Architecture

Team 15

Joe Wrieden

Benji Garment

Marcin Mleczko

Kingsley Edore

Abir Rizwanullah

Sal Ahmed

Edited by:
Team 13

Will Dunlop

Samin Alborzi Movahhed

Nam Duong

Jakub Grzmil

Darcy Adams

Davids Kacs

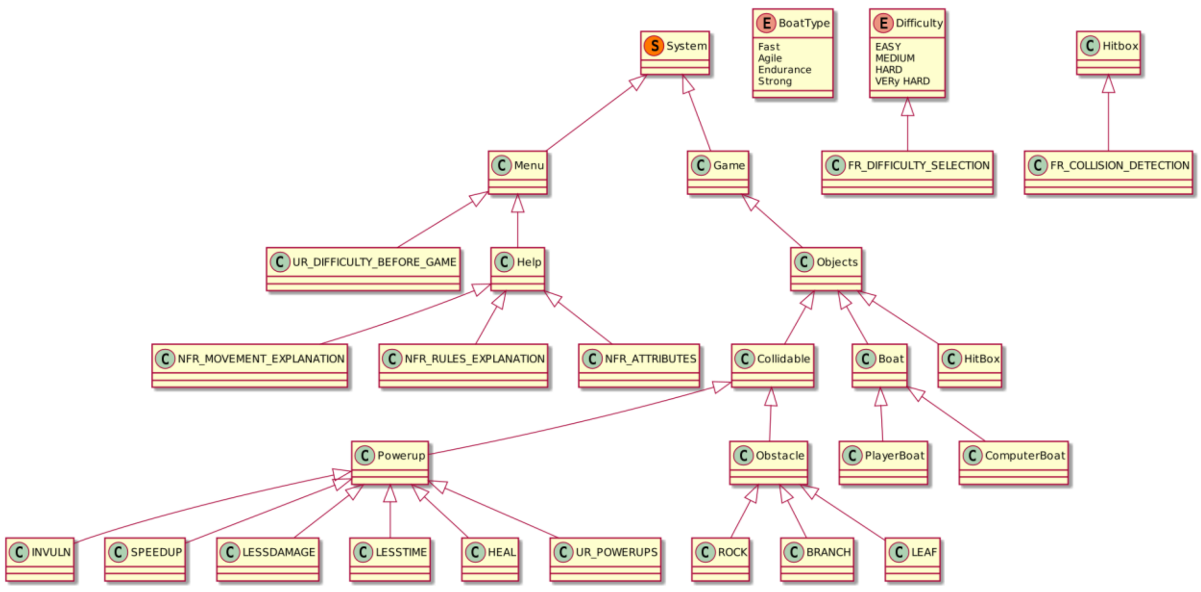
Architecture v0.6

Preface

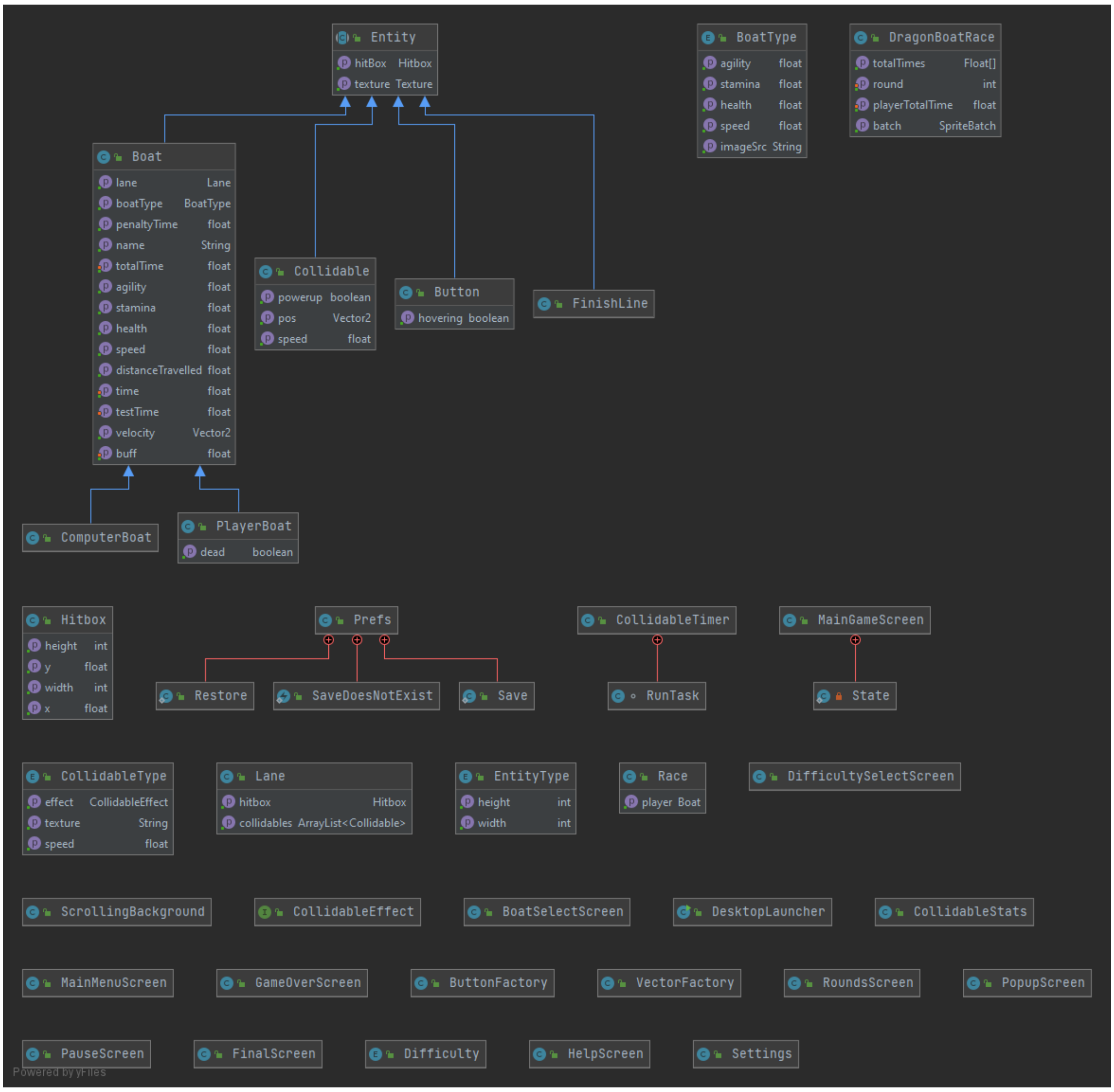
This document documents our various architectural iterations over the course of the project schedule. The document is added to as new requirements arise (refer to version history for this document's various iterations).

- v0.1 Creation of the document, and adding our abstract class diagram architectural representation.
- v0.2 Justifications for the abstract representation have been added but require revision.
- v0.3 Justifications for the abstract representation have been revised. Concrete class diagram added. Behavioural diagram may be added. Justifications for concrete diagram(s) needed.
- v0.4 Concrete diagram justifications added and revised. A behavioural state diagram may still be added and justified.
- v0.5 Last edits for final edition. Team decision to not add behavioural diagram due to time restraints for deliverables.
- v0.6 Edited version of this document, updating all the changes made to the project from Team 13

Abstract Representation of Software Architecture



<u>Concrete Representation of Software Architecture</u>



In our team’s Requirements Engineering, we used the PlantUML tool to create the UML class diagram above as a very high level, abstract representation of the game’s Software Architecture. UML was our choice of modelling language, as using natural language to describe architecture can be imprecise and verbose since there are many different ways of doing the same thing, however UML graphical modelling allows people from all backgrounds, technical or nontechnical, to grasp the gist of complex concepts that code aims to carry out. As well as this, it is an industry standard and not language nor technology dependent. This diagram is an updated version of the project’s concrete architecture, changed based on the new/updated classes.

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. The architecture shows the classes (with self explanatory names) that the game will consist of, and we checked that every component in this abstract model relates back to the user requirements, to ensure that we were not diverging from the requirements set out.

This static perspective of our architectural model's structure enabled us to visualise and break down the most important requirements of the system (as defined in the System Requirements) that we needed to prioritise working on, allowing us to be more confident about satisfying the requirements and having a reliable system, but also make changes easily. Had it been a lower level design closer to the detail of the code, making necessary design adjustments would have been costly and difficult on top of already having invested considerable time, resources and effort. Not only is it useful for bridging the communication gap between system stakeholders and software engineers, but also it aids project planning by allowing us to make decisions such as on allocating work or design problems concerning trade offs amongst potentially conflicting quality attributes before actual implementation. Thus it was advantageous to spend some time using a higher level design as we did.

We contemplated using behavioural diagrams to complement this structural diagram, however we felt secure enough to move onto the next stage of the Software Engineering process, since the class diagram captured our ideas coherently and simply enough for that stage and our program is relatively small scale.

