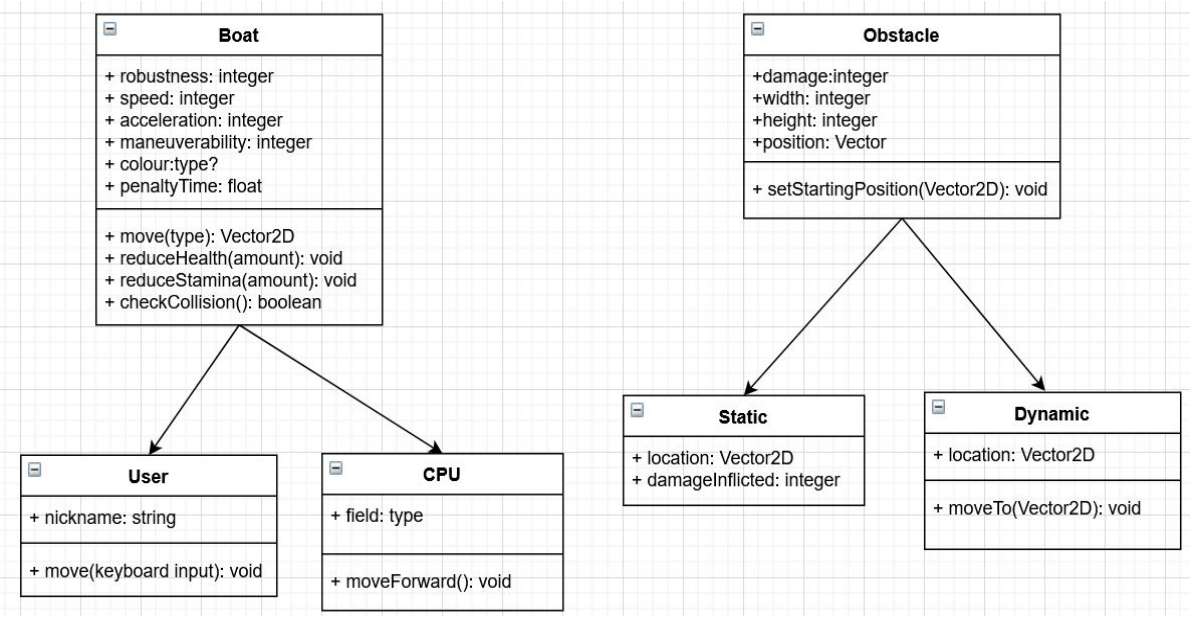
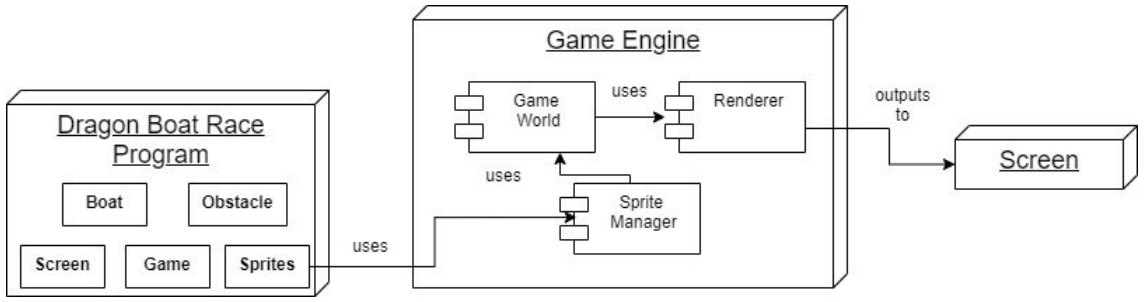
Abstract Representation of the Architecture

