Architecture

Team 14 - Taken On By Team 15

Joe Wrieden

Benji Garment

Marcin Mleczko

Kingsley Edore

Abir Rizwanullah

Sal Ahmed

Preface

This document documents the different architectural structures involved in the development of the project. It uses the Requirements set out in the requirement documentation in order to produce a well-thought design to base our implementation on.

Architectural Approach Decision

We first looked into many common pre-existing architectural approaches that we could reuse for our implementation, as this would influence the language we used to define our architectures. We looked into entity-component system approaches, as they are often used in game development due to an entity component system not being subject to the rigid class hierarchies of object oriented programming (which poses difficulties especially when entities that incorporate different types of functionalities need to be added to the hierarchy [3]), however we decided to use an object oriented approach via inheritance, due to the fact that the complexity of our game mechanic is not to the extent that OOP would result in inefficient code. 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. OOP is suitable for small scale projects as well. Additionally, we had to bear our time constraints in mind and stuck to the approach most members of the team had a solid understanding of. Finally, and most importantly, the nature of the project was such that all main classes in the game derived from the three classes: Boat, Obstacle and Power-Up - and so inherently leant towards an inheritance based approach.

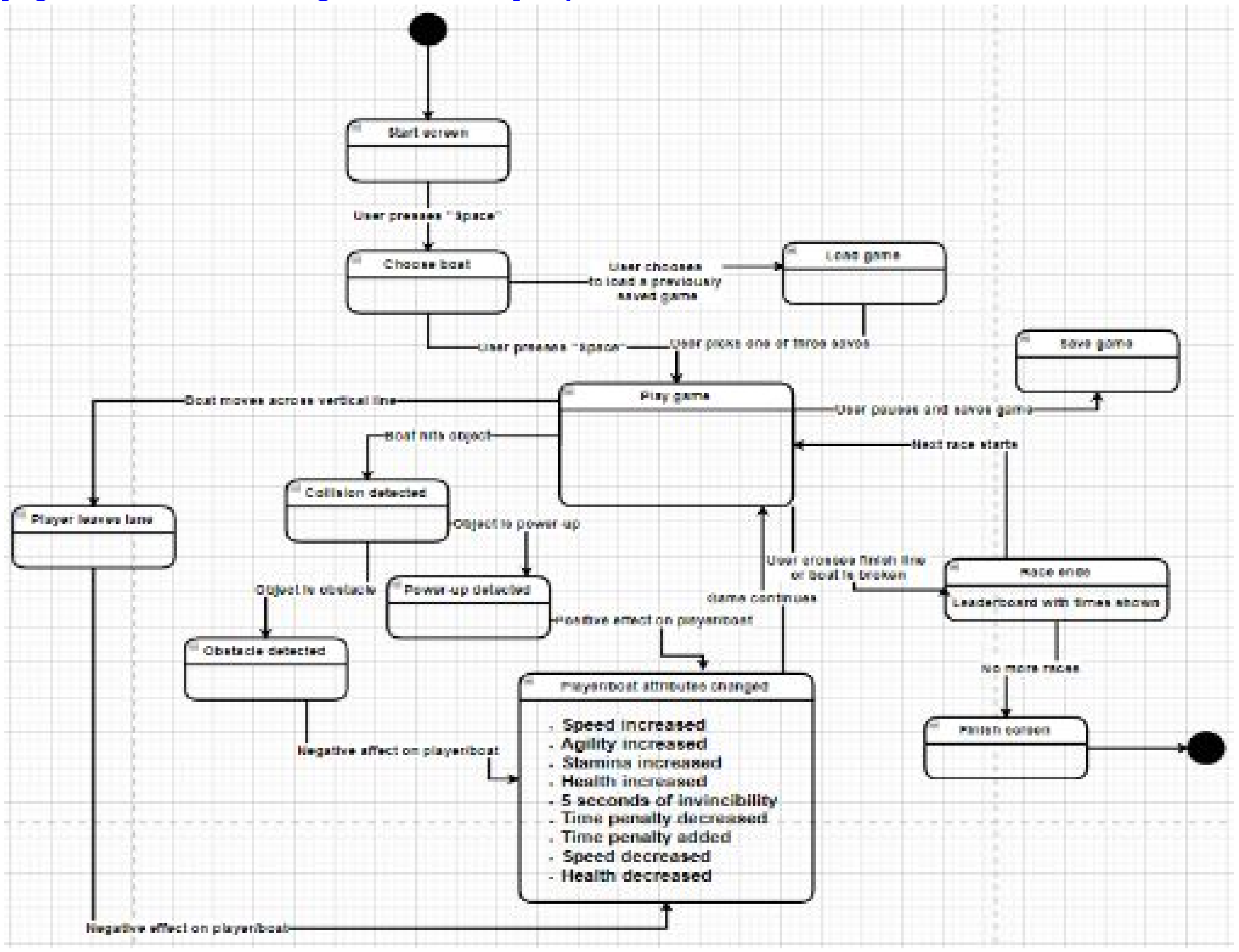
Language(s) and Tool (s) used to create the Abstract and Concrete Architectures

