Team17 - Gnocchi Games

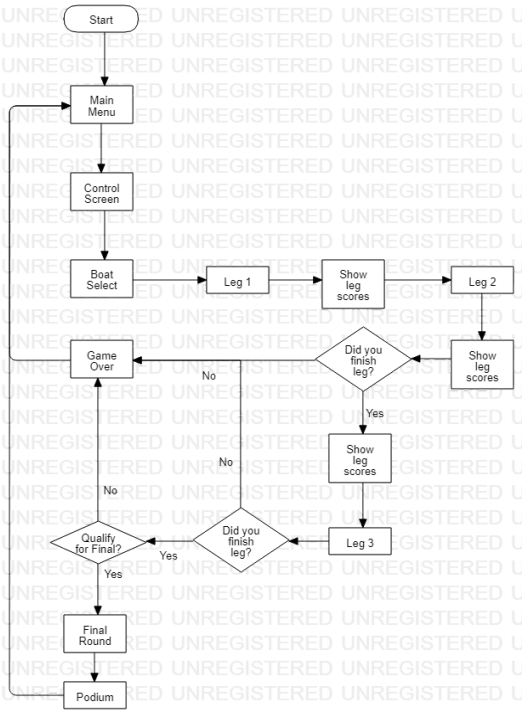
Henry Overton
Yousif Habib
Ben Dunbar
Lucy Newton
Adam Blanchet
Thomas Heenan

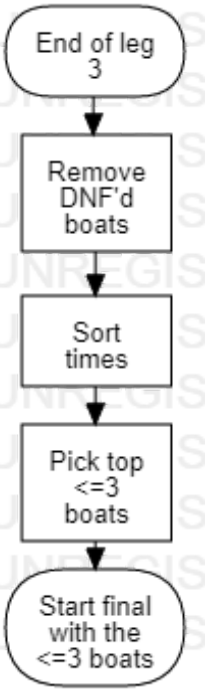
Architecture

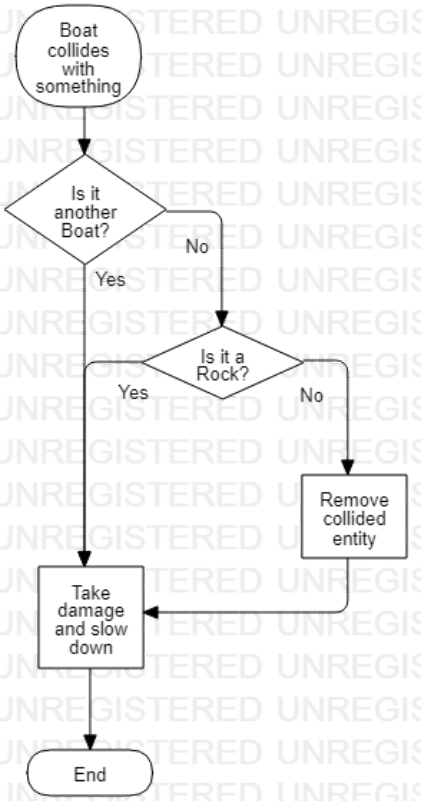
It was decided that the abstract architecture would be represented as a series of different diagrams, while the concrete architecture would just be a set of class diagrams detailing the methods and attributes as they will be implemented. Both are stored on the website.

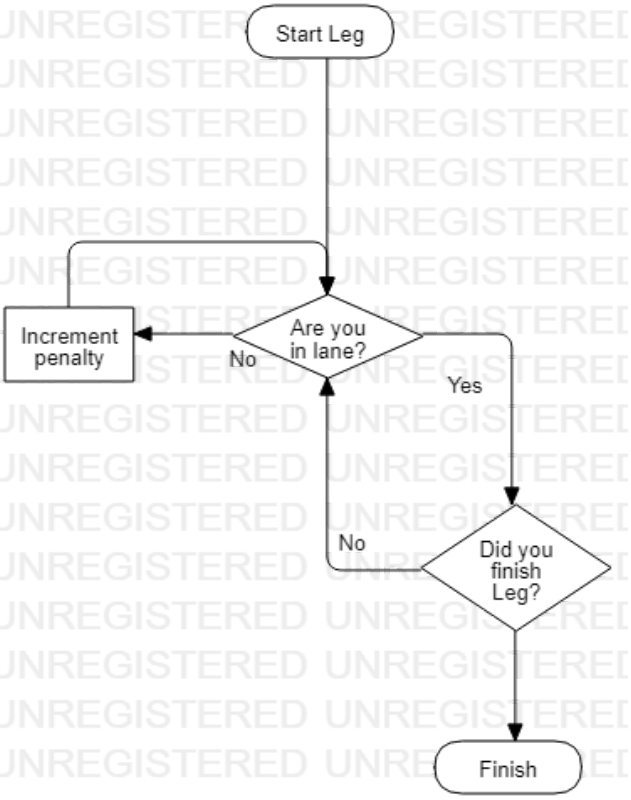
Abstract:

