Ping-pong Tournament Manager 2
Analysis and Design Document

Student: Han Ana-Maria
Group: 30433

Table of Contents

1. Requirements Analysis

1.1 Assignment Specification

Since the Ping-Pong Tournaments application is such a big success, the owners of the application wish to add a new feature to it: Paid Tournaments. They differ from the free tournaments available until now by the fact that they require an enrollment fee and offer a cash prize to the winner. For the moment, the player that finishes on the 1st position receives all the money from the enrollment fees of the tournament. From the beginning of the tournament until a player wins 1st place, the prize money is kept in the account of the Ping-Pong Association or in the account of the Tournament itself.

Based on Assignment A1, adapt your application to fit the new requirements.

The regular user can perform the following additional operations:

- Enroll into upcoming Tournaments, by paying the enrollment fee out of their account.
- View Tournaments by category: Finished, Ongoing, Enrolled, Upcoming.
- Search Tournaments by name and by type (free / paid)

The administrator user can perform the following additional operations:

- CRUD paid tournaments.
- Add money to any player's account.
- Withdraw money from any player's account

Every new upcoming tournament must be created with at least one month before its start date and must have an enrollment fee specified at creation time.

1.2 Functional Requirements

For the administrator:

- Login
- View tournaments, matches, players
- Create, update, and delete tournaments
- Delete players
- Add money to and withdraw money from player's account

For the regular user:

- Create an account
- Login
- View tournaments and matches
- Update their own score in a game
- Enroll into Upcoming Tournaments, by paying an enrollment fee from their account
- View tournaments by category
- Search tournaments by name and type

1.3 Non-functional Requirements

The non-functional requirements for this application would be:

- **MVC architectural pattern**, described below
- **JavaFX framework and Scene Builder**, which is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms. [2]
- All the inputs of the application will be validated against invalid data before submitting the data and saving it. (e.g. Tournament start date, Mandatory fields, the player has

enough money in his account, etc.)

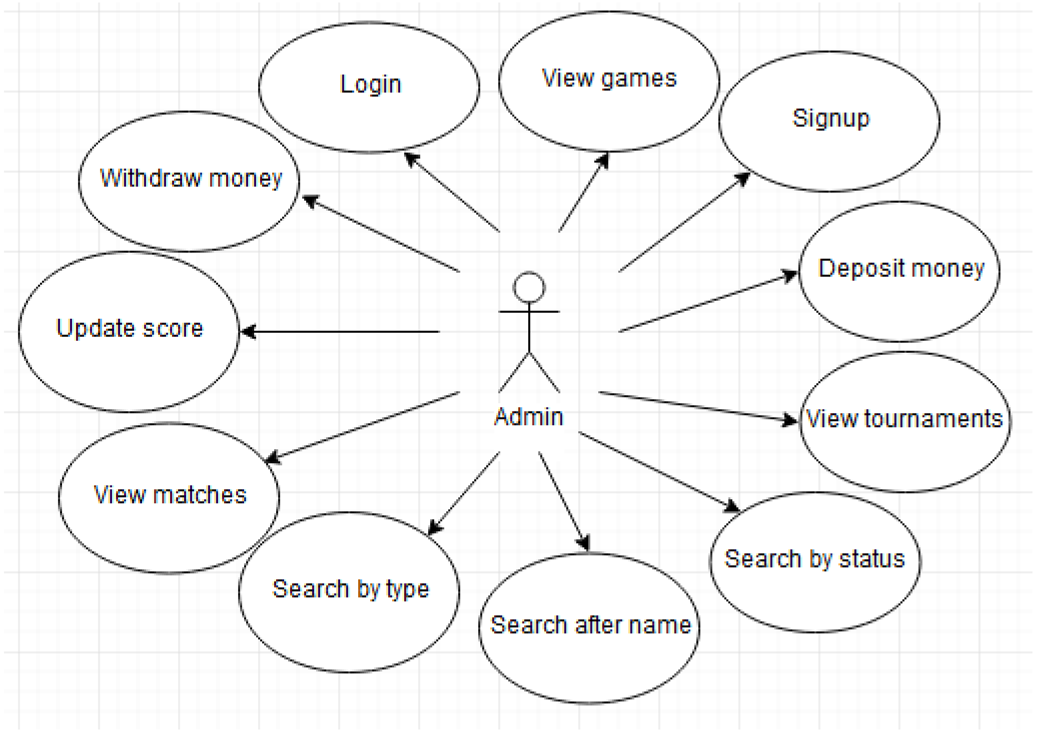
- The **Data Access Layer (DAL)** will be re-implemented using an ORM framework.
- **Abstract Factory,** to switch between the new and the old DAL implementation (implemented with JDBC or equivalent)
- The application will use a **config file** from which the system administrator can set which DAL implementation will be used to read and write data to the database.