Justification for Concrete Representation of Software 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 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 "Justification".

We used a class diagram form of structural modelling, as it is most applicable to the object oriented style used in our programming solution. It is clear that the class diagram builds from the abstract software architecture above and looks at the classes that are more specific to the code in more detail. The naming convention has also been kept the same from the abstract architecture as this allows us to clearly see the how we have built from it and causes less confusion. The tool used in order to make this diagram was the UML Class Diagram tool provided by the IntelliJ IDE.

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.

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:

- MainGameScreen: This will be where NFR_MOVEMENT_EXPLANATION and NFR_RULES_EXPLANATION will be satisfied.
- MainMenuScreen: Play a part in NFR_POSITIVE_UX. Now also satisfies FR_LOAD_GAME
- GameOverScreen: After FR_GAME_DURATION, the player’s outcome in the game is revealed, relating to NFR_END_SCREEN.
- Button: Linked to FR_INPUT_DETECTION, and this class shall process the input if it is a user click on a button.
- ScrollingBackground: This will allow the player to become invested in the simulation, and help build NFR_POSITIVE_UX.
- Boat: All objects that extend from this will be unique, relating to UR_BOAT_UNIQUENESS, and shall be described to the user as per NFR_ATTRIBUTES.
- PlayerBoat: Upon NFR_ATTRIBUTES being carried out, the player will choose their avatar, linking to the FR_CHOOSING_BOAT system requirement. It is controlled by standard WASD controls, relating to FR_INPUT_DETECTION and FR_MOVEMENT. If its health decreases according to numerous FR_HIT_DECREASED_BOAT_CONDITION on top of

FR_TIMED_DECREASED_BOAT_CONDITION, such that FR_BOAT_BREAKAGE occurs, game ends and NFR_END_SCREEN happens.

- ComputerBoat: An object of this class will be the player's competition in every race. Will be an object of type Boat, which was not chosen by the player in NFR_ATTRIBUTES.
- Obstacle: This is dependent upon FR_OBSTACLE_RATE and FR_OBSTACLE_SPAWN.
- ObstacleType: This class allows for variation in obstacles to make the player more invested in the game, aiding the fulfillment of NFR_POSITIVE_UX.
- Hitbox: This implements FR_COLLISION_DETECTION between object of type Boat and object of type Obstacle. If true (i.e hit), FR_HIT_DECREASED_BOAT_CONDITION occurs.
- Lane: Using this class, FR_BOUNDARY_DETECTION checks if the player is in lane, else FR_AWARD_PENALTY is carried out.
- Race: This shall be used to implement FR_FINAL_RACE and FR_QUALIFIER_RACES, and during its duration FR_DECREASED_BOAT_CONDITION shall be affecting the Boat objects.
- FinishLine: Upon reaching this before their opponent, a player will win that race. Provides a sense of achievement before the grand finale of the competition for NFR_POSITIVE_UX. Timings in FR_QUALIFIER_RACES determine whether FR_FINAL_RACE shall involve the player or not.
- BoatType: This is an ENUM class that holds the different predefined attributes of various boats, such as their relative staminas, healths, speeds and velocities - relating to NFR_ATTRIBUTES and FR_CHOOSING_UNIQUE_BOAT. This class makes it easier to add new boats to the game as values don't need to be redefined for each new boat created in the project. As a result, it will result in boats being less error prone.
- Entity: This ENUM class 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, instead they are just stored in the class - relating to NFR_ATTRIBUTES and FR_OBSTACLE_SPAWN and FR_CHOOSING_UNIQUE_BOAT.
- EntityType: This ENUM class makes it easier to add new boats to the game as values don't need to be redefined for each new boat created in the project. As a result, it will result in boats being less error prone. All attributes of the boats are predefined in that case - relating to NFR_ATTRIBUTES and FR_CHOOSING_UNIQUE_BOAT.
- DragonBoatRace: This class is used to initialise the game. This relates to UR_UX.
- Collidable: This class, linking to FR_POWERUP_RATE, controls the spawning of the powerups for the game, having 5 different powerups that can spawn.
- DifficultySelectScreen: Used to allow the selection of the difficulty, FR_DIFFICULTY_SELECTION, which in turn effects the spawn rate of the obstacles in the game, FR_OBSTACLE_RATE.
- PauseScreen: This is where FR_SAVE_GAME and FR_PAUSE_SCREEN are both satisfied.
- BoatSelectScreen: Allows FR_CHOOSING_UNIQUE_BOAT to be met.
- HelpScreen: NFR_RULES_EXPLANATION is displayed here.

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.