Architecture

Team 06:

Adam Hagan

Tudor Ciobanu

Armintas Zadeika

Sanjel Sadikaj

Liu Zhang

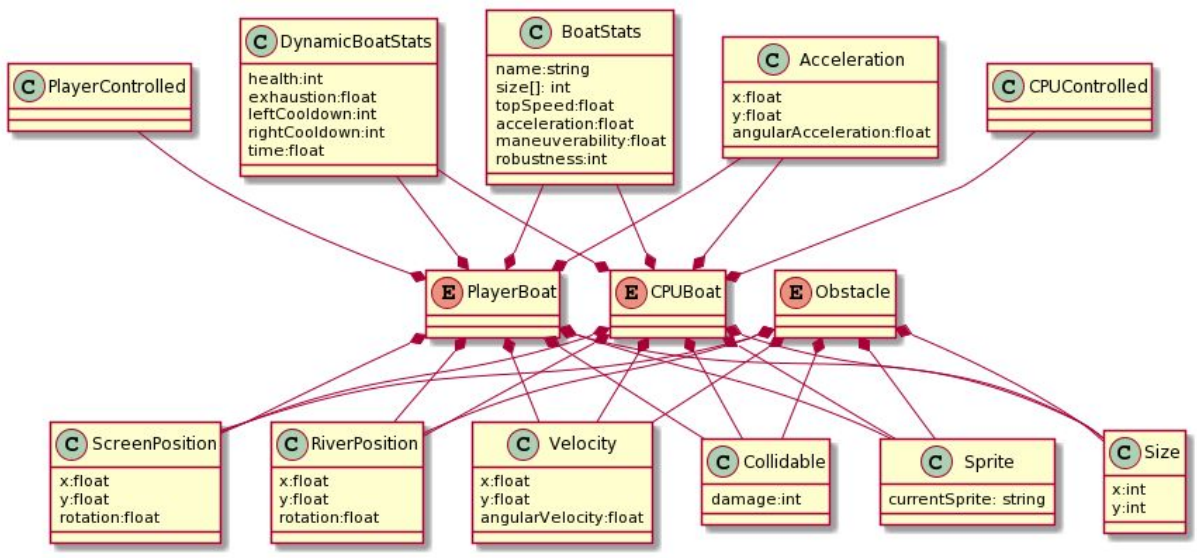
Na Tang

Languages and Tools Used:

- We made use of UML to create diagrams - specifically using PlantUML as it is easy to learn and the source to generate the diagrams can be easily stored as text, allowing us to modify them easily.
    - Our diagrams use a green **C** symbol to represent components, and an orange **E** to represent entities.
    - The ECS has been split into multiple diagrams, roughly organized into similar entities, to keep the diagrams from getting too large.
    - For the sake of readability, components in our diagrams have been arranged such that:
        - The bottom row of components are part of every entity in the diagram.
        - The top row of components are only part of some entities in the diagram.

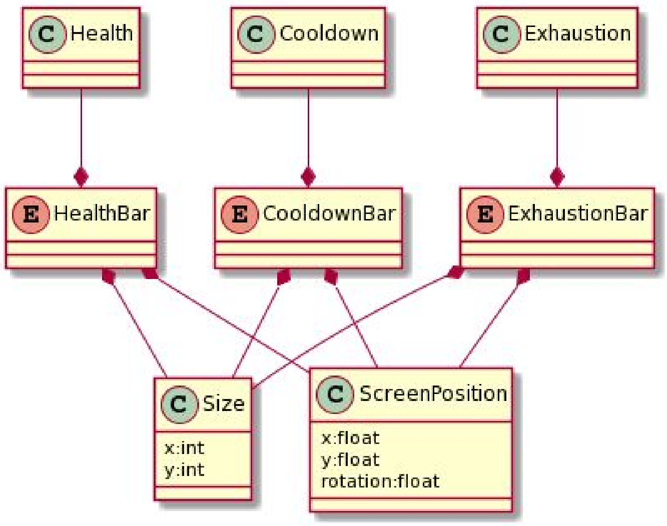
Abstract Architecture:

