Assignment 3

SEM Group 48

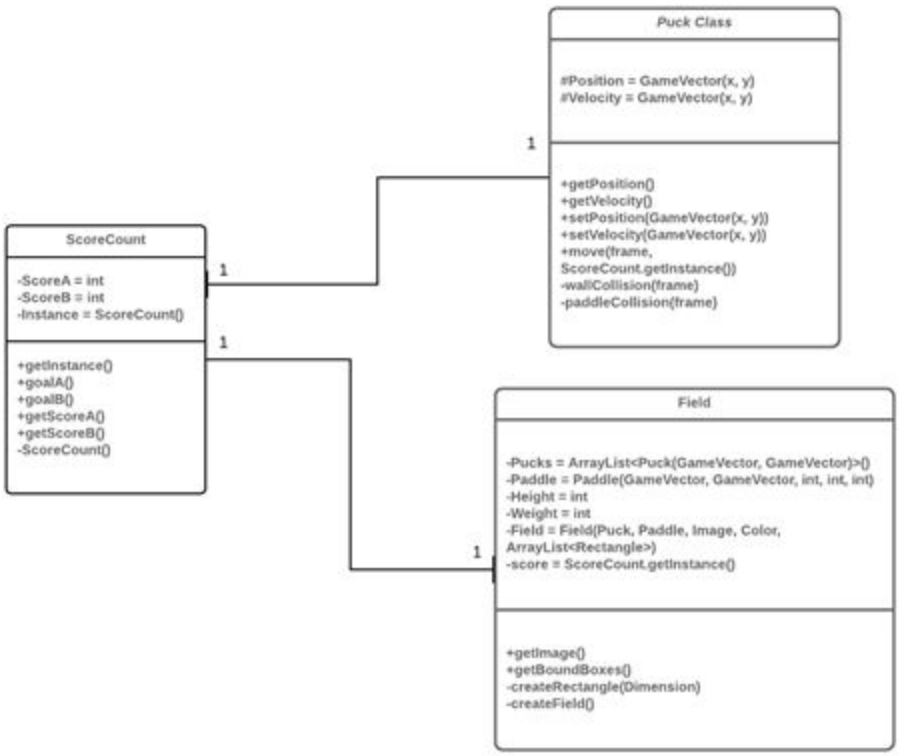
Exercise 1

The Singleton

Description:

We decided to make the class that counts the score in the game the singleton class. We ended up doing this because as it was implemented previously it at to be passed then 3 different classes, while only the last needed it, and the other 2 in the chain did not. We also knew we only wanted one instance of score at any given moment since multiple instances could create problems while counting. We did make the constructor private in the score class and then creating an instance with in it you could get from anywhere with the getInstance method. This way only one can exist at a time and it doesn't have to be passed around everywhere.

Class diagram:



The Builder

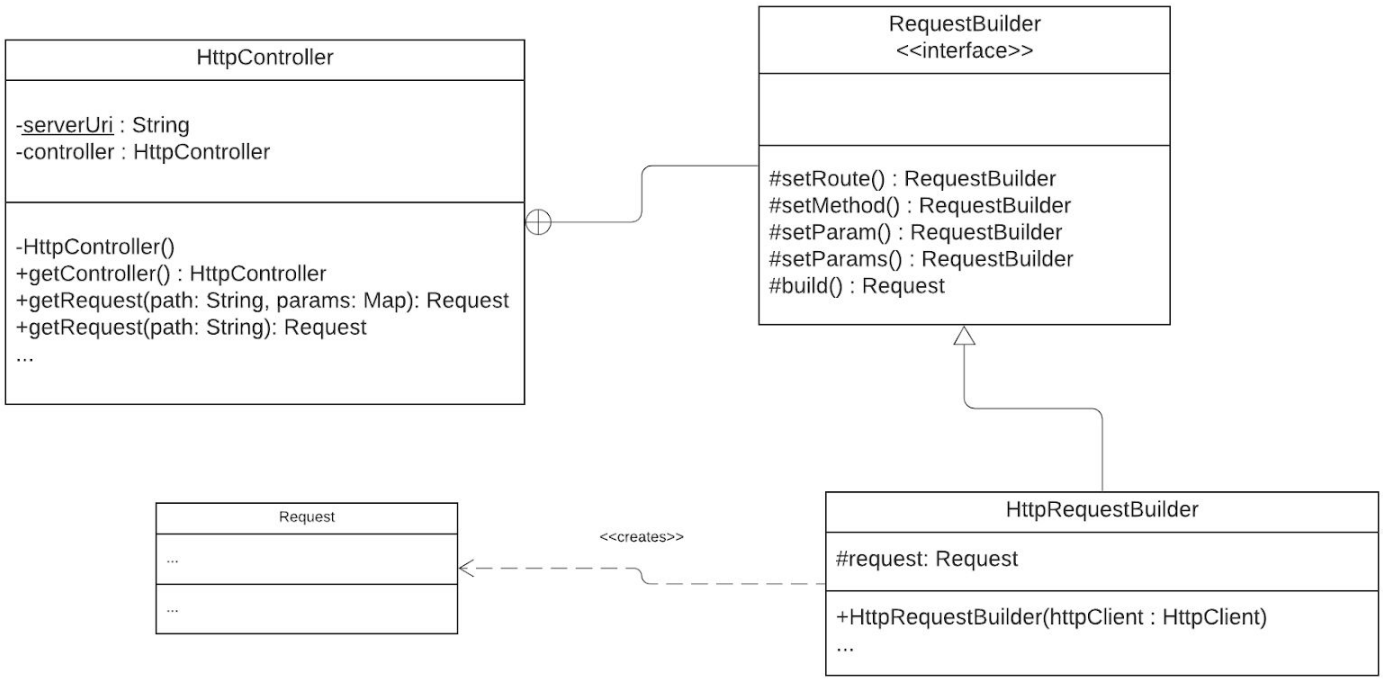
Description:

The builder pattern, in this case, makes it easier to generate and send an HTTP request, and it allows us to replace HTTPClient with minimal refactoring, we only need to alter the concrete builder, while we keep the rest of the client the same.

In this way, we separate the construction of the request, which is constructed by the Builder, from the controller which handles sending the request and receiving the response. It also decouples creating requests from the HttpController.

The HttpRequestBuilder (RequestBuilder) is used by the singleton HttpController to build a new request whenever getRequest (or similar) are called.

Class diagram:



Exercise 2

Sub Systems Descriptions

Basis: The first subsystem of the project is the basis package which contains all the low level classes needed for the working game. It contains the ScoreCount class to keep track of the game score, the rectangle class to work with collisions, the GameVector class to work with objects coordinates, velocity, etc and the puck and paddle object (which are made of the GameVector class) that extend the MovingEntity class.

Game: Another subsystem is the game package, it contains the high level classes necessary for the working game. They use classes from the basis package in order to have a working game. The field class contained in the game package which creates a field to play on uses pucks and paddles from the basis package. The Frame class which creates a frame to draw everything on uses the field class and thus pucks and paddles as well. The Game class which creates a game uses the field class and is also part of the game package as well as the MatchSocketHandler class that takes care of websockets connections.

App: The app subsystem contains all the database related components. In this package we take care of the database connection, login, registration, the friend system as well as the friend and general leaderboard, etc.

GUI package : The GUI package contains all the controllers which take care of the button clicks the user makes and handle the text the user enters. For each FXML file there is a unique controller class. We have a main screen which takes the user to either the login screen or the register screen depending on what button the user clicks. The login and register controllers handle the credentials the user enters and query them in the database. When the user is logged in the user can add friends, see the leaderboard and start a game.

Motivation for the use of the Model - View - Controller Pattern

The choice of the architectural pattern was very intuitive because first of all, the way we split the tasks is the same as the three layers from the MVC pattern. Some of us took care of the GUI part which corresponds to the View element, some of us implemented the game logic and services which correspond to the Controller element and finally some of us worked on the database, connections with it, etc which corresponds to the Model element of the MVC pattern. It makes it easier and faster for multiple programmers to collaborate together, we can work on different parts of the application without affecting other parts whereas the client-server architecture for example, is a centralized architecture and thus lacks in robustness.

Moreover, our game logic corresponds well to this pattern. As an example, if a user connects to the application and wants to retrieve the leaderboard of all best players in the game, the user will 1st click on the leaderboard button in the menu of the GUI (View), the control receives the request and will forward it to the database (Controller). Then the model will check the request (Model) and send the results back to the controller which will forward it to the view (Controller) where the user will see the data (View). This process is observed in all our application, when authenticating, retrieving friend stats, launching a game, etc and thus it is a reason why we chose this pattern.

Finally, by using this architectural pattern, we can also consider having multiple views for a single model which offers a lot of possibilities that we don't have in other patterns.