Since we decided upon an OOP approach for our implementation, we used UML diagrams to draw and design the architectures [1], as UML diagrams are commonly used in tandem with OOP approaches. Additionally, UML is an industry standard and not language nor technology dependent [2]. An alternative could have been to use natural language to describe our architecture, however this was decided against because of its imprecision and verbosity, with there being many different ways of doing the same thing. Comparatively, UML graphical modelling allows people from all backgrounds, technical or nontechnical, to grasp the gist of complex concepts that code aims to carry out without excessive reliance upon 'technical jargon'.

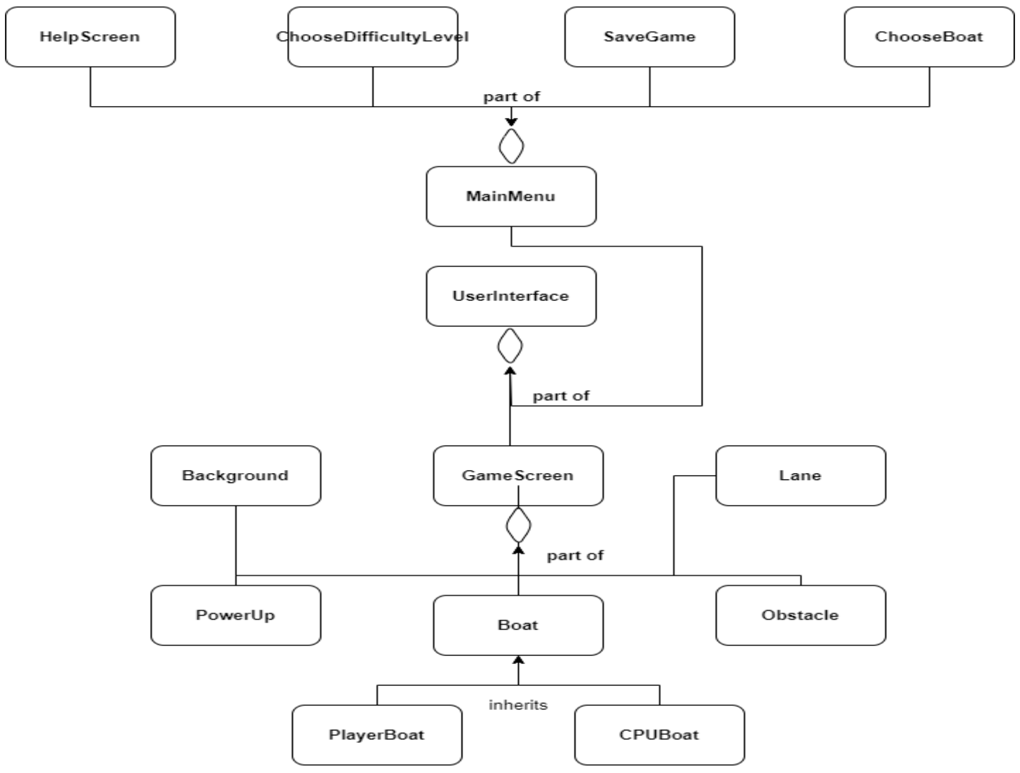
Draw.io was the tool we used to create our abstract architecture as the tool has many useful functionalities, one being that it allows for real time collaboration when used with Google Drive. In addition, there is a lot more ease of use when creating class diagram models than using other vector type softwares.