FlowCharts:



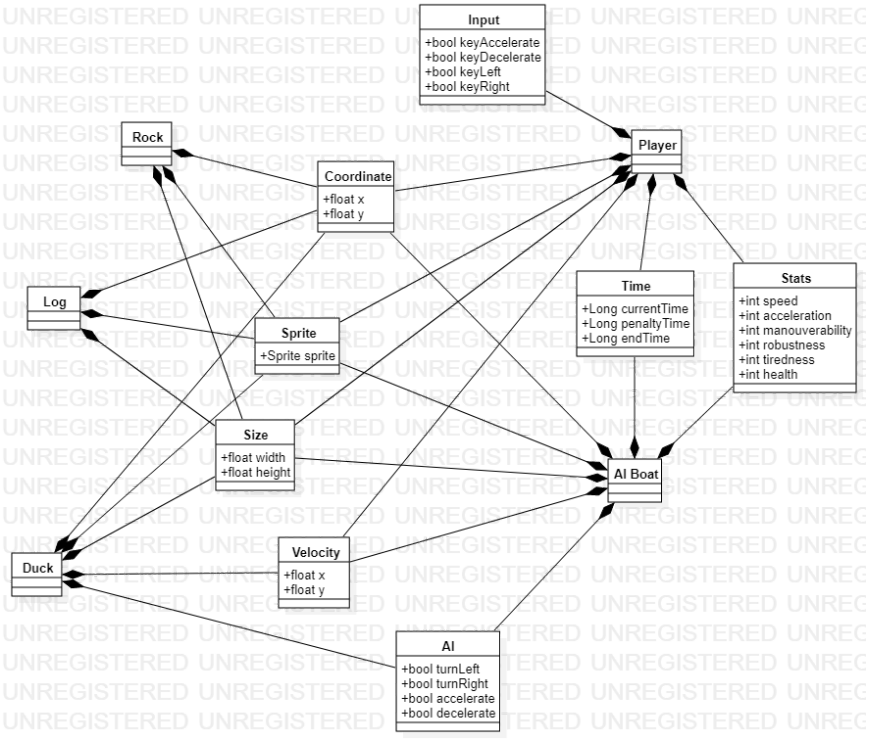






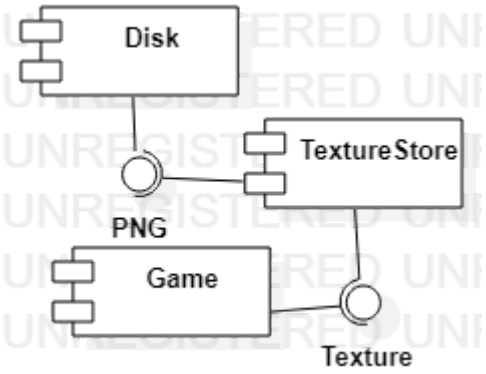
Flowcharts that show the logic the game follows. The first shows the different screens the game goes through. The second shows how we decided who qualifies for the final. The third shows how the game decides how the out of lane penalty is applied. The fourth shows how a collision with another entity will appear to the user.

Entities:



An entity-component diagram that shows all of the entities in the game as a congregation of base components.

Texture Handling:



A component diagram which shows how we will store and retrieve textures. The three components are the disk or storage medium, the game, and an intermediary texture store that handles taking the PNG files from the disk and providing them to the rest of the game system

Concrete: *(can be seen on the website as they're too big)*

The class diagrams show the individual classes and the inheritance hierarchy. Everything is shown as it will be implemented except for the omission of constructors, getters and setters. Any external class is represented as an interface in the diagram.

It is split into five sections which group similar classes together. "Entities" covers all the classes that inherit the "Entity" class, "Screens" shows all the different screens that are displayed by the game, "Utility" is for any classes that supplement other classes, "ScoreBoard" is for the "ScoreBoard" class and it's enumerations, and "Game" is for the main body of the game.

Due to some limitations of StarUML, things that would be implemented as a tuple in the actual code, are instead shown as a separate data type, and linked by an association to their respective class.

As the group had by now collected and arranged all of our requirements into User, Functional and Non-Functional Requirements we decided it was time to start thinking about turning the implementation into code. As the gap between code and requirements is so huge we decided to model our program using two architectures, abstract and concrete. Abstract is a low level of detail perspective on what the game's core fundamentals will be. Concrete, on the other hand, is a very detailed in-depth model of the entire game and reflects clearly how the game should be coded.

