STAMINA: Boats have stamina that is slowly depleted as they move
FR_STAMINA_RECOVERY: Once the boat runs out of stamina, they are forced to a stop until their stamina regenerates.
FR_STAMINA_RESET: When the reset method is called, the boat's stamina is reset before the start of the next round.
FR_LANE_PENALTY: If a boat leaves its designated lane, it's speed is drastically reduced until it returns to within the boundaries of its lane.
FR_COLLISIONS: Controlled by b2ContactListener, when the boat collides it loses health and velocity

Player

The player class inherits from the Boat class and adds player input detection to allow for manual control of the selected boat using the arrow keys to increase / decrease velocity. In the abstract architecture this class inherited directly from the MovingObject class but inherited from the Boat class in the final implementation as it was unnecessary to rewrite the boat code for both the player and the opponents.
Requirements met:
UR_MOVEMENT: Class allows user input to control the movement of their boat around the screen.

Opponent

The opponent class also inherits from the Boat class but builds upon this with the functionality of obstacle avoidance using obstacle detection and automated steering. In the abstract architecture we had methods for moving but had not discussed the method of pathfinding that we would use for obstacle avoidance. The opponents are given differing levels of difficulty through parameters such as maneuverability, max speed and acceleration that differ with each colour of boat.
Requirements met:
UR_LOSE: Having opponents that are able to beat the player fulfills this requirement as it means the player can lose to the opponent AI.
UR_TEAMS: The opponents have different colours that signify what team they are on.
FR_OPPONENTS: Opponent boats can dodge obstacles and attempt to beat the player to the finish line.
FR_OPPONENT_BOATS: Opponents have the same system for health as the player and so can also be broken by colliding with other boats or obstacles.

Obstacle

The obstacle class provides a superclass to the two types of obstacles in the game: branch and goose. This class contains the damage dealt by the obstacle along with the horizontal & vertical velocities. In the abstract architecture this class did not exist but there were very few discrepancies between the goose and branch class so it would have been inefficient to have two independent classes.

Requirements met:
UR_OBJECTS: Adds two types of collidable objects to the game that the player must avoid.
UR_LOSE: Obstacles introduce the ability to lose the game through too many collisions leading to the boat running out of health.

DragonBoatRace

The DragonBoatRace class is the class for managing and controlling the game as it is played. This class controls the UI, starts and renders the game graphics and keeps track of all of the players and obstacles in order to determine the outcome. In our abstract architecture we did not discuss a game management system as it was implied but the concrete architecture is heavily reliant on this component and could not function without it.

Requirements met:
UR_ROUNDS: This class controls the flow of rounds and reinstantiates the boats and obstacles when necessary to begin new rounds.
UR_BGM: Custom background music is played throughout the rounds within this class.
FR_BOAT_NUM: This instantiates a set number of boats for the player to choose from / race against.
FR_SIMULTANEOUS: The other boats are drawn and artificially controlled within the render function of this script.
FR_HP_RECOVERY: Runs the player / opponent reset methods at the end of the rounds that resets their robustness.
FR_WIN_SCREEN: If the player wins, a screen is produced showing the top 3 boats in the race.
FR_LOSS_SCREEN: If the player loses, a screen is produced offering to restart the race or end the game.
FR_BROKEN_SCREEN: If the player runs out of health, a screen is displayed notifying the player and asking if they would like to restart the leg or wait to forfeit.
FR_BOAT_STATS: Each boat instantiated into the game is given different stats to give them strengths and weaknesses.
FR_HEALTH_BAR: A health bar is drawn above the player to visually indicate how much health they have left.
FR_STAMINA_BAR: A stamina bar is drawn above the player to show how much stamina they have left
NFR_FIRST_ROUND: The first round's time is not recorded and does not impact the game.

SelectScreen

The SelectScreen class controls the selection of the boat the player will control. The screen shows the sprites of each boat along with a brief statement of the boats main strength e.g. "Acceleration". This class was introduced after the abstract architecture was produced as more research into the engine and language was done.

Requirements met:
FR_BOAT_CHOICE: Allows the player to choose their desired boat through UI buttons

TitleScreen

This class controls the welcome screen to the game that then follows on to the selection screen. This class was also introduced after the abstract architecture was constructed.

Requirements met:
FR_WELCOME: Provides a welcome screen to the game

References

[1] Lucidchart, *What is Unified Modeling Language*. [Online]. Available: https://www.lucidchart.com/pages/what-is-UML-unified-modeling-language#section_0