Student: Șinca Mădălina
Group: 30433

Table of Contents

1. Requirements Analysis

1.1 Assignment Specification

1.1.1 Objective

The objective of this assignment is to allow students to become familiar with architectural patterns.

1.1.2 Application Description

Use JAVA/C# API to design and implement an application for a ping-pong association that organizes tournaments on a regular basis. Every tournament has a name and exactly 8 players (and thus 7 matches). A match is played best 3 of 5 games. For each game, the first player to reach 11 points wins that game, however a game must be won by at least a two-point margin. The application should have two types of users: a regular user represented by the player and an administrator user. Both kinds of uses have to provide an email and a password in order to access the application.

The regular user can perform the following operations:
- View Tournaments
- View Matches
- Update the score of their current game. (They may update the score only if they are one of the two players in the game. The system detects when games and matches are won)

The administrator user can perform the following operations:
- CRUD on player accounts
- CRUD on tournaments: He creates the tournament and enrolls the players manually.

1.1.3 Application Constraints

● The data will be stored in a database. Use the Layers architectural pattern to organize your application. Use a domain logic pattern (transaction script or domain model) / a data source hybrid pattern (table module, active record) and a data source pure pattern (table data gateway, row data gateway, data mapper) most suitable for the application.
● All the inputs of the application will be validated against invalid data before submitting the data and saving it in the database.

1.1.4 Requirements

- Create the analysis and design document (see the template).
- Implement the application.
- Write at least one Unit Test for each method in the business layer (e.g. Test that a game ends if the score is 11-8).

1.1.5 Deliverables

The following files will be uploaded on your personal github account in a new repository:
- Analysis and design document.
- Implementation source files.
- SQL script for creating and populating the database with initial values.
- Build file (e.g. pom.xml)

The link to this repository must be sent to vlad.buzea@student.utcluj.ro after presenting the application at the laboratory. In the absence of this email the grade of the assignment is 0, regardless of the circumstances.

1.2 Functional Requirements

The functional requirements for this project are:

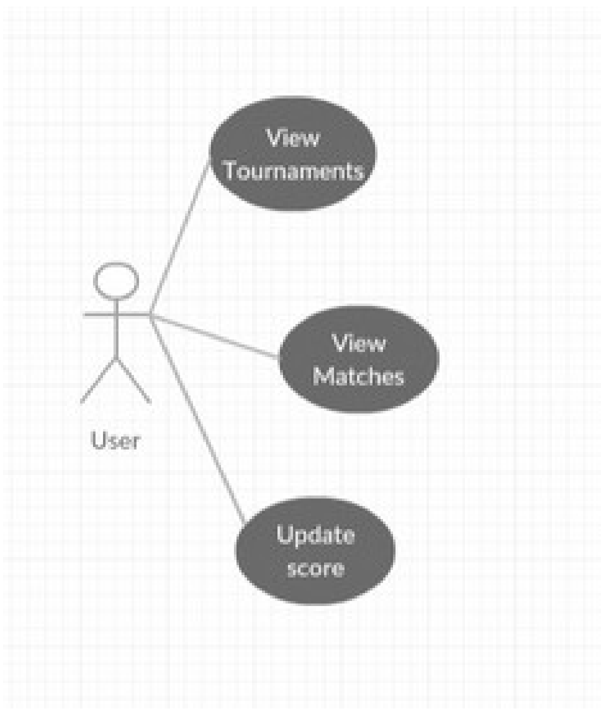
- Login for both Players and Admins
- Find and CRUD operations on accounts
- Find and CRUD operations on tournaments
- View tournaments
- View matches
- Update score and detecting when games and matches end
- Junit tests
- Implementing using a 3-tier architecture

1.3 Non-functional Requirements

The non-functional requirements for this project are:

- Performance: a response time of maximum 2 seconds
- Availability: 16hours/day
- Security: the passwords will be encrypted when sent to the database
- Testability: hardest part to test is the ending of the match/game and update score, this is done by both printing out results and Junit tests
- Usability: the application will work on any computer that has java installed