2. Use-Case Model

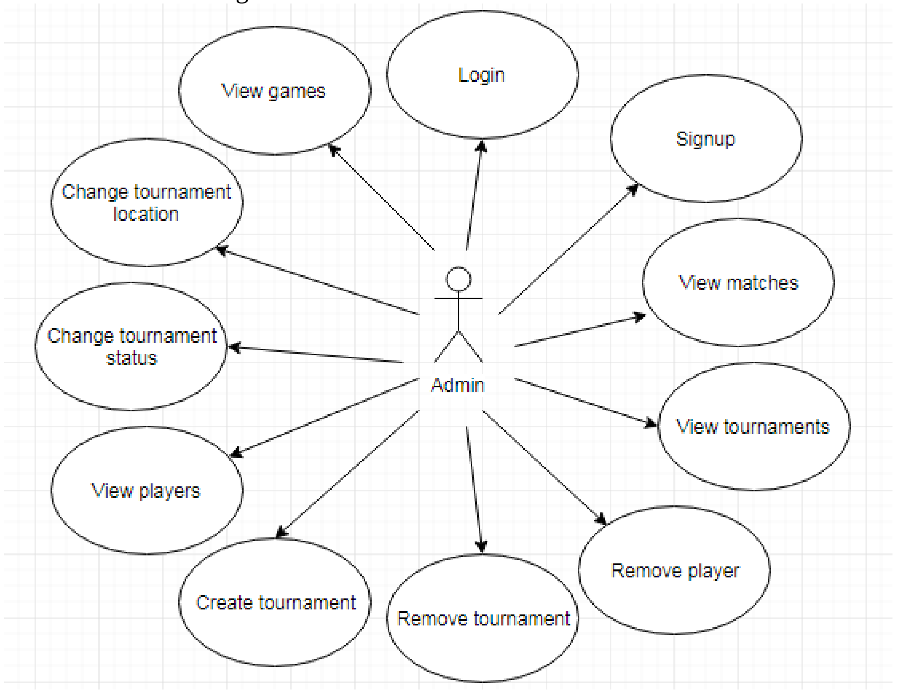
Use-case description:

- Use case: change password (for user)
- Level: user-goal level
- Primary actor: player (regular user)
- Main success scenario: the player logs in successfully, clicks on the “View matches” button to see all the matches, then they must select one of the matches they are currently involved in, click on the “View games” button, and then select on the ongoing game and just press “Update score”, and it will automatically increment their own score by 1.
- Extensions:
    - Player doesn’t enter the right credentials and cannot login
    - Player clicks on “View games” without previously pressing on “View matches”
    - Player attempts to update the score of a finished game
    - Player attempts to view the games of a match they are not involve in

Use-case diagram for player:



Use-case diagram for administrator:



3. System Architectural Design

3.1 Architectural Pattern Description

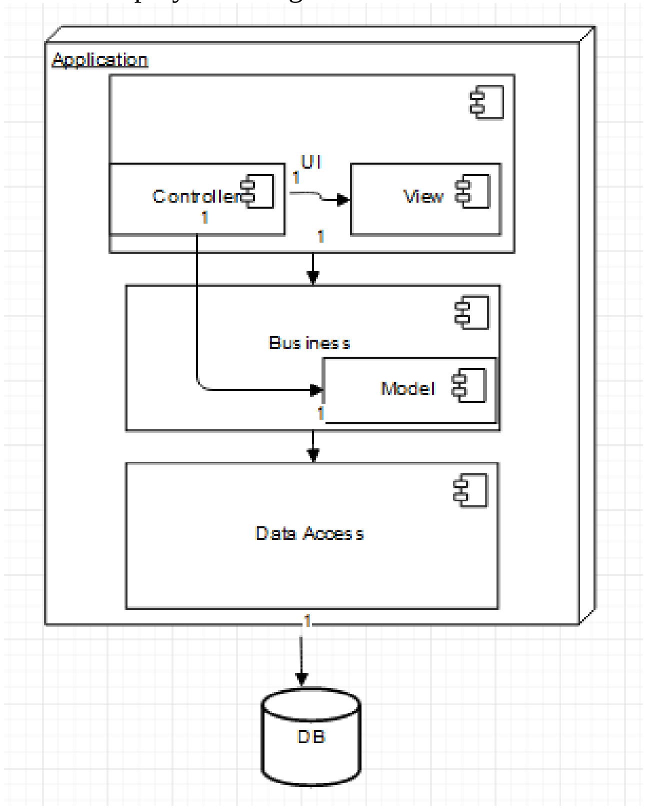
Model–view–controller (MVC) is an architectural pattern commonly used for developing user interfaces that divides an application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development. [1]

This pattern is used to separate application's concerns.

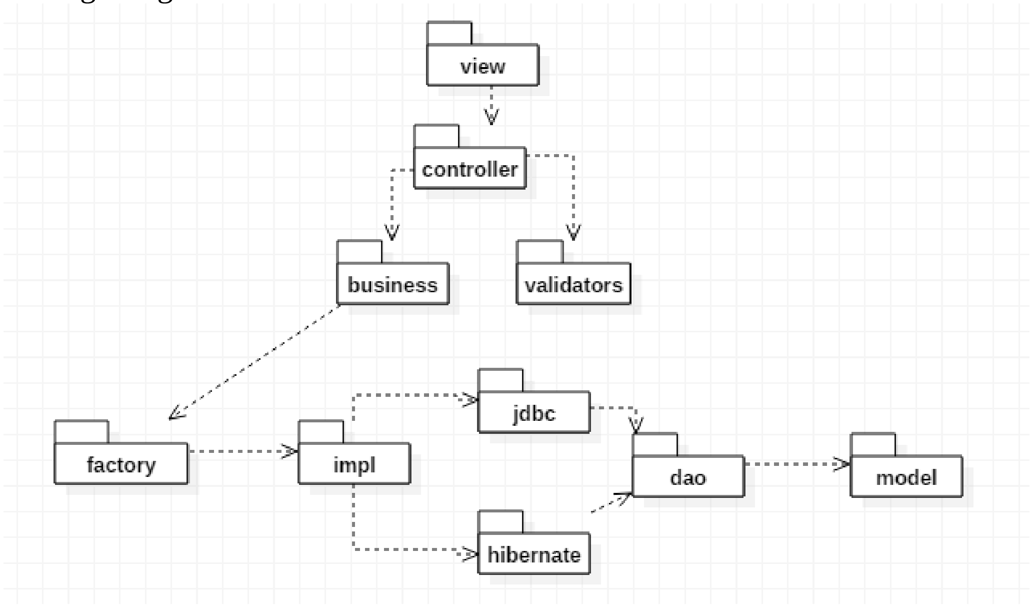
- **Model -** Model represents an object carrying data. It is the central component of the pattern. It expresses the application's behavior in terms of the problem domain, independent of the user interface. It directly manages the data, logic and rules of the application.
- **View -** View represents the visualization of the data that model contains. Multiple views of the same information are possible.
- **Controller -** Accepts input and converts it to commands for the model or view. Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