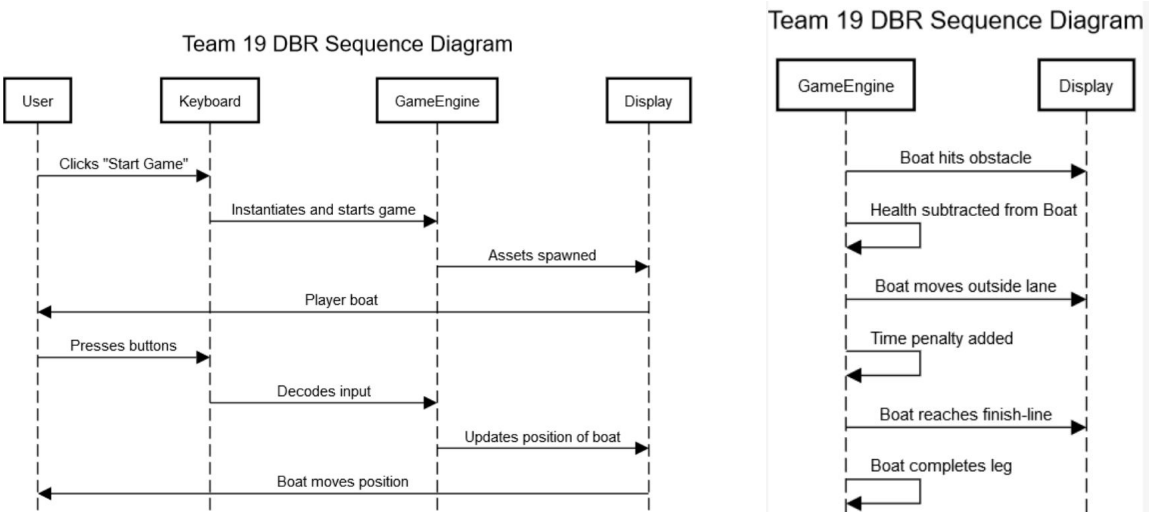
This is our abstract UML class diagram which highlights our main two classes of a boat and an obstacle plus their relevant variables and methods.



Our abstract UML component diagram below shows how the game, classes and sprites interact with the game engine so that the game can run and be displayed. The rendering engine part of LibGDX allows the game to be displayed to the user through the users screen.