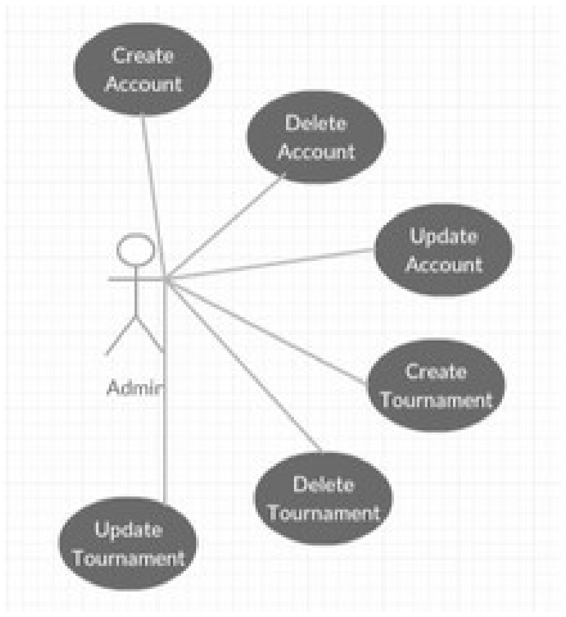
2. Use-Case Model



Use case: View Tournaments
Level: user-goal level
Primary actor: User
Main success scenario:

- open application
- enter login information
- click the login button
- user pane opens
- choose the view tournament button and click it
- view the tournaments

Extensions: wrong login information, and error is shown, have to retype the information



Use case: Create Tournament
Level: user-goal level
Primary actor: Admin
Main success scenario:

- open application
- enter admin login information
- admin goes to the create tournament pane by clicking the corresponding button
- introduces the information required for creating the tournament
- clicks the create tournament button

Extensions: wrong/missing information, and error is shown, have to retype the information

3. System Architectural Design

3.1 Architectural Pattern Description

The three-tier architecture is a special case of the Multitier architecture, for the three-tier we have 3 distinct levels: Data Access Level, Business Level and the User Interface Level, this architecture allows any one of the three tiers to be upgraded or replaced independently.[2]

In the Data Access Layer, we have the model-classes of the tables from the database where we just define the attributes, the database-connection class and the DAO-classes where the find and CRUD operations are implemented (also here you can find the DAO interfaces).

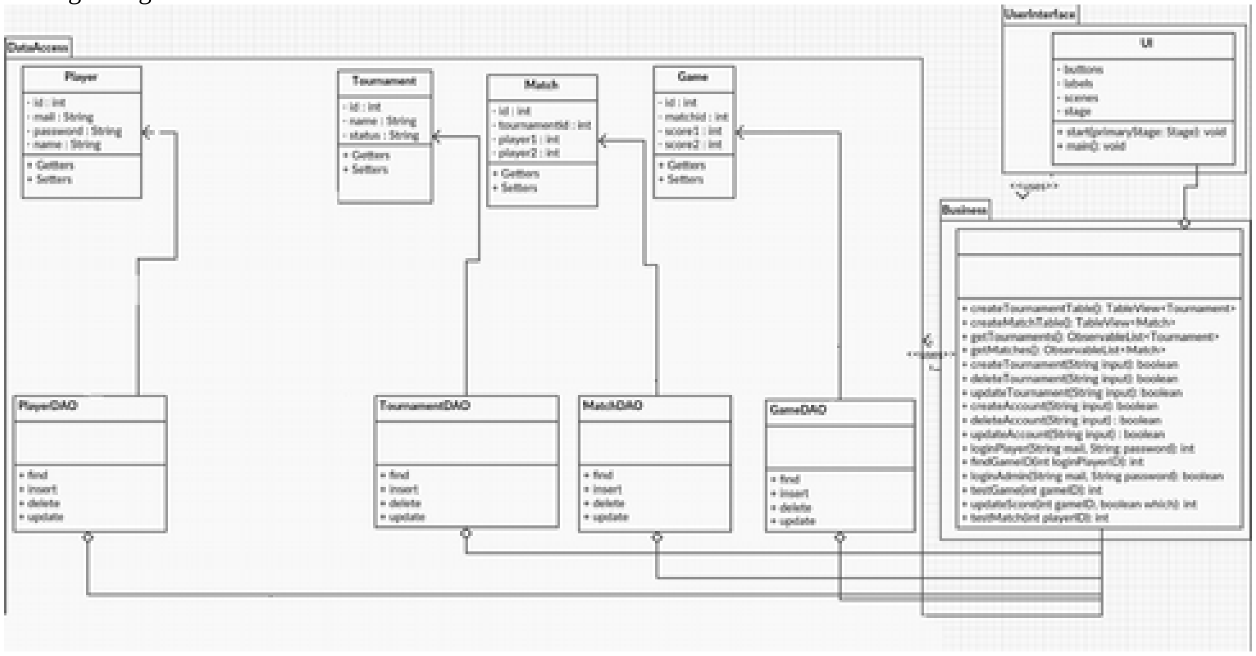
In the Business Layer, we have a class containing all the functions that use the Data Access Layer, like login which uses the find method.