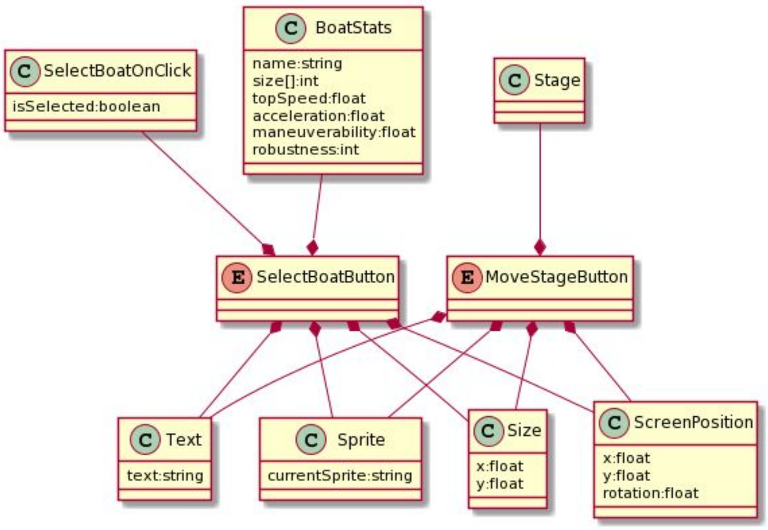
- We opted to use an Entity Component System is a widely used architectural pattern that follows composition over inheritance - all functionality is implemented by systems, which act on entities depending on which components they have**.**
    - Each system has a defined set of components, and will act on all entities with those components.
    - Communication between systems is limited, and is done through events.
- This diagram shows the entities and components of our river entities
    - Of note is BoatStats and DynamicStats - while they are very similar, they cannot be merged into a single component as BoatStats is used in MoveStageButton whereas DynamicBoatStats isn't.
- Common functionality between player boat control and CPU boat control will be implemented using events - the control systems will fire events that are caught by a common boat control system, which implements the functionality.



- The river game also contains objects like finish lines, lane markings and the river border, but we decided not to feature these as entities in the ECS, as we decided they are exceptional cases that would be easier to hard-code into systems (such as river borders being handled as bounds on the position of entities).

- The bars for each stat are very simple - while the bars are not explicitly part of the requirements, we decided that a visible cooldown was necessary for our choice of movement system, and the other bars were simple enough extensions that they were worth implementing.

