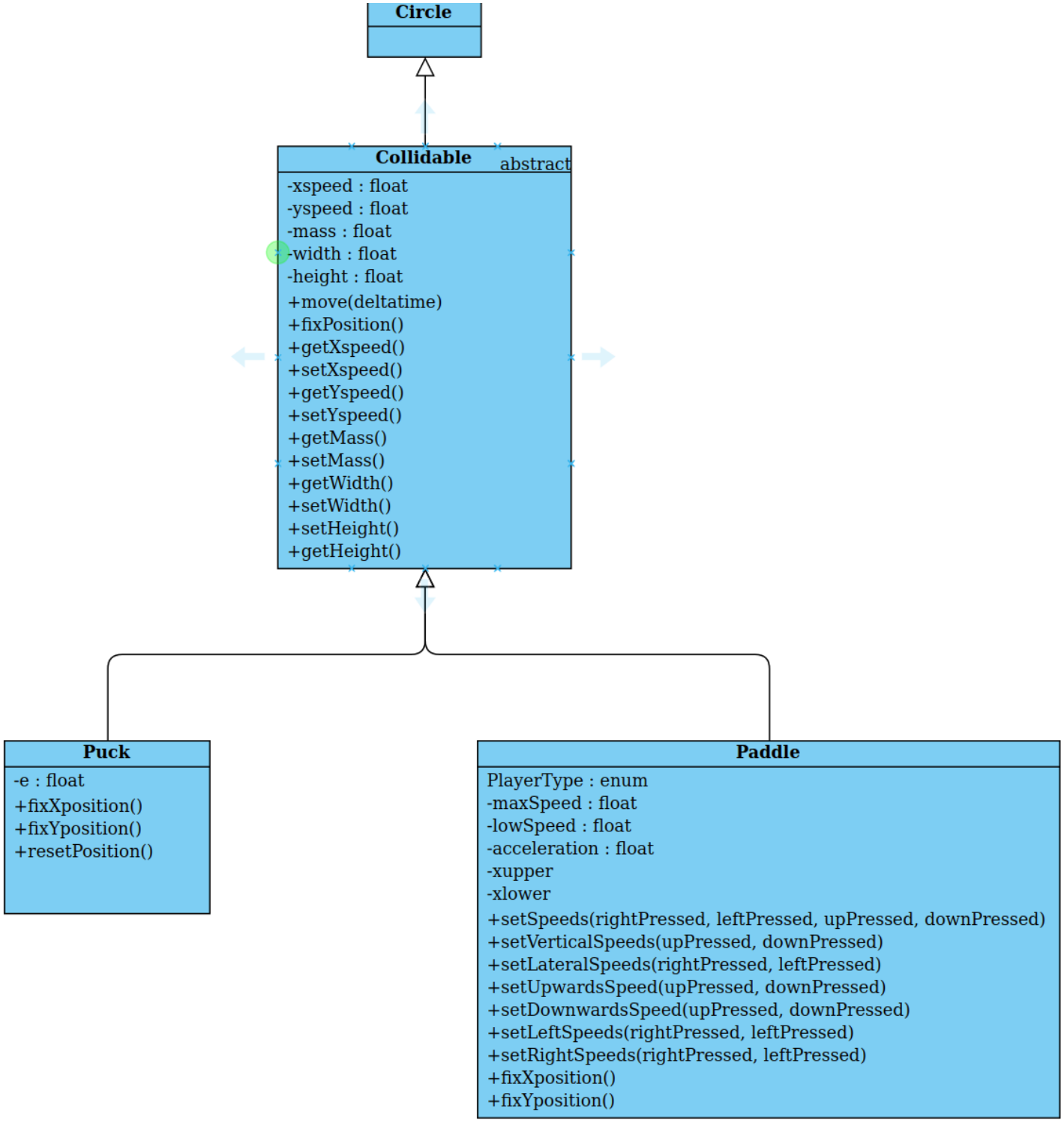
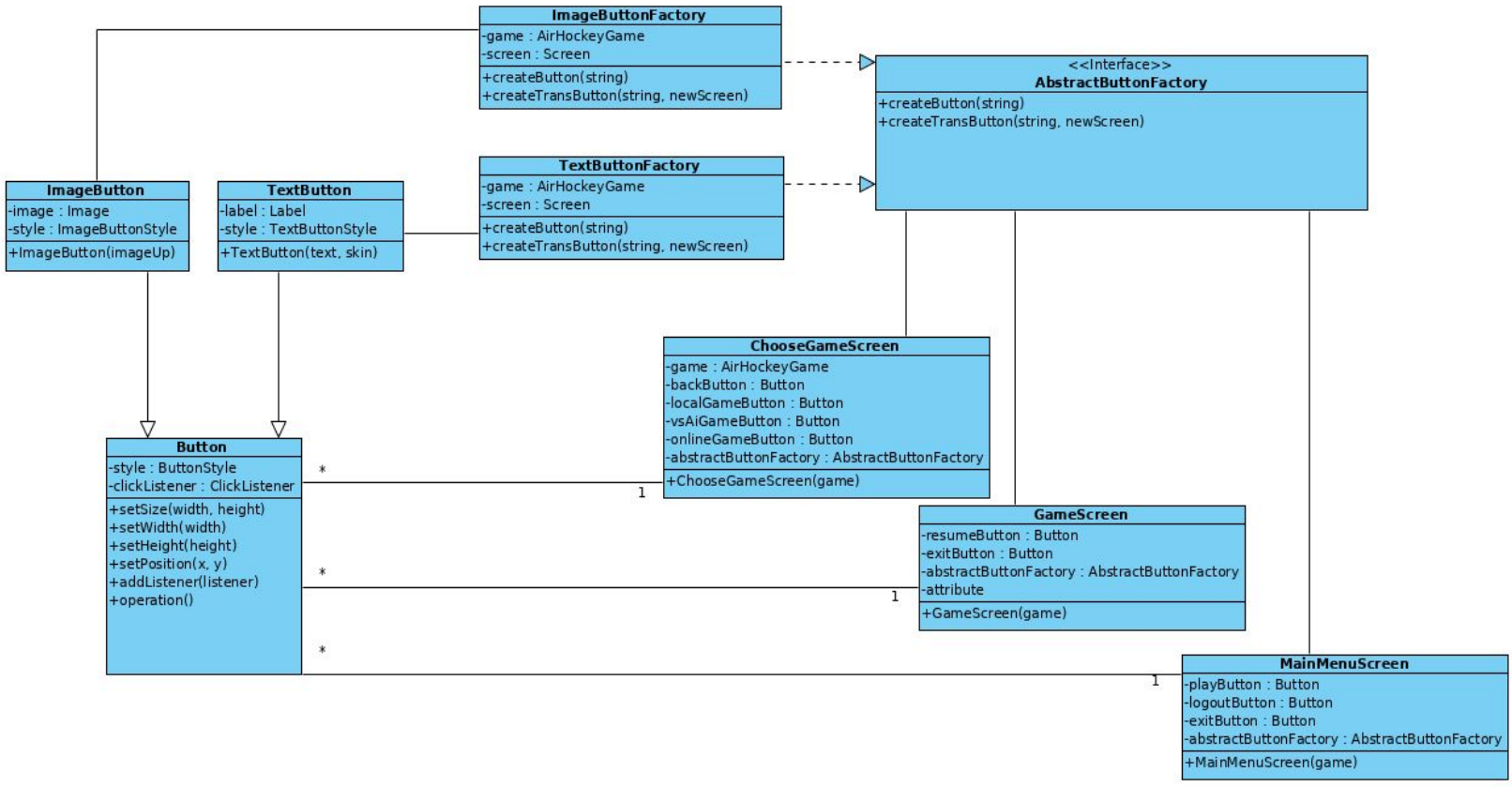
Why:
The puck and paddle have similar physical properties and behaviour in the real world, and hence should have similar properties and behaviour when implemented in our game. The puck and paddle are closely connected; they will collide with each other, and they will both be restricted by the boundaries of the board (where the game is played). But they also will differ in some regards (the puck can move around the whole board, whereas the paddles should be restricted to their own sides of the board). This led me to choose the template design pattern, as I could encapsulate shared behaviour, but also let the subclasses redefine parts of their behaviour that are unique.
I used this design pattern to implement invariant behaviour between the Puck and Paddle, and to factor out this common behaviour so that code is not duplicated (move method, fixPosition method and getters and setters of various common attributes).
I further used this pattern to define where the concrete subclasses had to do certain common tasks (such as fixing the X and Y co-ordinate positions), but the implementation would be different for each class. In this way these subclasses redefined certain parts of the behaviour that were necessary.