In the User Interface Layer, in my implementation there is one class that contains all the components of the user interface like labels, buttons, scenes, panes, etc. and their attributes.

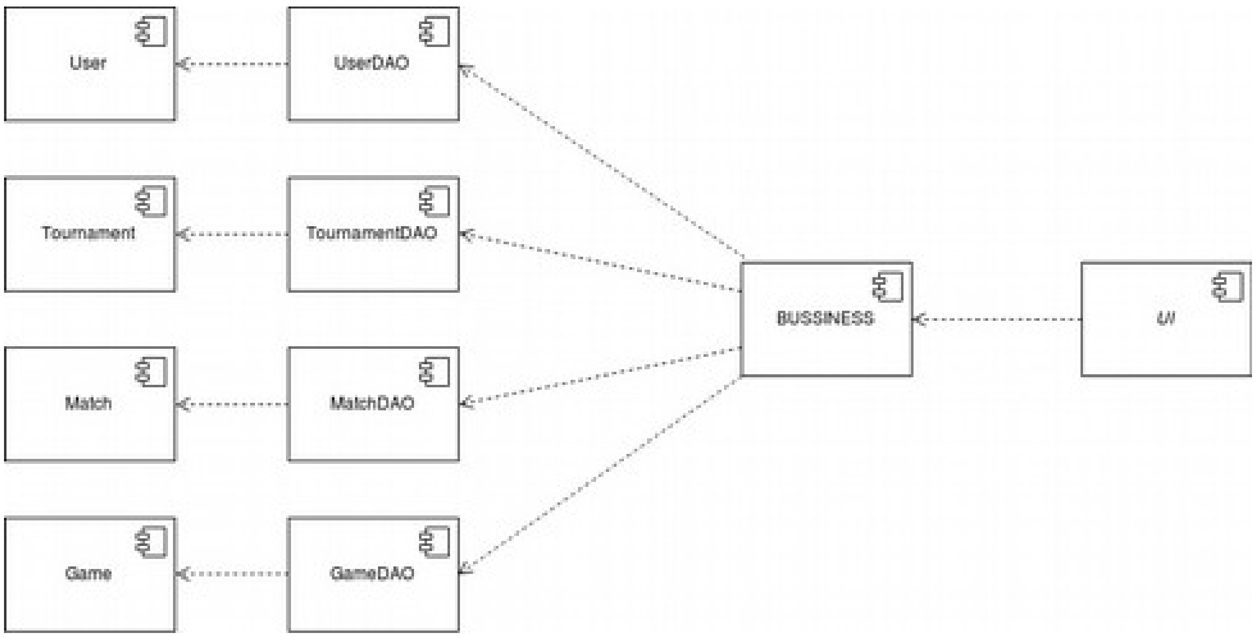
3.2 Diagrams

The main architectural pattern used in this project is the three-tier architecture, the use of it can be clearly seen in the package diagram where the classes are divided into 3 packages: DataAccess, Business and UserInterface, it can be seen that there are no relations between the UserInterface and DataAccess, all operations between these 2 are done using the Business package.