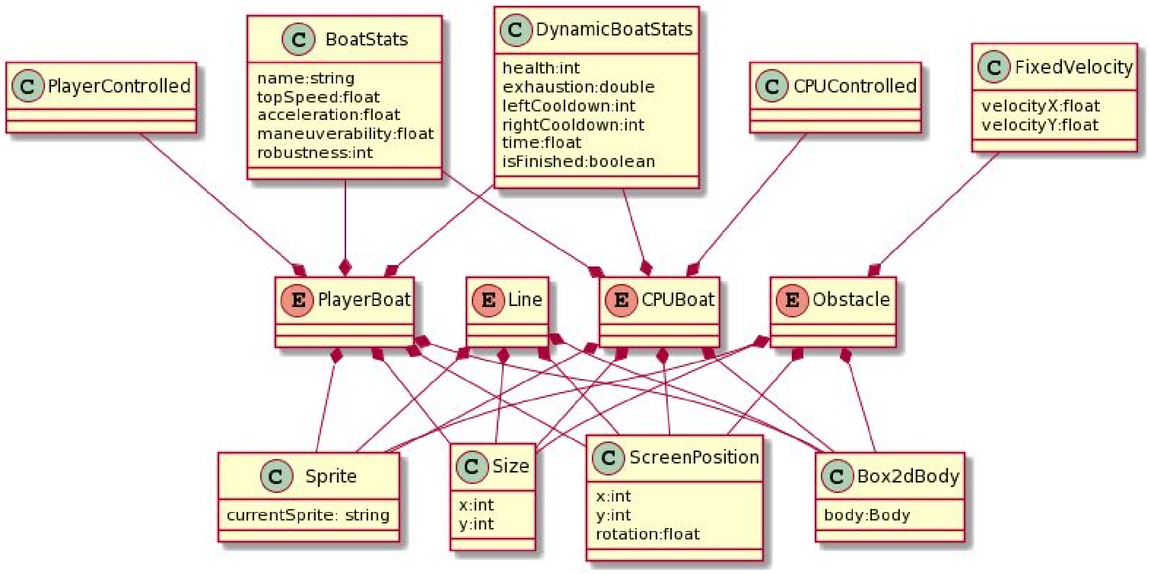




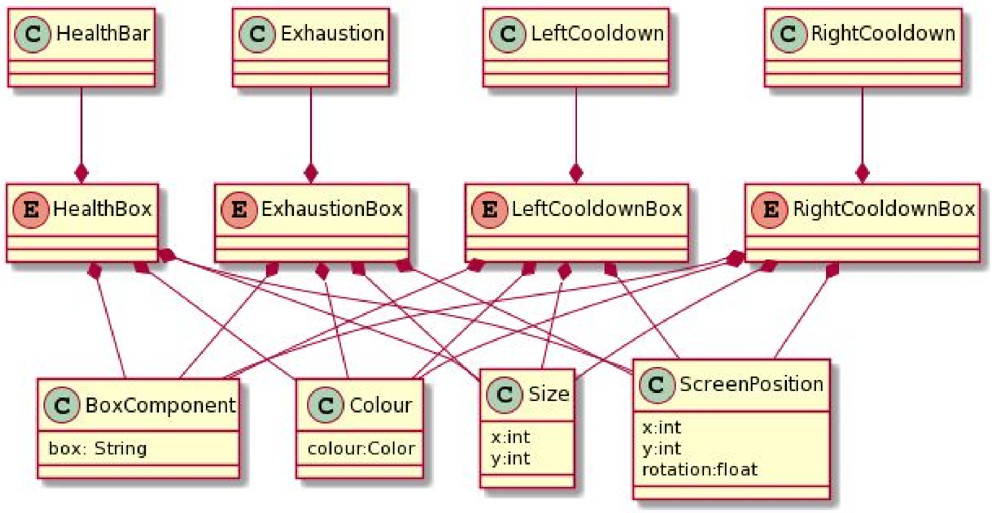
- The buttons on the main menu and results screen are also implemented as entities - moving between stages will be done by a system that will delete all entities and build the specified screen when an event is fired.

Concrete Architecture:

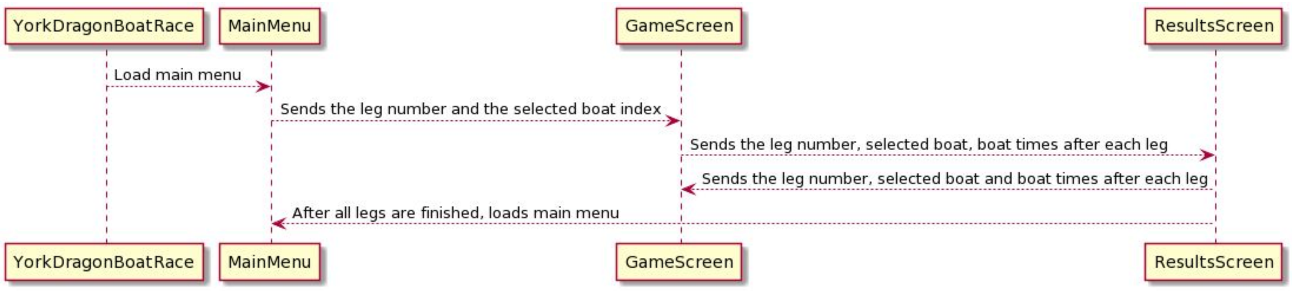
- The difference between concrete and abstract diagramas for the river entities is that RiverPosition, Collidable, Acceleration are replaced by Box2dBody and Velocity has been renamed to FixedVelocity.