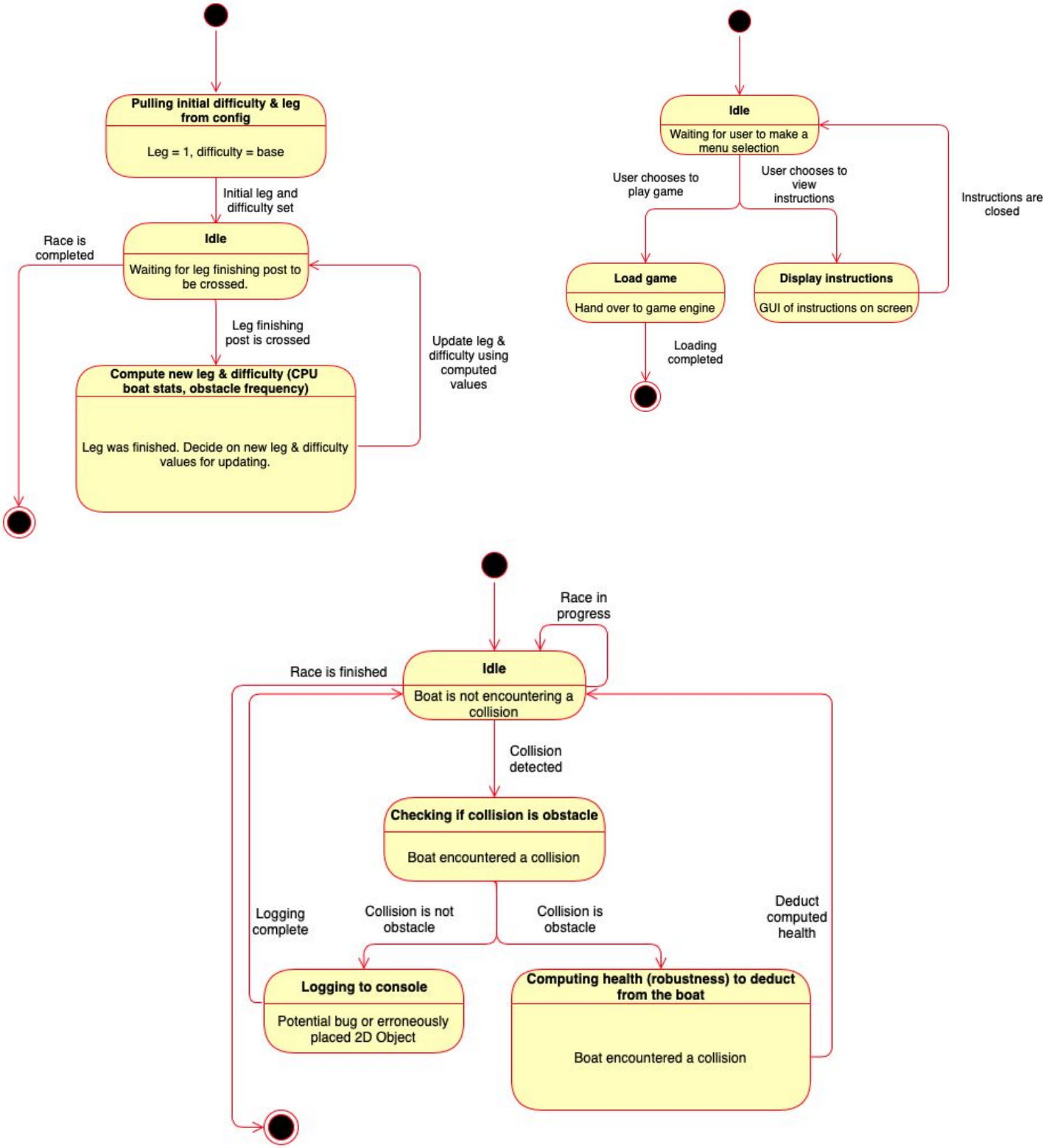


We also created an abstract sequence diagram which shows how we expect the user to interact with the game and the actions that take place throughout the game and how finally the game is displayed to the user



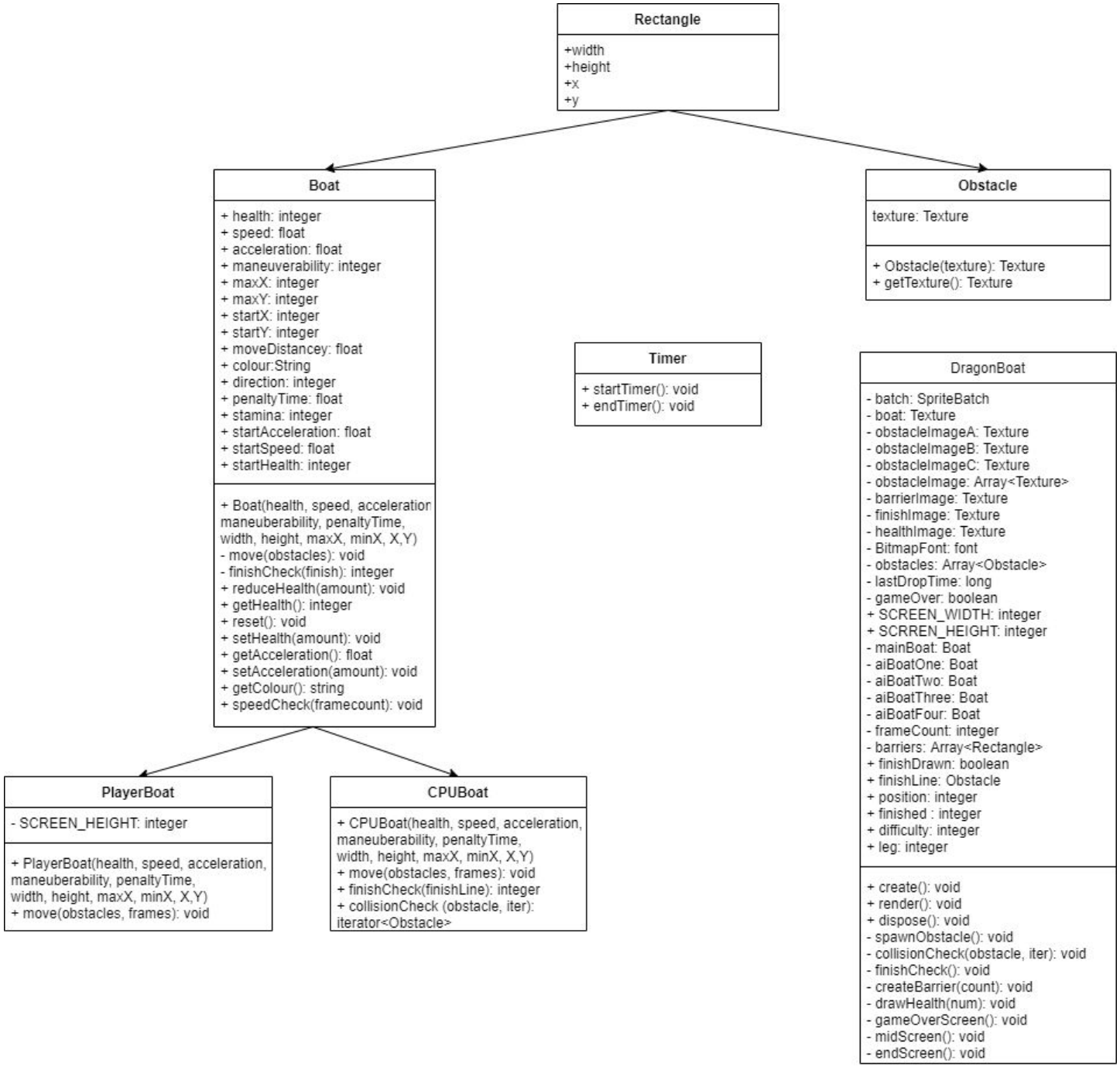
Concrete Representation of the Architecture

For our concrete representations we made a class diagram of all our classes that we are going to implement and also a state diagram(below) of the different states the game is in.