For the concrete architecture, we have decided to use the UML Class Diagram tool provided by the IntelliJ IDE, as it allows for seamless creation of diagrams, and the diagrams appropriately reflect the structure of actual classes and methods used in the application.

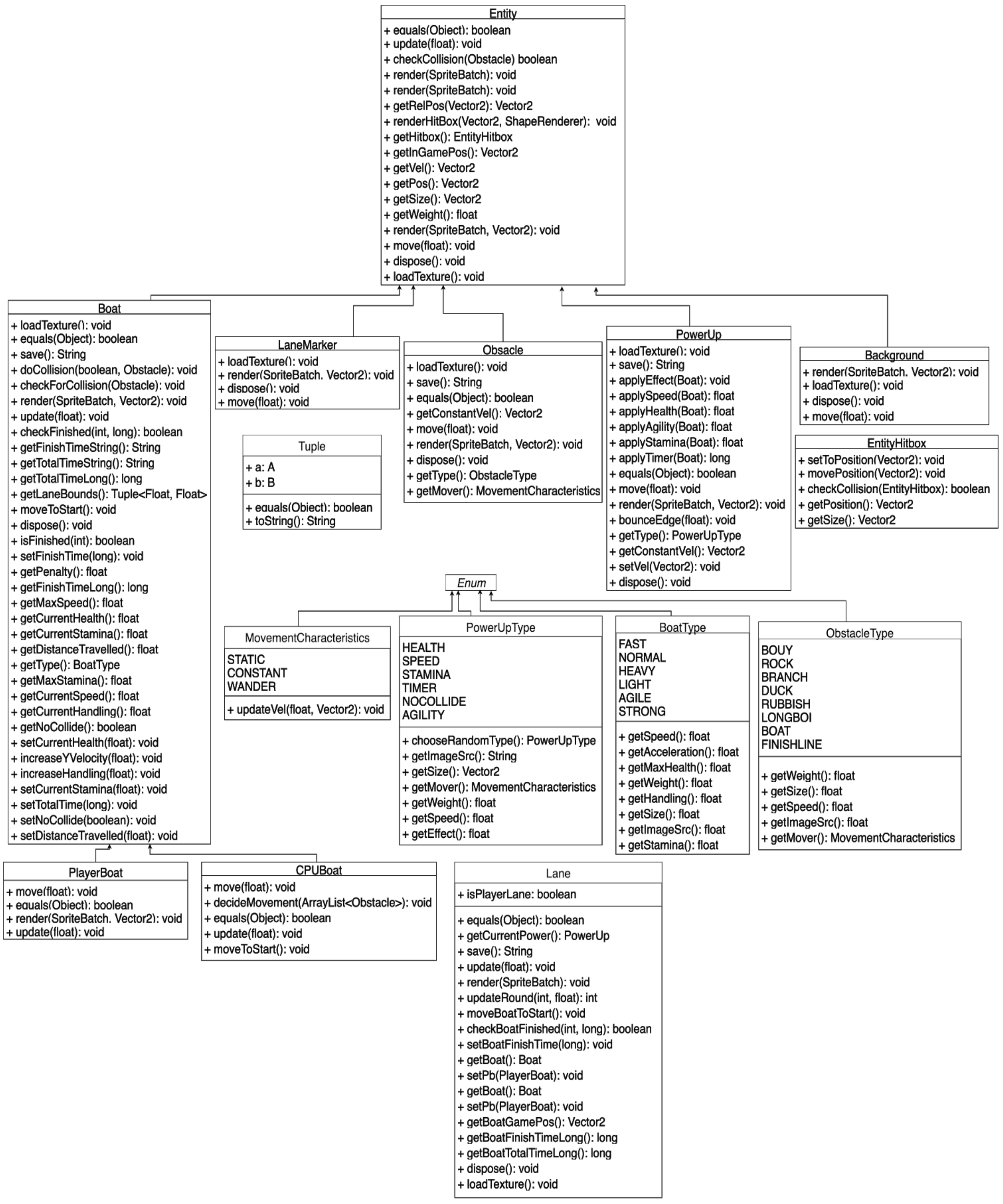
[Fig. 1 State Machine Diagram - Abstract] - Updated