For our abstract architecture we began to brainstorm about the task and obtained valuable information through customer meetings. This led to entity-component diagrams, flowcharts, and component diagrams for the game being created. Throughout the abstract section our objective was to gain an understanding of how our game should play out and what as a team we would need to create and how long it would take. We thought about the game from a customer and user perspective when developing our ideas to try and keep to our user requirements.

One example of our abstract architecture is our entity-component diagram, where all the entities in the game are described as a congregation of components. This included the boat the player will use, computer controlled enemy boats, and the obstacles in the game. We decided on an entity-component diagram as having the classes be boiled down to their base components allowed us to easily understand how the entities would act and look like in the actual game, allowing for easy creation of classes in the concrete architecture. Another example would be our component diagram, which details how the game will store and retrieve textures for the sprites. It was decided that having an intermediary component would streamline the texture request system, and allow for easier memory management.

The concrete architecture builds off our abstract models and requirements directly, taking the modules created in our abstract models we further broke down every part to its lowest level. This was done using StarUML, a piece of software used to give a visual UML representation of classes and the associations between them. This would give us a very structured and in-depth outline of what to code when it came to implementation. We split the UML models into classes based on the modules, these were: Entities, Screens, Utility, Scoreboard, and Game which was the main class that called and directed all others. All of these classes link back to one or more of our requirements.

The Entities model as an example, starts with the class entity in which it is defined that all entities will have the attributes position, sprite, height, width and velocity. It also contains functions to detect collisions. There are two children of Entity; Boat and Obstacle which both inherit all the attributes and functions. The Boat subclass contains many more attributes, relating to stats and penalties, and functions to do with moving the boat. As for Obstacle it just contains the damage it will do. It also contains a few children, Rock and TreeLog which don't move and duck which moves horizontally across the screen. The Entities model covers many requirements such as UR_MOVE_BOAT, UR_OBSTACLES, UR_OBSTACLES_DAMAGE, UR_TIRED from our user requirements and all the related functional and nonfunctional requirements seen in our requirements table. Entities was also heavily influenced by entity-component diagrams that already had all of our entities and some basic attributes listed out. Using overlapping attributes such as position and Sprite we

were able to easily create the UML and added many more in depth attributes and functions relating to each entity.

We also drew up our flowcharts on StarUML in full describing all choices and possibilities we intended the game to have as well as all states it could reach. This allowed for simple logic to define where and how our user would move in a clear manner. It was however chosen to leave out certain details such as the controls and in leg choices the user would take as this was considered too intricate to create before implementation could be completed.

Other methods of concrete architecture were considered however we could not find another chart or diagram that fit our needs for example we considered a User Case Diagram that shows the interaction between users and different states of the game however with only one user and a handful of states the diagram would have served no more purpose than a flowchart which we had already created.

When designing both the concrete and abstract architectures we constantly had our requirements, specifically the user requirements as all functional and non-functional requirements lead back to them. To start, Game covers a lot of the requirements as it will be the class to hold the logic of the game and what user actions trigger changes in state. It ties into; UR_PLAYABLE, UR_RUNNABLE, UR_CHOOSE_BOAT, UR_COMPLETEABLE. As for the Entities class as stated above it was designed based on; UR_MOVE_BOAT, UR_OBSTACLES, UR_OBSTACLES_DAMAGE, UR_TIRED. For example; The AIBoat child class of Entities, was designed to guide the computer controlled boats through the level and satisfies; UR_AI, UR_FAIR, UR_DIFFICULT. The last group of classes is the screens which display all information not in the racing part of the game such as options, selecting a boat and the winning/ losing screens. These were designed with; UR_PLAYABLE, UR_CHOOSE_BOAT, UR_CASUAL, UR_INSTRUCTIONS and UR_COMPLETEABLE, in mind. The RaceLegScreen for example, will cover very similar requirements but is based on the the logic of the leg not the player controlling the boat so it covers; UR_COMPLETEABLE, UR_FAIR, UR_AI, UR_DIFFICULT, UR_TIME_PUNISHMENT, UR_OBSTACLES, UR_OBSTACLES_DAMAGE, UR_PRACTICE. A lot of our classes overlap in the requirements they meet meaning there are many different ways in which we are approaching the same requirements. This will allow us to satisfy all the requirements in one way or another.

We believe that all requirements will be implemented into our game by the end of the time we have. Any that are not met will be noted with a reason as to why not and how we would intend to implement them. All UML diagrams, Flowcharts and Requirements can be found through our website.