Here is our final class diagram, showing the inheritance of objects and all their attributes and methods. We introduced a DragonBoat class which controls the game such as checking for any collisions with an obstacle or to check if a boat has reached the finish line. It also sets the images of boats and obstacles.

Languages to describe these structures: we used plantUML and diagrams.net to create these diagrams. We found plantUML to be incredibly powerful in terms of quickly creating visual diagrams, but diagrams.net had real-time cooperative editing which we found incredibly useful for some of the more complex class diagrams.



Systematic Justification for the Abstract and Concrete Architecture

In Software Architecture there are two general types of models: prescriptive, and descriptive, which relate directly to the abstract and concrete levels of architecture respectively.
Thus, an abstract architecture is a non-implementation-specific outline of a system that is being designed, with only the core most important details mentioned (low level of detail).
Concrete architectures describe a system that already exists, reflecting the implementation decisions, and with more descriptive detail in general.

We will justify each individual architecture, as well as explaining how the concrete architecture builds on the abstract.

Abstract Architecture Justification

Abstract architectures allow the developer to design based on user requirements prior to implementation, without being affected by constraints from the infrastructure (such as the game engine). This produces a simplified design that can be expanded upon in the future.
For our abstract implementation we created a class diagram, component diagram and a sequence diagram.

Our abstract class diagram shows the initial classes we thought we would need: boat (FR_BOAT_PLAYER) and obstacle (UR_OBSTACLE). Initially, we thought this would be all we would need as the majority of the objects can inherit from these classes such as the player and CPU boats. Our component diagram highlights how our program will interact with the game engine and how the game engine will allow the game to be displayed to the player and all the components involved in doing this. Finally, the sequence diagram shows how different interactions between the user and the program allow for sequences of actions. For example, if the user clicks “Start Game” with a keyboard press, the game is started by the Game engine and assets are shown via the display.

From an abstract view, our diagrams successfully represent the core elements of the game, as well as showing the main classes and actions that need to be implemented concretely, allowing us to build on top of these diagrams to produce our concrete design.

Concrete Architecture Justification

The concrete design is more detailed, and focuses on final implementation decisions that are directly affected by the infrastructure used like the Game Engine and language chosen.
To display our concrete architecture we created a class diagram and a state diagram. Our class diagram builds on the abstract one we produced, and goes into more detail on inheritance and relationships between classes, as well as all of their attributes and their methods. It also shows which extra classes, attributes and methods we added that we didn’t previously identify or include in our abstract architecture.

This concrete class diagram introduces three new classes: DragonBoat, Rectangle and Timer. During implementation we decided that, to simplify our boats and obstacles, they should be represented by (and inherit from) the rectangle class, allowing us to use all of the attributes associated with a rectangle. This provided all the functionality that we needed and simplified our code. DragonBoat was introduced as the LibGDX-specific game class, which

performs important functions like initialisation of the game and boats (UR_GAME_IS_PLAYABLE), checking if the game is over (UR_GAME_OVER), managing difficulty (FR_DIFFICULTY), and distributing and checking for penalties (UR_FIXED_PENALTY). One thing we missed in our initial diagram was a timer, which this concrete architecture now contains, satisfying additional requirements (FR_TIMER_LEG and FR_TIMER).

We also decided to use a state diagram instead of a sequence diagram, as they allowed the creation of more detailed diagrams without loss of readability. The state diagram shows the different states that the game can be in, and how the user's interactions with the systems can move the game from state to state. This allows us to identify the various states of the game we must implement to fully satisfy our elicited requirements. It builds upon the abstract sequence diagram as it has a lot more of the states and actions with more detail that we may not have identified in our abstract design. For example in the abstract we just have "Boat hit obstacle" leads to "Health subtracted from Boat", whereas in the state diagram we have: a check for a collision (and then checking if the collision is with an obstacle), followed by the subtraction of health.