[Fig. 2 Class Diagram - Abstract] - Updated



[Fig.3 Class Diagram - Concrete] - Updated


Entity
+ equals(Object): boolean
+ update(float): void
+ checkCollision(Obstacle) boolean
+ render(SpriteBatch): void
+ render(SpriteBatch): void
+ getRelPos(Vector2): Vector2
+ renderHitBox(Vector2, ShapeRenderer): void
+ getHitbox(): EntityHitbox
+ getInGamePos(): Vector2
+ getVel(): Vector2
+ getPos(): Vector2
+ getSize(): Vector2
+ getWeight(): float
+ render(SpriteBatch, Vector2): void
+ move(float): void
+ dispose(): void
+ loadTexture(): void
Boat
+ loadTexture(): void
+ equals(Object): boolean
+ save(): String
+ doCollision(boolean, Obstacle): void
+ checkForCollision(Obstacle): void
+ render(SpriteBatch, Vector2): void
+ update(float): void
+ checkFinished(int, long): boolean
+ getFinishTimeString(): String
+ getTotalTimeString(): String
+ getTotalTimeLong(): long
+ getLaneBounds(): Tuple<Float, Float>
+ moveToStart(): void
+ dispose(): void
+ isFinished(int): boolean
+ setFinishTime(long): void
+ getPenalty(): float
+ getFinishTimeLong(): long
+ getMaxSpeed(): float
+ getCurrentHealth(): float
+ getCurrentStamina(): float
+ getDistanceTravelled(): float
+ getType(): BoatType
+ getMaxStamina(): float
+ getCurrentSpeed(): float
+ getCurrentHandling(): float
+ getNoCollide(): boolean
+ setCurrentHealth(float): void
+ increaseYVelocity(float): void
+ increaseHandling(float): void
+ setCurrentStamina(float): void
+ setTotalTime(long): void
+ setNoCollide(boolean): void
+ setDistanceTravelled(float): void
LaneMarker
+ loadTexture(): void
+ render(SpriteBatch, Vector2): void
+ dispose(): void
+ move(float): void
Obsacle
+ loadTexture(): void
+ save(): String
+ equals(Object): boolean
+ getConstantVel(): Vector2
+ move(float): void
+ render(SpriteBatch, Vector2): void
+ dispose(): void
+ getType(): ObstacleType
+ getMover(): MovementCharacteristics
PowerUp
+ loadTexture(): void
+ save(): String
+ applyEffect(Boat): void
+ applySpeed(Boat): float
+ applyHealth(Boat): float
+ applyAgility(Boat): float
+ applyStamina(Boat): float
+ applyTimer(Boat): long
+ equals(Object): boolean
+ move(float): void
+ render(SpriteBatch, Vector2): void
+ bounceEdge(float): void
+ getType(): PowerUpType
+ getConstantVel(): Vector2
+ setVel(Vector2): void
+ dispose(): void
Background
+ render(SpriteBatch, Vector2): void
+ loadTexture(): void
+ dispose(): void
+ move(float): void
EntityHitbox
+ setToPosition(Vector2): void
+ movePosition(Vector2): void
+ checkCollision(EntityHitbox): boolean
+ getPosition(): Vector2
+ getSize(): Vector2
Tuple
+ a: A
+ b: B
+ equals(Object): boolean
+ toString(): String
Enum
MovementCharacteristics
STATIC
CONSTANT
WANDER
+ updateVel(float, Vector2): void
PowerUpType
HEALTH
SPEED
STAMINA
TIMER
NOCOLLIDE
AGILITY
+ chooseRandomType(): PowerUpType
+ getImageSrc(): String
+ getSize(): Vector2
+ getMover(): MovementCharacteristics
+ getWeight(): float
+ getSpeed(): float
+ getEffect(): float
BoatType
FAST
NORMAL
HEAVY
LIGHT
AGILE
STRONG
+ getSpeed(): float
+ getAcceleration(): float
+ getMaxHealth(): float
+ getWeight(): float
+ getHandling(): float
+ getSize(): float
+ getImageSrc(): float
+ getStamina(): float
ObstacleType
BOUY
ROCK
BRANCH
DUCK
RUBBISH
LONGBOI
BOAT
FINISHLINE
+ getWeight(): float
+ getSize(): float
+ getSpeed(): float
+ getImageSrc(): float
+ getMover(): MovementCharacteristics
PlayerBoat
+ move(float): void
+ equals(Object): boolean
+ render(SpriteBatch, Vector2): void
+ update(float): void
CPUBoat
+ move(float): void
+ decideMovement(ArrayList<Obstacle>): void
+ equals(Object): boolean
+ update(float): void
+ moveToStart(): void
Lane
+ isPlayerLane: boolean
+ equals(Object): boolean
+ getCurrentPower(): PowerUp
+ save(): String
+ update(float): void
+ render(SpriteBatch): void
+ updateRound(int, float): int
+ moveBoatToStart(): void
+ checkBoatFinished(int, long): boolean
+ setBoatFinishTime(long): void
+ getBoat(): Boat
+ setPb(PlayerBoat): void
+ getBoat(): Boat
+ setPb(PlayerBoat): void
+ getBoatGamePos(): Vector2
+ getBoatFinishTimeLong(): long
+ getBoatTotalTimeLong(): long
+ dispose(): void
+ loadTexture(): void