How:
The Collidable class is the Abstract SuperClass. The concrete classes are the Puck and Paddle classes. They are subclasses of the Collidable class. They inherit the attributes xspeed, yspeed, mass, width and height. (The super class Collidable itself inherits from libgdx' Circle class, hence inheriting the Circle class' behaviour and attributes).
The puck and paddle classes inherit the move, fixPosition, and various getter and setter methods for the shared attributes.
I used abstract methods to control subclass extensions. The abstract Collidable class defines two abstract methods fixXposition and fixYposition. These are methods that are implemented independently in the Puck and Paddle classes as their behaviour is slightly different in these respective classes. The paddle has further methods to set its speed, as this is the object that the user can directly control.


Circle
Collidable
abstract
-xspeed : float
-yspeed : float
-mass : float
-width : float
-height : float
+move(deltatime)
+fixPosition()
+getXspeed()
+setXspeed()
+getYspeed()
+setYspeed()
+getMass()
+setMass()
+getWidth()
+setWidth()
+setHeight()
+getHeight()
Puck
-e : float
+fixXposition()
+fixYposition()
+resetPosition()
Paddle
PlayerType : enum
-maxSpeed : float
-lowSpeed : float
-acceleration : float
-xupper
-xlower
+setSpeeds(rightPressed, leftPressed, upPressed, downPressed)
+setVerticalSpeeds(upPressed, downPressed)
+setLateralSpeeds(rightPressed, leftPressed)
+setUpwardsSpeed(upPressed, downPressed)
+setDownwardsSpeed(upPressed, downPressed)
+setLeftSpeeds(rightPressed, leftPressed)
+setRightSpeeds(rightPressed, leftPressed)
+fixXposition()
+fixYposition()

Abstract Factory Design Pattern for Buttons



There is only one of each screen (ChooseGameScreen, GameScreen and MainMenuScreen), each having buttons. Buttons can be of 2 types, ImageButton or TextButton. In order to create one you need to call the correct Button Factory (ImageButtonFactory for ImageButtons and TextButtonFactory for TextButtons). Both of those factories inherit the AbstractButtonFactory interface which defines 2 methods, one for creating a basic button and one for creating a screen transition button.