3.2 Diagrams

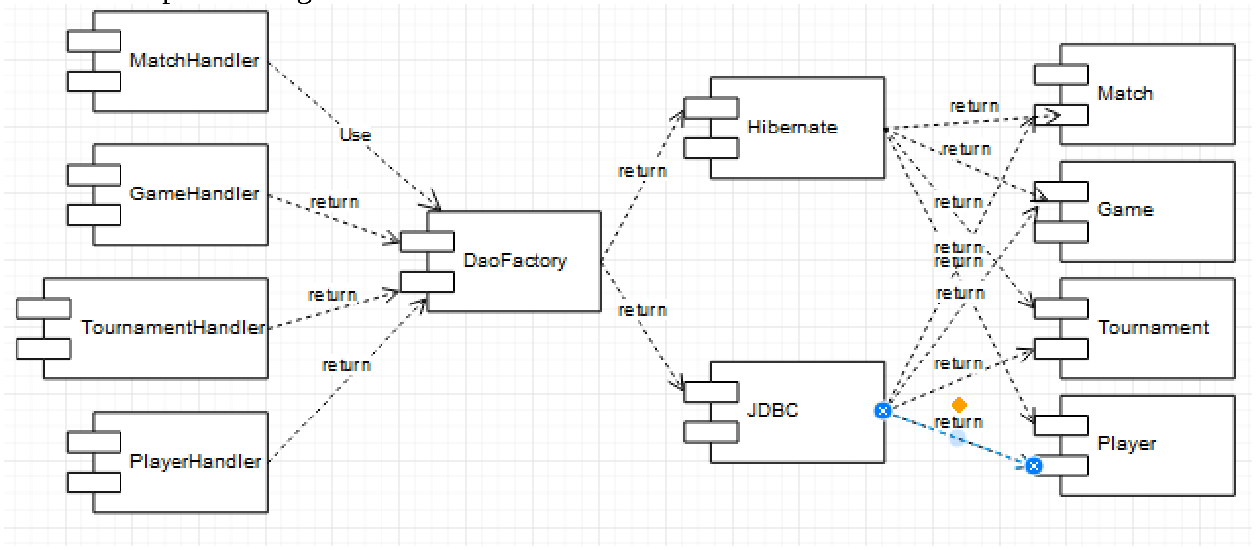
Deployment diagram:



Package diagram:

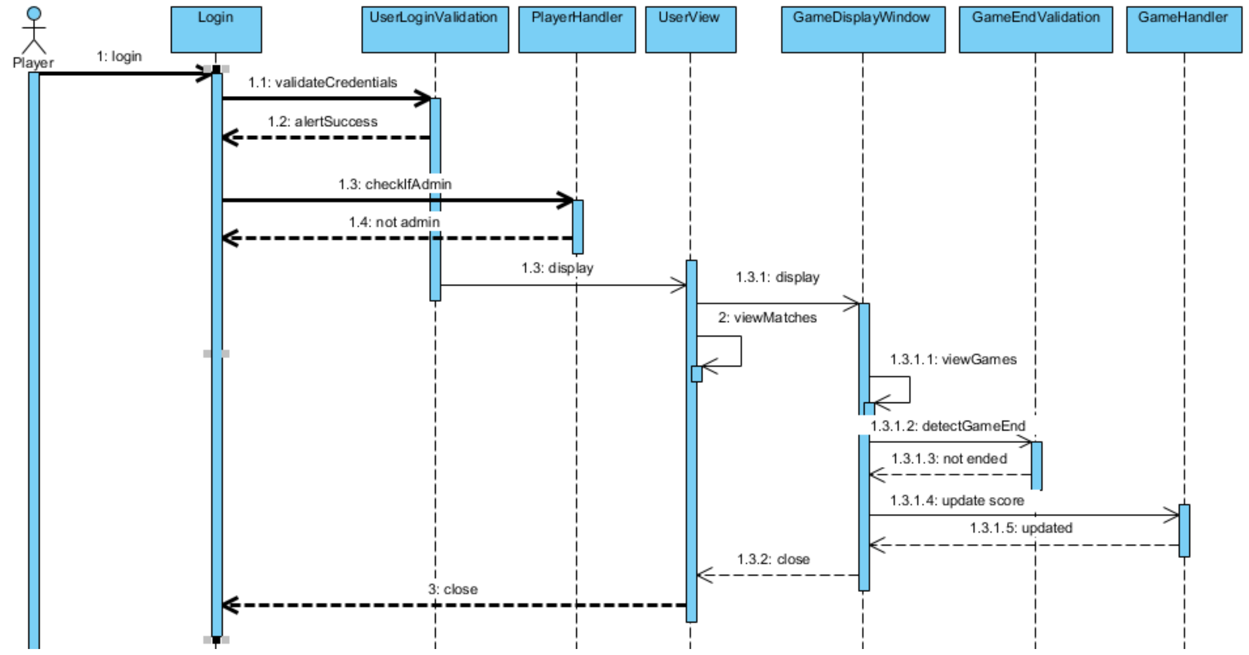


Component diagram:



4. UML Sequence Diagrams

Sequence diagram for updating the score of a game:



5. Class Design

5.1 Design Patterns Description

5.1.1 Data Access Object

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

- **Data Access Object Interface** - This interface defines the standard operations to be performed on a model object(s).
- **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. [3]