Justification of the Abstract architecture

By incorporating the production of this design into our Software Engineering process (as opposed to skipping over this stage and implementing a lower level approach), any necessary changes were easier and less costly in terms of time, resources and effort than they would have been in the implementation stage. Not only do these models provide a bridge for the communication gap between the stakeholders and software engineers, but also all project planning in our project is based on them, such as deciding on the allocation of work and outlining any problems that could affect the actual implementation.

The abstract architecture consists of a structural diagram, as well as a behavioural diagram, which is a state machine diagram. In our structural view of the abstract architecture (see: Fig. 2), a simple class relationship diagram was used to map relationships and show aggregations and inheritance between classes. This model is also reflective of the OOP approach. This architecture was developed from the requirements that we generated in the requirement report (this can be found here). We checked that every system requirement mapped to at least one component in this composite architectural model, in order to ensure that we were not deviating from the original requirements set out. The architecture generated reflects the choices that were made before the actual implementation of the project. The abstract architecture acts as the basis for our lower level design, the concrete architecture (see Fig. 3).

Comparatively, its behavioural counterpart (see: Fig. 1) captures the dynamic aspects of the system (as defined in the System Requirements) by describing object communication (as in how the different components interact with each other and their effects on said components) while displaying the different stages that the parts of the game will go through based on the events that occur. We chose to use a state chart diagram in particular, because state diagrams define different states of an object, with the state being a reaction to external or internal events. This could be used in clarifying to all those involved in the system how the object states of the various components of the system will change in practice, depending on the various conditions and events occurring, according to the functional and nonfunctional system requirements (the document containing our requirements can be found here).

Though there is some overlap between the different views, together the two components of the abstract architecture allow us to visually see and break down the key requirements of the system, without increasing the technicality difficulty, while allowing developers to see what they need to incorporate into the implementation - both behaviourally and structurally.

Justification of the Concrete architecture

Further on in our Software Development Lifecycle, we developed a concrete representation of what we have planned for the Software Architecture of the game. This concrete representation is composed of a structural diagram representing the static features of the system. We used a class diagram form of structural modelling, as it is most applicable to the object oriented style used in our programming solution. The UML class diagram created during the abstract phase has been expanded on and now fully describes each class, and specifies its relationships, attributes and methods. The naming convention has for the most part stayed the same from the abstract architecture as this allows us to clearly see the how we have built from it and causes less confusion. Further details were also incorporated based on trial and error during coding, for example when we decided to opt against integrating a help screen into the implementation, as defined in the abstract architecture, instead deciding to make a user manual, due to our not having enough time to change Team 14's implementation too much. The concrete architecture together with the JavaDocs work as the system documentation which may prove useful for future developers working on this project.