Exercise 2 – Assignment 3

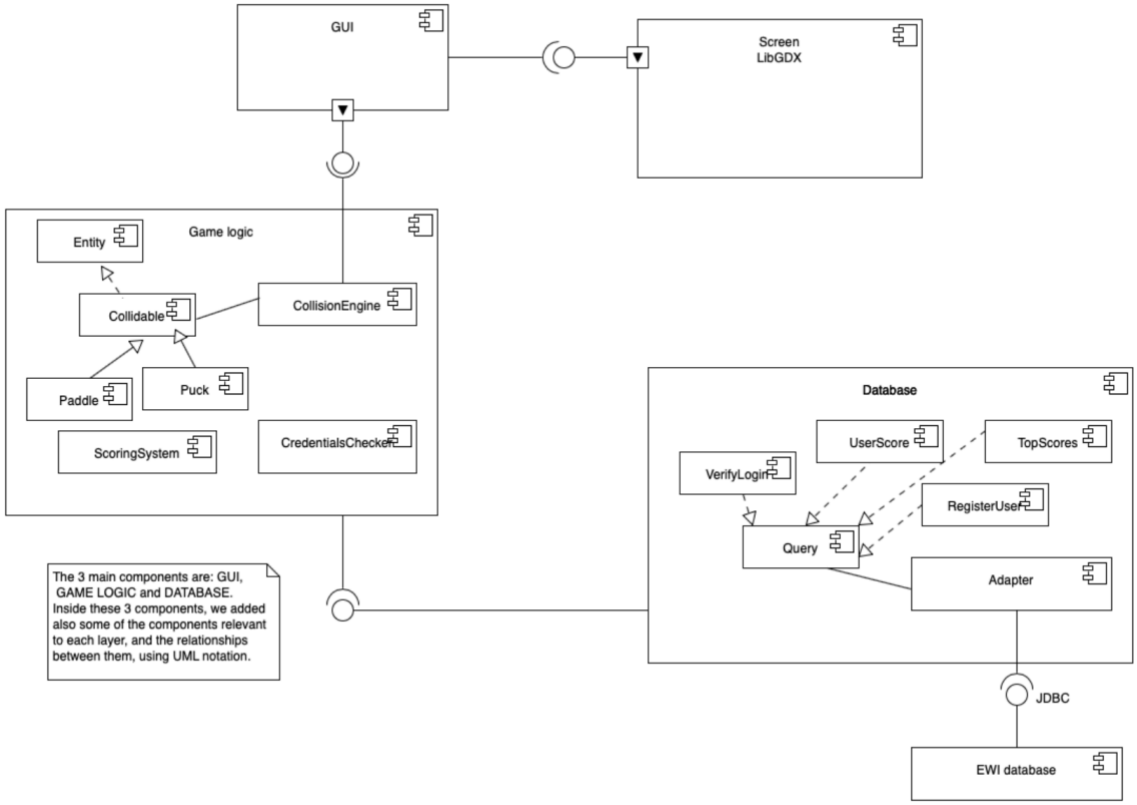
Software architecture

The architecture we chose is the Layered Architecture, so the game has 3 main parts (layers). Firstly, we have the graphical user interface which makes use of the LibGDX framework and of the Screen interface that this library provides. Secondly, there is the game logic framework that makes the connection between the GUI and the database, the last being the third and final layer.

GUI
The graphical user interface is designed to render the screens and change between them whenever the user requires that. It also takes the user's input, such as credentials and then calls the functions from the game logic layer to check this input with the database. This happens in the CredentialsChecker class. Given that LibGDX is the framework we work with, it would have been more difficult to use the Model-View-Controller architectural design, even though it is the most used one for desktop applications. The reason for this difficulty is given by the fact that LibGDX generates a lot of Screens and the Controller is hard to define for them.

GAME LOGIC
The second layer, the game logic has the functionality of being the connector between the GUI and the database. After the user enters his/her credentials, the CredentialsChecker verifies if they correspond to the data from the database. This layer also takes care of the game's physics. The class CollisionEngine, which makes use of the Paddle, Puck, Board and Goal classes (first two extend the Collidable abstract class, all of them implement the Entity interface), handles the intersection between the objects that are on the board.
In this layer, we also have an InputHandler interface which is implemented by each of the Paddle1InputHandler and Paddle2InputHandler classes. These are not shown in the component diagram since the collisions are more important for this layer. Furthermore, the input handler takes care of the user's moves and with the Direction class, these inputs are transformed in directions for the paddles.
There is also a scoring system that determines the winner of the game, given the time and the amount of points gathered by each player. In the scoring system component, there is also the HUD class, implementing the interface with the same name, which shows on the screen the time clock and the players' scores. The BasicScoreSystem class, that extends the abstract ScoringSystem calculates the score, adding 1 to the actual score every time a player scores, passing the new score to the HUD so it can be displayed on the screen.

DATABASE
The third and final layer, the database, makes use of the JDBC library and it is hosted on the TU Delft server. Its main purpose is being an interface for the interactions with the game data that is stored there, most often being used for verifying that a user exists, action happening every time a player logs in. The Query interface takes care of these, using prepared statements to avoid injection vulnerabilities. It makes use of the Adapter, a class with the purpose of connecting to the server. There is also a PreparedStatements class that contains only of getters of statements, used by the classes implementing the Query Interface. This class has the purpose of separating the actual SQL code and the execution of the queries, that happens in VerifyLogin, RegisterUser, UserScore, TopScores. Each of these classes has only one functionality, being designed following the Strategy Pattern.


GUI
Screen
LibGDX
Game logic
Entity
Collidable
CollisionEngine
Paddle
Puck
ScoringSystem
CredentialsChecker
Database
UserScore
TopScores
VerifyLogin
RegisterUser
Query
Adapter
JDBC
EWI database
The 3 main components are: GUI, GAME LOGIC and DATABASE. Inside these 3 components, we added also some of the components relevant to each layer, and the relationships between them, using UML notation.