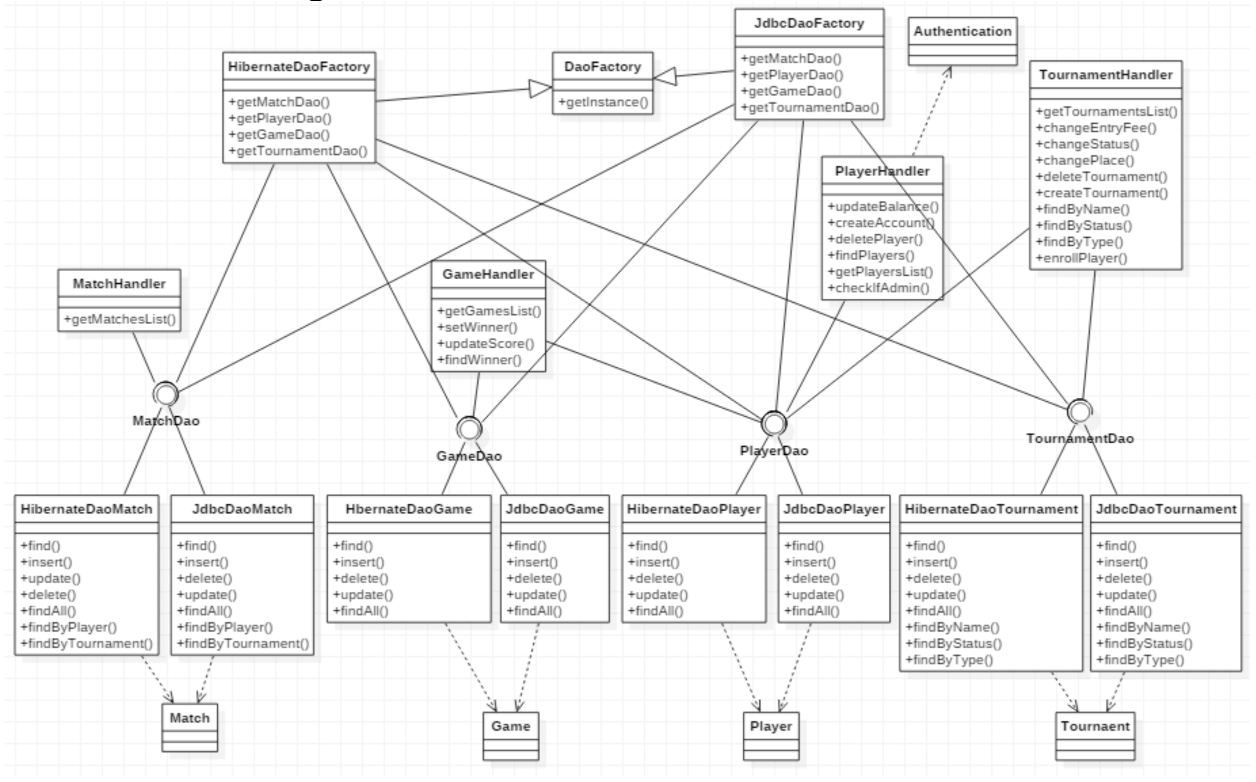
5.1.2 Table Module

A *Table Module* organizes domain logic with one class per table in the database, and a single instance of a class contains the various procedures that will act on the data. The primary distinction with *Domain Model* is that, if you have many orders, a *Domain Model* will have one order object per order while a *Table Module* will have one object to handle all orders.

The strength of *Table Module* is that it allows you to package the data and behavior together and at the same time play to the strengths of a relational database. On the surface *Table Module* looks much like a regular object. The key difference is that *it* has no notion of an identity for the objects it's working with.

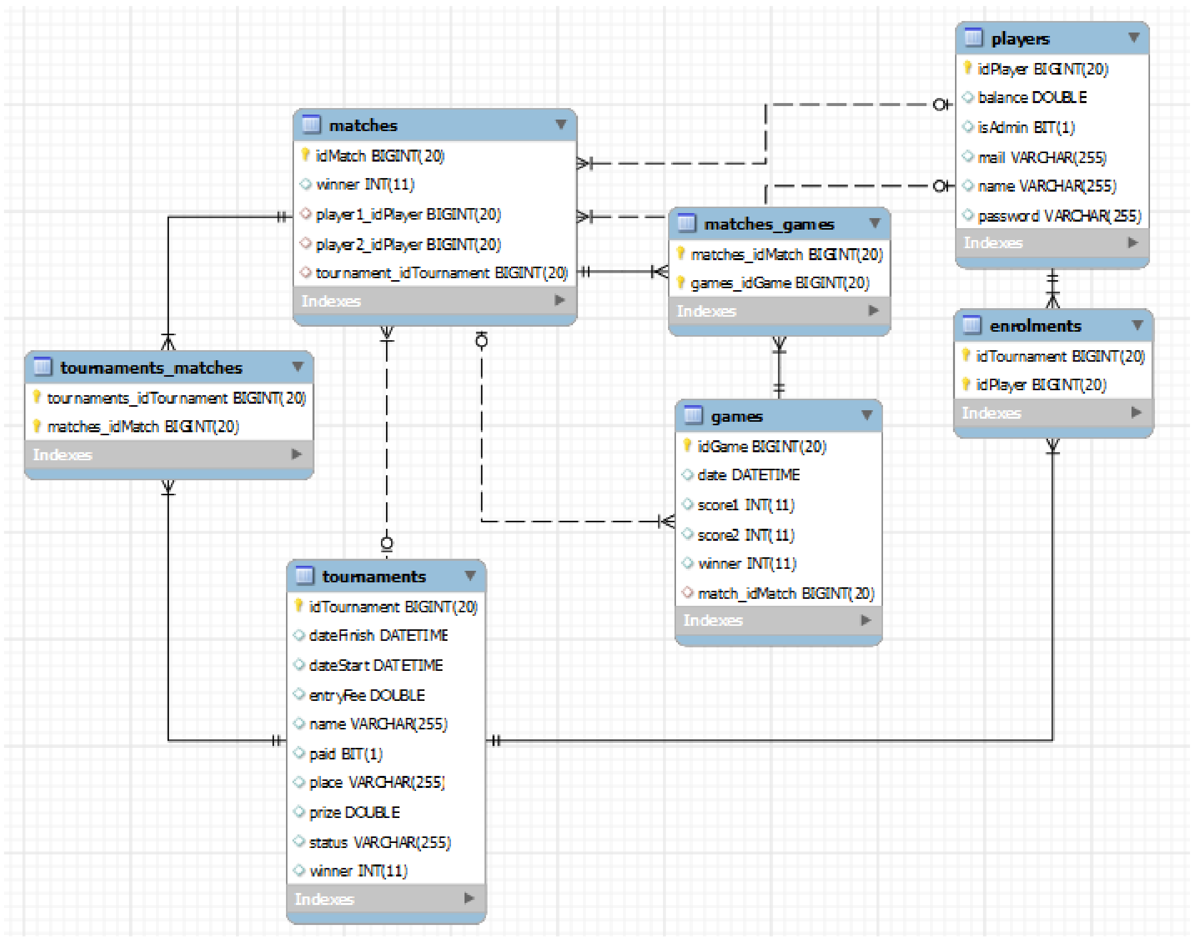
Usually you use *Table Module* with a backing data structure that's table oriented. The tabular data is normally the result of a SQL call and is held in a *Record Set* that mimics a SQL table. The *Table Module* gives you an explicit method-based interface that acts on that data. Grouping the behavior with the table gives you many of the benefits of encapsulation in that the behavior is close to the data it will work on. [4]