- The bars are almost identical to the abstract - the only difference is that the CooldownBar has been split up to 2 entities - LeftCooldownBox and RightCooldownBox.



- Screens like the MainMenu, GameScreen and ResultsScreen are used to initialise stages. They exist outside the ECS, and only one exists at any given time.
    - Since data does not persist between screens, screens transfer data when they switch. For example, the MainMenu sends the initial leg number and the boat number that the player selected to the GameScreen.


YorkDragonBoatRace
MainMenu
GameScreen
ResultsScreen
Load main menu
Sends the leg number and the selected boat index
Sends the leg number, selected boat, boat times after each leg
Sends the leg number, selected boat and boat times after each leg
After all legs are finished, loads main menu

- All of the screen system change events are triggered using libgdx's built in rectangle objects as a button, with the exception of GameScreen which has its screen change attached to an ECS event.

Systematic Justification for the Abstract & Concrete Architectures:

- Our abstract architecture is an Entity Component System.
  - We chose an ECS for our architecture as it provides significant modularity and simplicity compared to other architectures (like simple object oriented ones) - allowing multiple people to work on related components of the game without interfering with each other.
- Main differences between abstract architecture and concrete are:
  - Scope of ECS was reduced to just the river game.
    - Functionality outside the river game was very simple and had almost nothing in common with entities within the river game.
    - Simple menu functionality was handled adequately by the built-in libraries of the engine, so ECS was deemed excessive.
  - Most physics related components were replaced by Box2dBody, as we opted to use Box2D to solve our physics.
    - We quickly discovered that doing physics ourselves was impractical, as collisions are hard to calculate and collisions of more than 3 bodies are very difficult to even approximate.
    - Box2D's "body" object contained almost all the data we were storing about physical objects, so many components were rendered obsolete.
  - A number of events that were planned weren't implemented.
    - Events relating to boat control were replaced with systems that directly controlled boats using a common boat functions library, as the observer pattern wasn't appropriate for the 1-1 relationship between boats and their actions.
    - As the stages/screen system was moved outside the scope of the ECS, the stage movement event has also been limited in scope to just the game screen.
- Notes regarding specific systems:
  - Box2D does not use pixels as its unit of measurement, instead 1 unit is one meter. Distances and sizes significantly far from 1 meter incur precision and performance penalties, so the objects on the river are scaled to around 0.1m to 0.3m.
- When rendering objects on the river, all entities are scaled up from their box2D dimensions and positioned relative to the player's boat. This creates the illusion of a camera following the player.

Relating Concrete Architecture to Requirements

System/Class	**Requirement Met**	**Notes**
PlayerBoatControl.java BoatControlCommon.java	FR_BOAT_MOVEMENT FR_PADDLER_STAMINA_ DECREASING FR_BOAT_TIMES FR_ZERO_ROBUSTNESS	In addition to handling the user's inputs, PlayerBoatControl.java also ends the race early if the player runs out of robustness.
PhysicsUpdate.java	FR_COLLISION_PHYSICS FR_MOVING_OBSTACLES	
CollisionLogic.java	FR_ROBUSTNESS_LOSS	
GameScreen.java	FR_REPLENISHING_LEGS FR_NUMBER_OF_BOATS FR_OBSTACLE_COLLISION FR_INCREASED_OBSTACLE S	GameScreen.java is responsible for initialising the game, with features according to the leg. When the screen is constructed, boats are constructed from their base data from scratch, refreshing the stats.
ResultsScreen.java	FR_RACE_LEGS FR_FINAL_LEG	ResultsScreen.java is responsible for determining if the player can continue playing depending on the results of the previous race.
MainMenu.java	FR_PICK_BOAT NFR_DIFFERENT_BOAT	
InfoScreen.java	FR_INSTRUCTIONS NFR_INSTRUCTIONS	