Architecture

Team 15
Team 14

Joe Wrieden
Benji Garment
Marcin Mleczko
Kingsley Edore
Abir Rizwanullah
Sal Ahmed

Babar Khan
George Rogers
Jacob Turner
James Crump
Chloe Remmer
Shijie Lin

Architecture v0.5

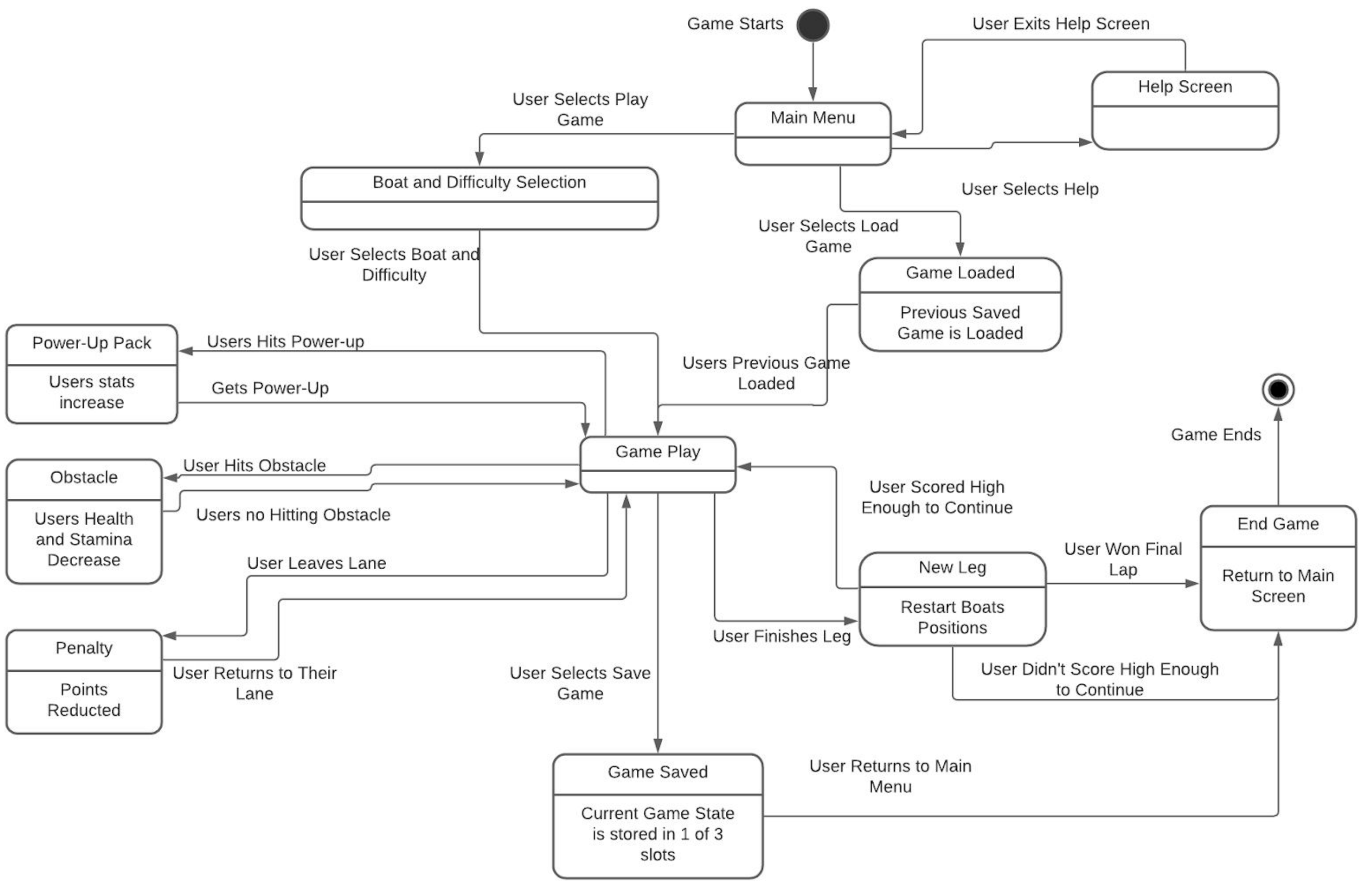
Preface

This document documents our various architectural iterations over the course of the project schedule. The document is added to as new requirements arise (refer to version history for this document's various iterations).