5.2 UML Class Diagram



6. Data Model

The database diagram for this project looks like this:



The model consists of four classes, corresponding to the four tables in the figure above: Player, Tournament, Match, and Game, but there are also other 3 tables corresponding to the relationships that establish between the four mentioned before: a many-to-many relationship between players and tournaments, a one-to-many relationship between matches and tournaments, 2 one-to-many relationships between players and matches, and a one-to-many relationship between matches and games.

Each table has a unique identifier – for the Player, the identifier is the player's email address -, while for the other three, it is an integer number. The **Player** only contains information about the user (admin or players), and it is linked to the Match table through two foreign keys, one for each player that takes part in that match. A player can participate in one or more matches.

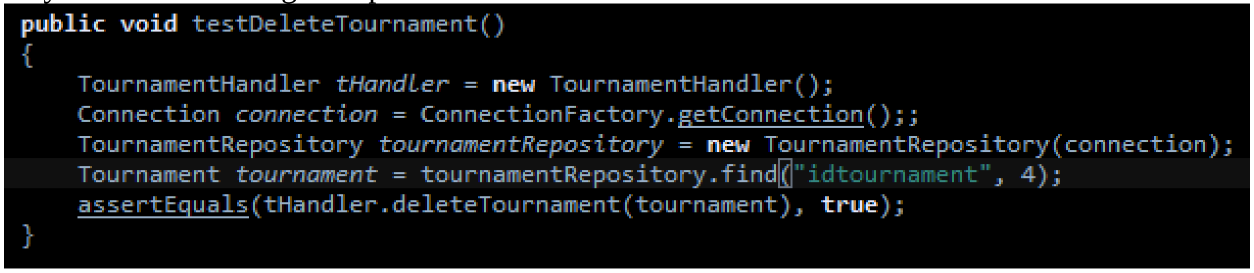
The **Match** table specifies the two players that take part in a certain match and the tournament to which it belongs. Each match consists of 3 up to 5 games, and for each game the winner is the first player to reach 11 points wins that game, however a game must be won by at least a two-point margin.

In the **Game** table, we store the score of each player (the logic for keeping the scores and correctly assigning them to each user is done in the code) and the match it belongs to (it can be easily inferred from there who the players are and in what tournament they are currently playing).

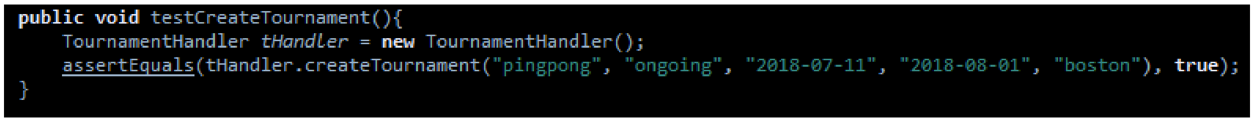
The **Tournament** table only contains data about the tournaments themselves. A tournament consists of a total of exactly 7 games (4 in quarter finals, 2 in semi-finals, 1 final).