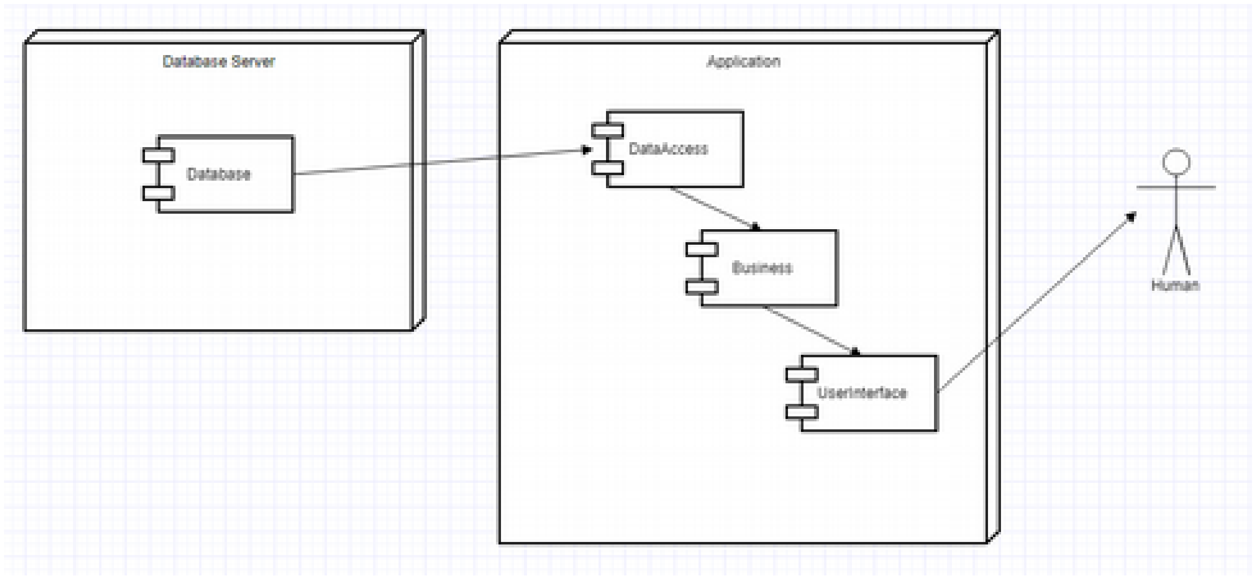
Package Diagram


DataAccess
Player
- id : int
- mail : String
- password : String
- name : String
+ Getters
+ Setters
Tournament
- id : int
- name : String
- status : String
+ Getters
+ Setters
Match
- id : int
- tournamentId : int
- player1 : int
- player2 : int
+ Getters
+ Setters
Game
- id : int
- matchId : int
- score1 : int
- score2 : int
+ Getters
+ Setters
PlayerDAO
+ find
+ insert
+ delete
+ update
TournamentDAO
+ find
+ insert
+ delete
+ update
MatchDAO
+ find
+ insert
+ delete
+ update
GameDAO
+ find
+ insert
+ delete
+ update
UserInterface
UI
- buttons
- labels
- scenes
- stage
+ start(primaryStage: Stage): void
+ main(): void
<<uses>>
Business
+ createTournamentTable(): TableView<Tournament>
+ createMatchTable(): TableView<Match>
+ getTournaments(): ObservableList<Tournament>
+ getMatches(): ObservableList<Match>
+ createTournament(String input): boolean
+ deleteTournament(String input): boolean
+ updateTournament(String input): boolean
+ createAccount(String input): boolean
+ deleteAccount(String input) : boolean
+ updateAccount(String input) : boolean
+ loginPlayer(String mail, String password): int
+ findGameID(int loginPlayerID): int
+ loginAdmin(String mail, String password): boolean
+ testGame(int gameID): int
+ updateScore(int gameID, boolean which): int
+ testMatch(int playerID): int
<<uses>>