We checked that every system requirement relates forward to at least one component in this architectural model, in order to make sure everyone's understanding was thorough and up to standard before implementation. The components of the concrete architectural diagrams' relation to system requirements (in turn derived from user requirements, so we are making sure that we are still following through with the requirements we elicited) are justified under "Systematic Justification of the Concrete architecture".

The diagram provides a critical link between the requirements engineering and the actual design of the software we implemented. It identifies the main structural components in the system and the relationships between them. By generating the concrete architecture from the abstract architecture (the precursor of which in turn was the functional and nonfunctional requirements), it is ensured that we keep to the requirements set out by the stakeholders, and as a result the concrete architecture helps reinforce these requirements into our software implementation. A key difference in the two representations is that this representation has moved away from explicitly referring to the nonfunctional requirements, as they have been incorporated into the design, and since this one is more technical and closer to the code. Having this concrete representation of the design we have settled on will allow it to be easier for us to map the necessary components into the actual code in the organisational way we decided upon.

Relating the Concrete Architecture to the Requirements Systematically

Component	**Justification**
Entity	This is an ENUM class which makes it easier and less error prone to add new obstacles and boats to the program, as values don't need to be defined every time. The values are stored in this class.
Boat	This consists of both the player boats and the CPU boats. All objects that extend from this will be unique and have unique stats which relates back to UR_Option. The unique specs are speed, acceleration, weight, strength and handling.
Obstacle	This class consists of all the obstacles that will be present in the game. The requirement FR_Damage is dependent on this.
LaneMarker	Allows for a vertical line to be drawn between lanes, relates to UR_Race_Track.
PowerUp	This class enables an effect to be applied to the user boat, affecting it's velocity and health, relating to UR_Power_Ups.
Background	Allows for the background of the game screen to be rendered. Relates to UR_Race_Track.
Tuple	A small class to assist with efficiency.
MovementCharacteristics	Allows for the change of the velocity of the boat. Relates to UR_Power_Ups and UR_Damage.
EntityHitbox	Allows for the ability to check whether objects have collided. Mainly used for checking collisions between the user boat and obstacles or power-ups.
Lane	This will cause a negative effect upon the player and will affect their finishing time . This relates to UR_Penalty. It also allows for the spawning of power-ups and obstacles, as well as the player boat and AI boats. It will relate to UR_Power_Ups and UR_Damage.
PlayerBoat	Gives the user the ability to control the boat, this relates to UR_Race_Track.
CPUBoat	This class holds the difficulty of the boat. It also controls the direction/movement of the AI boat. This relates to UR_Difficulty_ Level.
BoatType	Holds all the pre-set attributes of the different boats, relates to UR_Option.
PowerUpType	This class allows a power-up to be chosen, and it holds all the attributes of each power-up. This relates to UR_Power_Ups.
ObstacleType	This class holds all the attributes of each obstacle. This relates to UR_Damage.

Reference

[1] “Unified modeling language (UML),” *Geeksforgeeks.org*, 27-Oct-2017 [Online]. Available: https://www.geeksforgeeks.org/unified-modeling-language-uml-introduction [Accessed: 18-Jan-2021].

[2] “UML Online Training”, *Tutorials Point,* [Online]. Available: https://www.tutorialspoint.com/uml_online_training/index.asp [Accessed: 18-Jan-2021]

[3] “Entity Component Systems”, *guru99* [Online]. Available: https://www.guru99.com/entity-component-system.html [Accessed: 18-Jan-2021]

[4] “OOP Pros and Cons”, *Green Garage Blog,* [Online]. Available: https://greengarageblog.org/6-pros-and-cons-of-object-oriented-programming [Accessed: 17-Jan-2021]

[5] “Behaviour vs Structural Diagrams”, [Online]. Available: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/behavior-vs-structural-diagram [Accessed: 17-Jan-2021]