7. System Testing

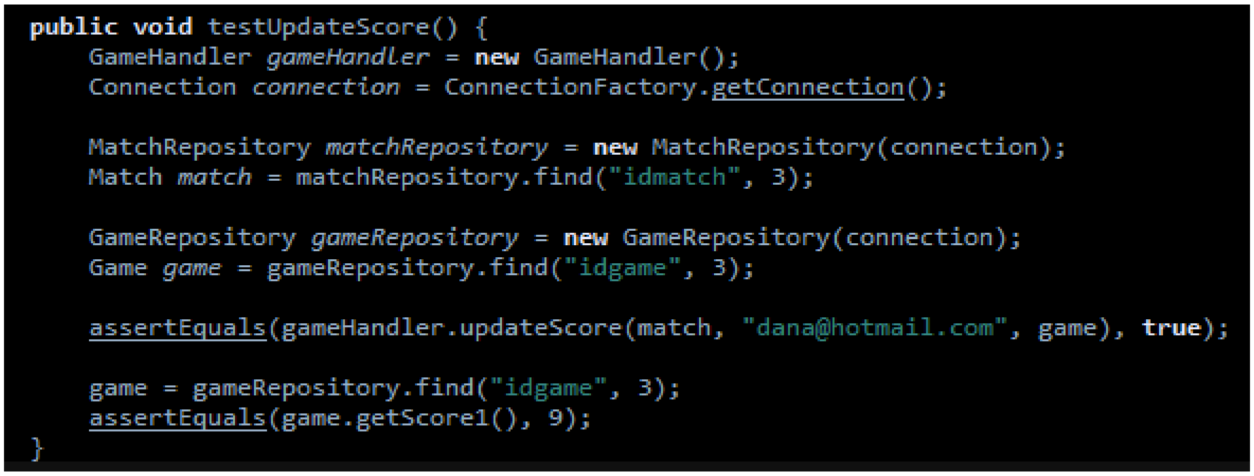
At first, testing the application was done through System.out.println(), and by checking the rows in the database to see if they had the expected values. Later on, I used Junit tests, which involved testing parts (units) of the code in the Business and Data Access layers to see whether they worked according to expectations or not.



```java
public void testDeleteTournament()
{
    TournamentHandler tHandler = new TournamentHandler();
    Connection connection = ConnectionFactory.getConnection();;
    TournamentRepository tournamentRepository = new TournamentRepository(connection);
    Tournament tournament = tournamentRepository.find("idtournament", 4);
    assertEquals(tHandler.deleteTournament(tournament), true);
}
```



```java
public void testCreateTournament(){
    TournamentHandler tHandler = new TournamentHandler();
    assertEquals(tHandler.createTournament("pingpong", "ongoing", "2018-07-11", "2018-08-01", "boston"), true);
}
```



```java
public void testUpdateScore() {
    GameHandler gameHandler = new GameHandler();
    Connection connection = ConnectionFactory.getConnection();

    MatchRepository matchRepository = new MatchRepository(connection);
    Match match = matchRepository.find("idmatch", 3);

    GameRepository gameRepository = new GameRepository(connection);
    Game game = gameRepository.find("idgame", 3);

    assertEquals(gameHandler.updateScore(match, "dana@hotmail.com", game), true);

    game = gameRepository.find("idgame", 3);
    assertEquals(game.getScore1(), 9);
}
```

8. Bibliography

[1] “Layered architecture pattern”, by Suhanov Begench: https://www.slideshare.net/suhanovbegench/layered-architecture-style
[2] “What is JavaFX?”, from Oracle: https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm
[3] “Data Access Object Pattern”, from Tutorials Point: https://www.tutorialspoint.com/design_pattern/data_access_object_pattern.htm
[4] “Framework Design Guideline: Domain Logic Pattern”, by Martin Fowler: http://www.informit.com/articles/article.aspx?p=1398617&seqNum=3
[5] Useful tutorials: https://github.com/buzea/SoftwareDesign2018