Component Diagram

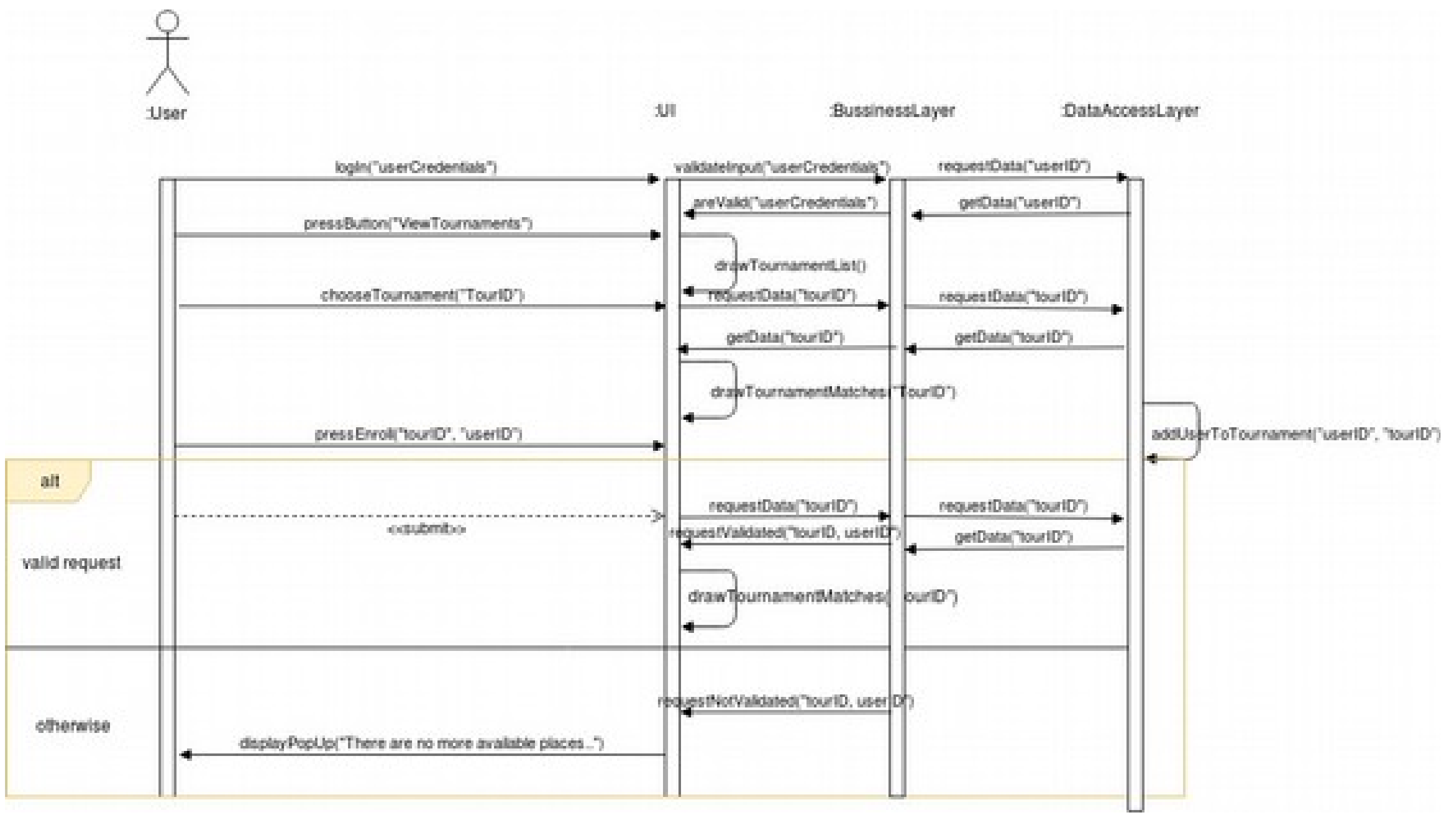


Deployment Diagram



4. UML Sequence Diagrams

The following diagram is a Sequence Diagram for when a player wants to enroll into a tournamment:


:User
:UI
:BussinessLayer
:DataAccessLayer
login("userCredentials")
validateInput("userCredentials")
requestData("userID")
areValid("userCredentials")
getData("userID")
pressButton("ViewTournaments")
drawTournamentList()
chooseTournament("TourID")
requestData("tourID")
requestData("tourID")
getData("tourID")
getData("tourID")
drawTournamentMatches("TourID")
addUserToTournament("userID", "tourID")
pressEnroll("tourID", "userID")
alt
valid request
<<submit>>
requestData("tourID")
requestData("tourID")
requestValidated("tourID, userID")
getData("tourID")
drawTournamentMatches("tourID")
otherwise
requestNotValidated("tourID, userID")
displayPopUp("There are no more available places...")

5. Class Design

5.1 Design Patterns Description

The three-tier architecture is an architecture in which the functional process logic, data access and user interface are developed and maintained as independent modules on separate platforms. Three-tier architecture is a software design pattern and a well-established software architecture.