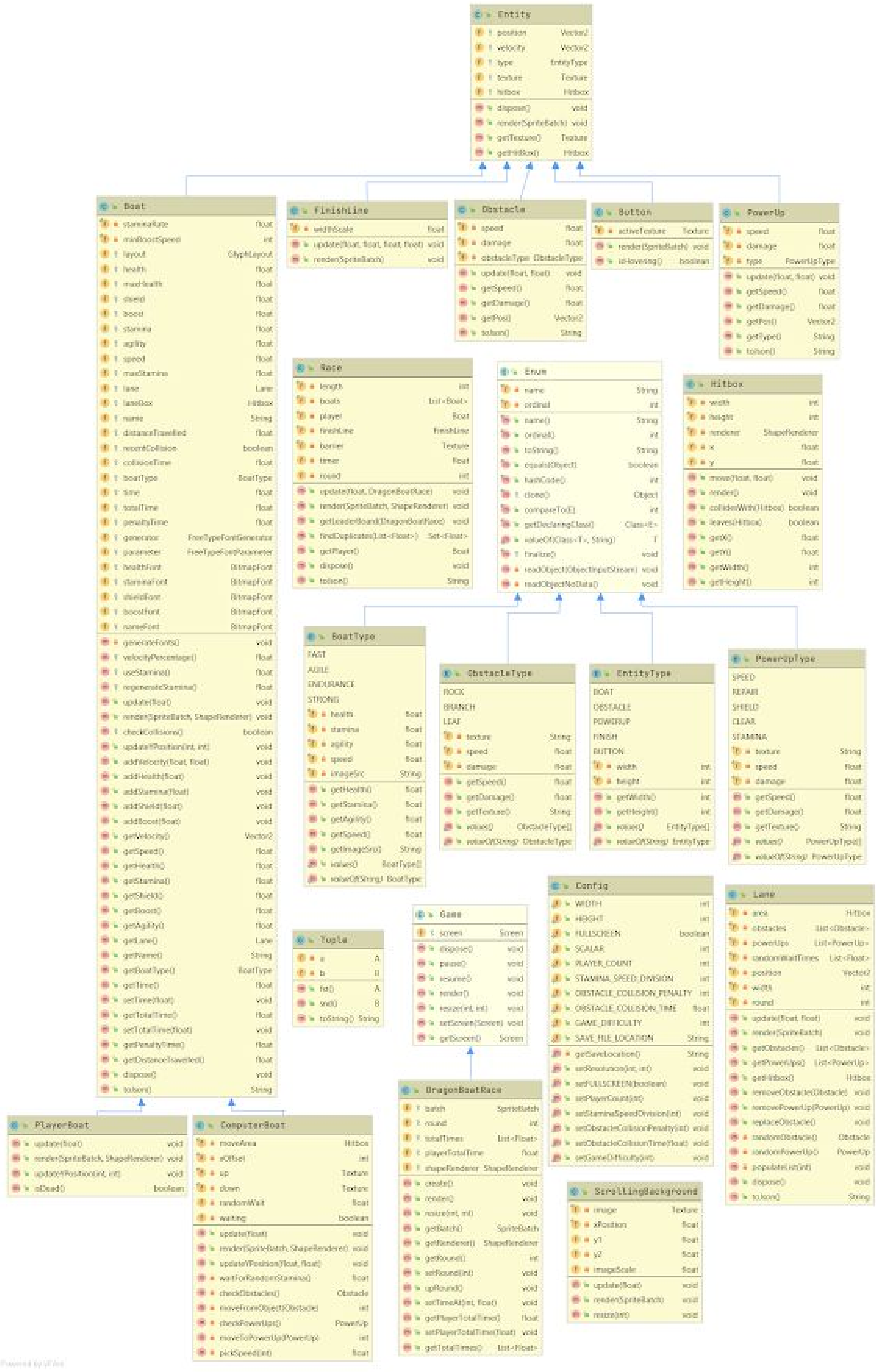
Tools Used

We developed a UML State Diagram for our Abstract Architecture using the online tool LucidChart. LucidChart allows us to represent the game in a clear and concise manner, in a natural language, whilst still allowing teams other than the Software Development Team to understand the structure of the code. A key feature of LucidChart is that it allows for collaboration between people, which we thought was beneficial considering more than one person was working on the development of the architecture. We decided to focus on using a UML Structural Diagram for the Concrete Architecture, as we felt it would give the Software Development Team a better idea of what objects need to be implemented and the different sections can be easily seen and broken up into different tasks. We debated using another Behavioural Diagram instead of a Structural Diagram for the Concrete Architecture, however, given that we were taking over another groups code, thought it'd be more beneficial to look at the current structures of their code and how we can reuse or add to them rather than focusing on how the objects collaborate with each other. Therefore, the tool we used to develop the Concrete Architecture was the UML Class Diagram tool provided by IntelliJ IDE.

Abstract Representation of Software Architecture



Concrete Representation of Software Architecture



Justification for Abstract Representation of Software Architecture

This architecture was developed from the requirements we elicited, but reflects our decisions made prior to actual implementation, and serves as a basis for our lower level design, the concrete representation of our Software Architecture, which can be found further down this document.