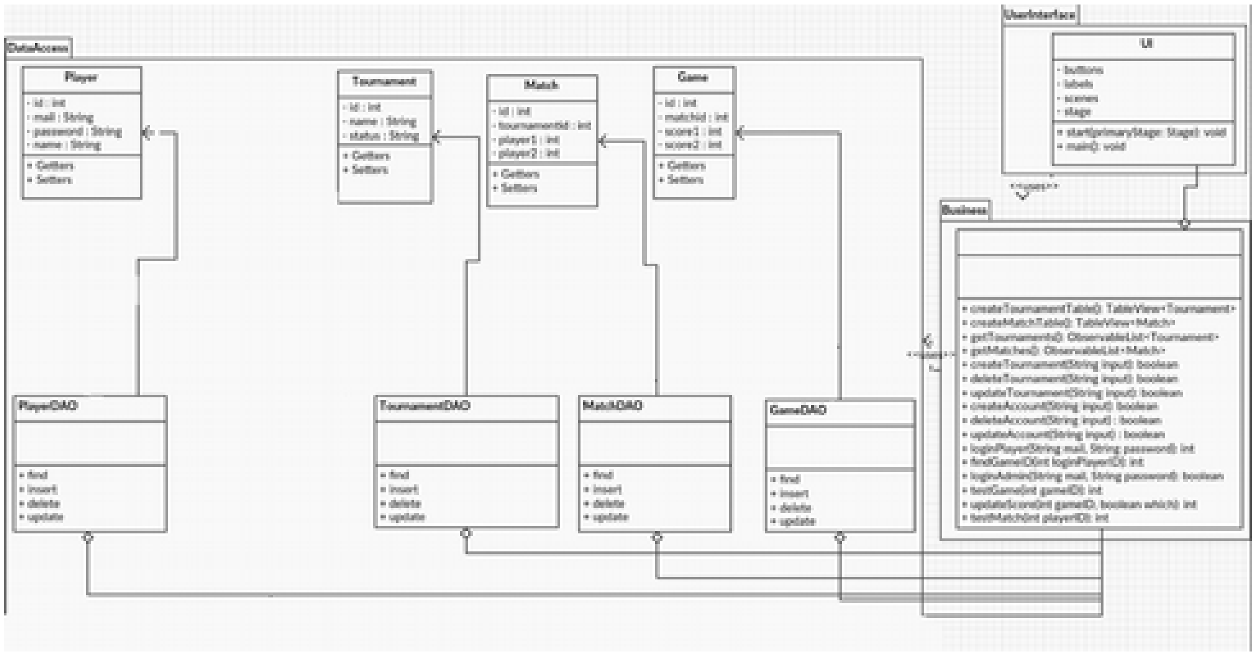
Data Access Object Pattern or DAO pattern is used to separate low level data accessing operations from high level business services. Following are the participants in Data Access Object Pattern.

- **Data Access Object concrete class -** This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.
- **Model Object or Value Object -** This object is simple POJO containing get/set methods to store data retrieved using DAO class.

A Domain Model Design Pattern is one that creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line on an order form.[3]