Our Abstract Architecture was designed to focus on the key objects required to implement the game and the necessary actions a user must take to enter a different 'stage' of the game. It allowed us to visualise and break down the important requirements of the game and gave the Software Development Team a better understanding of how different aspects worked together. By using a higher level design with low complexity, it allowed anyone of any technical abilities to understand what sections were necessary to complete the game and would help us in bridging the communication gap between system stakeholders and software engineers.

Justification for Concrete Representation of Software Architecture

To develop our Concrete Architecture, we looked at our Abstract Architecture to detect any key states or objects necessary for the game. Due to how the Abstract Architecture is broken down, we could see if there are any similarities between different objects, such as Power-Ups and Obstacles and so could reuse a base class using inheritance. The Abstract Architecture also helped us identify any functionalities that the user must have and where different stats of the boats need to change and this helped for identifying different fields and methods of the object's.
We also looked into common pre existing architectural patterns that we could reuse. We could have used an entity-component system, as it is notable in game development due to its not being subject to the rigid class hierarchies of object oriented programming (especially difficult when entities that incorporate different types of functionalities need to be added to the hierarchy), however we decided to use an object oriented approach via inheritance for the most part, due to the fact that the complexity of our game mechanic is not to such an extent that it would result in inefficient code which would become increasingly difficult to maintain. Inheritance also allows for efficient code reuse, since changes in the parent class affect all children, avoids duplicity and data redundancy, and reduces time and space complexity. Additionally, we had to bear our time constraints in mind and stuck to what we had a solid understanding of.