5.2 UML Class Diagram

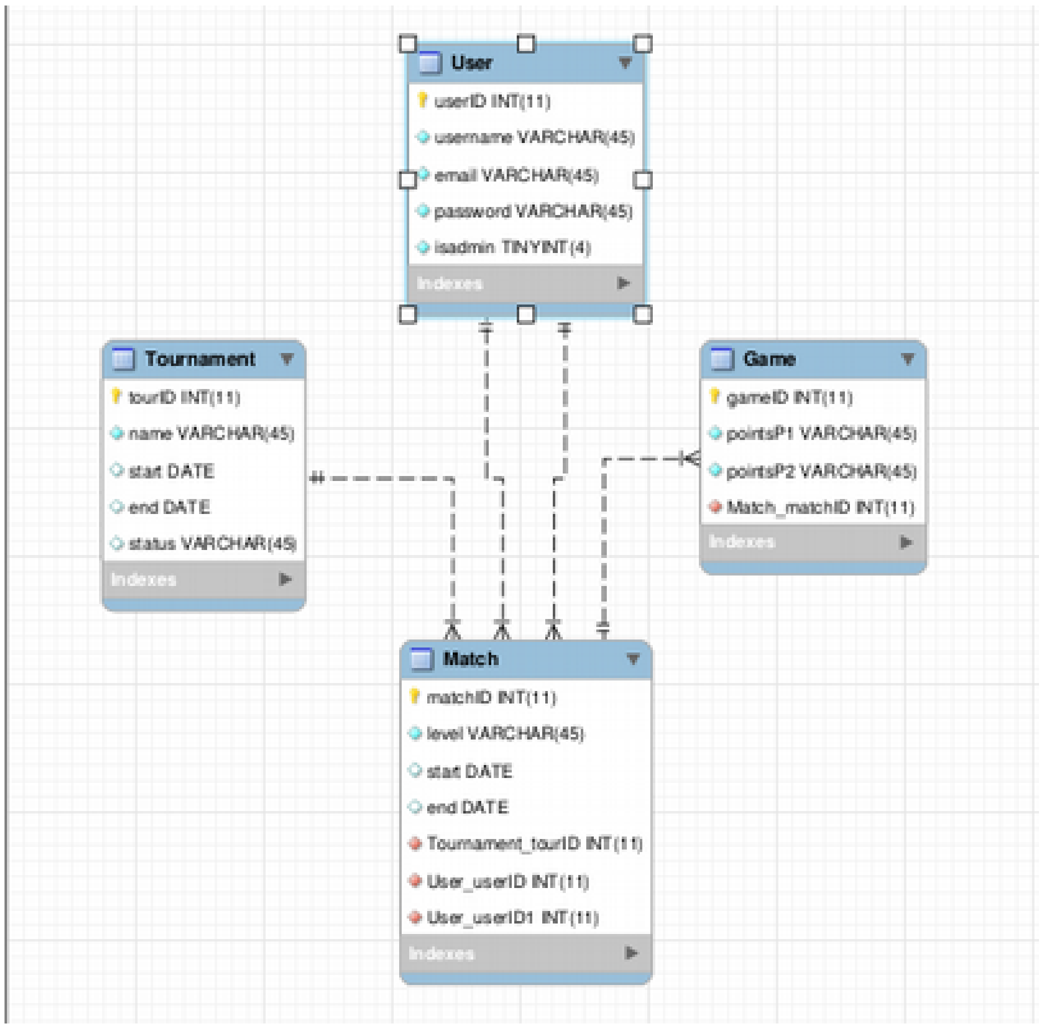
The User Interface layer contains only the GUI class, the Business layer only the Functions class, and the Data Access all the rest, it can be seen in the diagram that there are relations only between the User Interface and Business and between Business and Data Access.


DataAccess
Player
- id : int
- mail : String
- password : String
- name : String
+ Getters
+ Setters
Tournament
- id : int
- name : String
- status : String
+ Getters
+ Setters
Match
- id : int
- tournamentId : int
- player1 : int
- player2 : int
+ Getters
+ Setters
Game
- id : int
- matchId : int
- score1 : int
- score2 : int
+ Getters
+ Setters
PlayerDAO
+ find
+ insert
+ delete
+ update
TournamentDAO
+ find
+ insert
+ delete
+ update
MatchDAO
+ find
+ insert
+ delete
+ update
GameDAO
+ find
+ insert
+ delete
+ update
UserInterface
UI
- buttons
- labels
- scenes
- stage
+ start(primaryStage: Stage): void
+ main(): void
<<uses>>
Business
+ createTournamentTable(): TableView<Tournament>
+ createMatchTable(): TableView<Match>
+ getTournaments(): ObservableList<Tournament>
+ getMatches(): ObservableList<Match>
+ createTournament(String input): boolean
+ deleteTournament(String input): boolean
+ updateTournament(String input): boolean
+ createAccount(String input): boolean
+ deleteAccount(String input) : boolean
+ updateAccount(String input) : boolean
+ loginPlayer(String mail, String password): int
+ findGameID(int loginPlayerID): int
+ loginAdmin(String mail, String password): boolean
+ testGame(int gameID): int
+ updateScore(int gameID, boolean which): int
+ testMatch(int playerID): int
<<uses>>

6. Data Model

For this project, the data was modeled in the following way:
Player: id, mail, password, name
Admin: id, mail, password, name
Tournament: id, name, status
Match: id, tournament id, player1 id, player2 id
Game: id, match id, score of player1, score of player2

All models have id to be able to look for instances more easily, the id is the primary key in the database.
Admin and Player have mail and password for login and a name because it is a human custom.
Tournaments have a name and status.
Match contains the id of the tournament they are a part of and the id of the players that are playing
Game contains the id of the match (a match ends at 3/5 game wins) and of course the score of each player to know when the game ends



.

7. Bibliography

[1] Project idea and tutorials: https://github.com/buzea/SoftwareDesign2018
[2] 3-tier architecture: https://en.wikipedia.org/wiki/Multitier_architecture
[3] DAO pattern: https://martinfowler.com/eaaCatalog/domainModel.html