Justification for how the concrete architecture builds from system requirements:

Class	**Paren t**	**Justification and Related Requirements**
Entity	-	This class makes it easier to add new objects such as boats and obstacles to the game, as their main values are already stored in the class *Requirements: UR_OBSTACLES, UR_POWERUPS, FR_CHOOSING_UNIQUE_BOAT*
Boat	Entity	Defines key fields and methods of boat type, all instances of this will be unique *Requirements: FR_CHOOSING_UNIQUE_BOAT, UR_BOAT_UNIQUENESS*
PlayerBoat	Boat	Boat specific to the player and is controlled by standard WASD controls. Health, Stamina can be increased or decreased by collision with power-ups and obstacles respectively *Requirements: FR_CHOOSING_UNIQUE_BOAT, UR_PADDLERS_STAMINA_DECREASE, UR_OBSTACLE_COLLISION, UR_MOVEMENT, UR_POWERUPS, FR_HIT_DECREASED_BOAT_CONDITION, FR_INPUT_DETECTION, FR_POWERUP_EFFECTS*

Computer Boat	Boat	Boats other than players that are controlled by computer, will be players competitors *Requirements: FR_QUALIFIER_RACES, FR_FINAL_RACE*
FinishLine	Entity	Class to mark the end of a leg of the race, players score determines if they move on to next leg or win the race *Requirements: FR_QUALIFIER_RACES, FR_FINAL_RACE, UR_GAME_END, UR_RACE_TOTAL*
Obstacle	Entity	Main fields and methods needed for an obstacle object *Requirements: UR_OBSTACLES, FR_OBSTACLE_POWERUP_RATE, FR_OBSTACLE_SPAWN*
Button	Entity	Processes the input if a user clicks on a button *Requirements: FR_SAVE_RELOAD, FR_INPUT_DETECTION, FR_CHOOSING_UNIQUE_BOAT*
Power-Up	Entity	Contains main fields and methods needed for a power-up object *Requirements: UR_POWERUPS, FR_OBSTACLE_POWERUP_RATE, FR_POWERUP_EFFECTS*
Race	-	Used to implement end of a race and, during its duration, the boats stats *Requirements: FR_FINAL_RACE, FR_QUALIFIER_RACES, FR_HIT_DECREASED_BOAT_CONDITION*
Hitbox	-	Used to detect collisions between boat objects and obstacle or power-up objects *Requirements: FR_COLLISION_DETECTION, FR_POWERUP_EFFECTS, FR_HIT_DECREASED_BOAT_CONDITION*
BoatType	Enum	An Enum class that holds predefined attributes of specific boat objects. *Requirements: NFR_ATTRIBUTES, FR_CHOOSING_UNIQUE_BOAT*
ObstacleType	Enum	An Enum class that holds predefined attributes of specific obstacle objects *Requirements: UR_OBSTACLES, UR_DIFFICULTY*
EntityType	Enum	Helps in adding the predefined objects such as boats, powerups and obstacles to game *Requirements: FR_CHOOSING_UNIQUE_BOAT, UR_OBSTACLES, UR_POWERUPS*
PowerUpType	Enum	An Enum class that holds predefined attributes of specific powerup objects *Requirements: UR_POWERUPS, FR_POWERUP_EFFECTS*
Tuple	-	Helper class to store two different objects
DragonBoatRace	Game	Class used to initialise the game *Requirements: FR_SAVE_RELOAD, FR_SCREEN_DISPLAY, FR_INFORMATION_DISPLAY*
Config	-	Used to reload a previously saved game and contains settings of a game *Requirements: FR_SAVE_RELOAD, UR_SAVE_RELOAD*
Lane	-	Used to check if player is still in their lane or not, if not then issues a penalty *Requirements: UR_PLAYER_PENALTY, FR_BOUNDARY_DETECTION*
ScrollingBackground	-	Makes it seem like race background is continuously moving, creates invested user experience *Requirements: NFR_POSITIVE_UX*

Bibliography

- “Software Engineering”, Ian Sommerville, Chapter 6
- ”UML Online Training”, Tutorials Point
- “Use Case Models and State Models”, Binary Terms
- Software Architecture: Foundations, Theory, and Practice, R.N. Taylor, N. Medvidovic, and E.M. Dashofy John Wiley & Sons, 2008.
- Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. Documenting Software Architectures: Views and Beyond. Addison- Wesley, Boston, 2002.