Ping-pong Tournament Manager
Analysis and Design Document

Student: Han Ana-Maria
Group: 30433

Table of Contents

1. Requirements Analysis

1.1 Assignment Specification

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.

1.2 Functional Requirements

For the administrator:

- Login
- View tournaments, matches, players
- Create, update, and delete tournaments
- Delete players

For the regular user:

- Create an account
- Login
- View tournaments and matches
- Update their own score in a game
- Change their address and password

1.3 Non-functional Requirements

The non-functional requirements for this application would be:

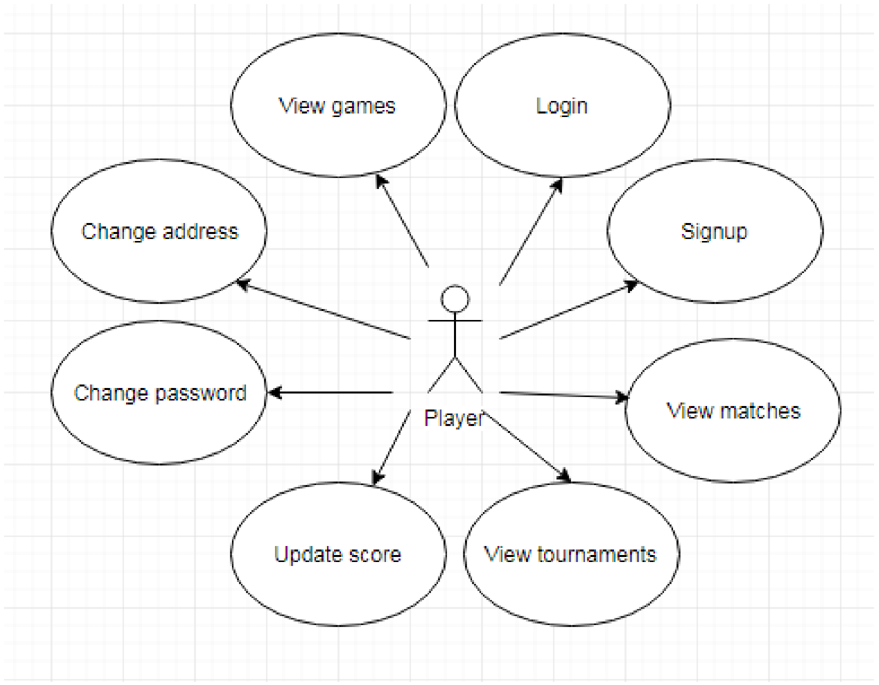
- **3-tier architectural pattern**, described below
- **JavaFX framework**, which is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms. [2]
- **Table Module,** for the Domain Logic design pattern, which handles the business logic for all the rows in a database table or view.
- **Data Access Object,** for the Database Logic design pattern, which is used to separate low level data accessing API or operations from high level business services.
- **JDBC,** a java API to connect and execute query with the database. JDBC API uses JDBC drivers to connect with the database.

2. Use-Case Model

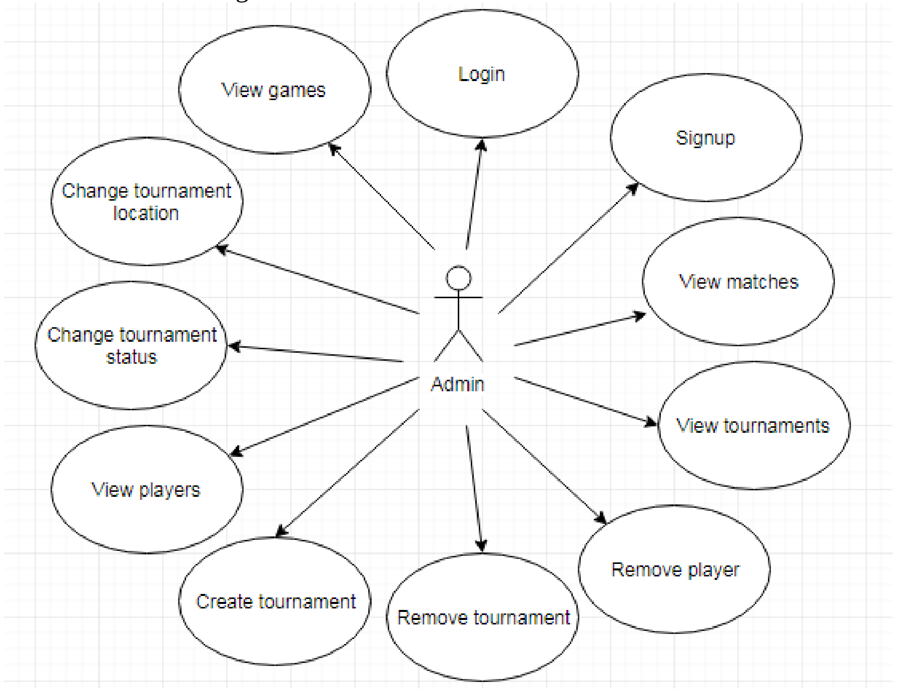
Use-case description:

- 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

The 3-tier architectural pattern organizes a system into a set of layers, each of which provide a set of services to the layer above. Layering consists of a hierarchy of layers, each providing service to the layer above it and serving as client to the layer below. Each layer should be cohesive, loosely coupled to the layers underneath, and at roughly the same level of abstraction.

The context for using this pattern: when working with a large, complex system, and we want to manage complexity by decomposition.

The advantages of using the 3-tier pattern are:

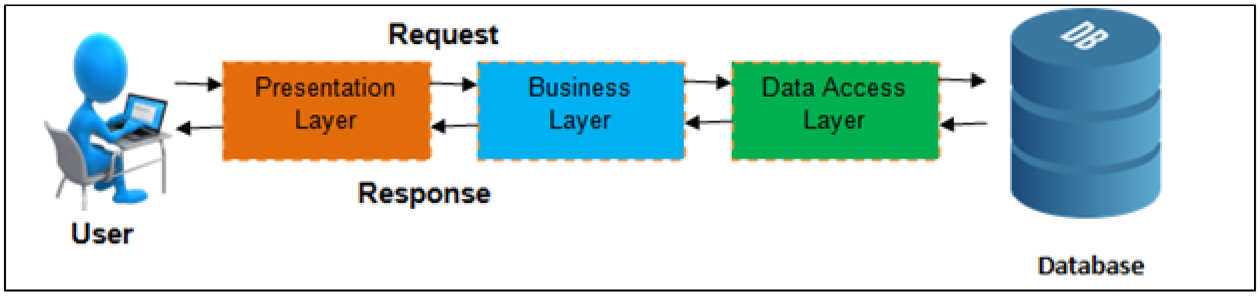
- Different components of the application can be independently deployed, maintained, and updated, on different time schedules, making it possible for team members to work in parallel on different parts of the application with minimal dependencies and to test the components independently of each other.
- Each layer may hide private information from the other layers.
- Because of its cohesiveness and low coupling to the lower layers, each layer makes it easier to be reused by others and to be replaced or interchanged.
- Increased flexibility, maintainability, and scalability, because we separate the user interface from the business logic, and the business logic from the data access layer.

The disadvantages are:

- Extra overhead of passing through layers, and changes will pass slowly as well to higher layers.
- Sometimes, it is difficult to cleanly assign a functionality to the appropriate layer.
- Cannot be used for simple applications because it adds complexity. [1]

3.2 Diagrams

The system's conceptual architecture looks as follows:

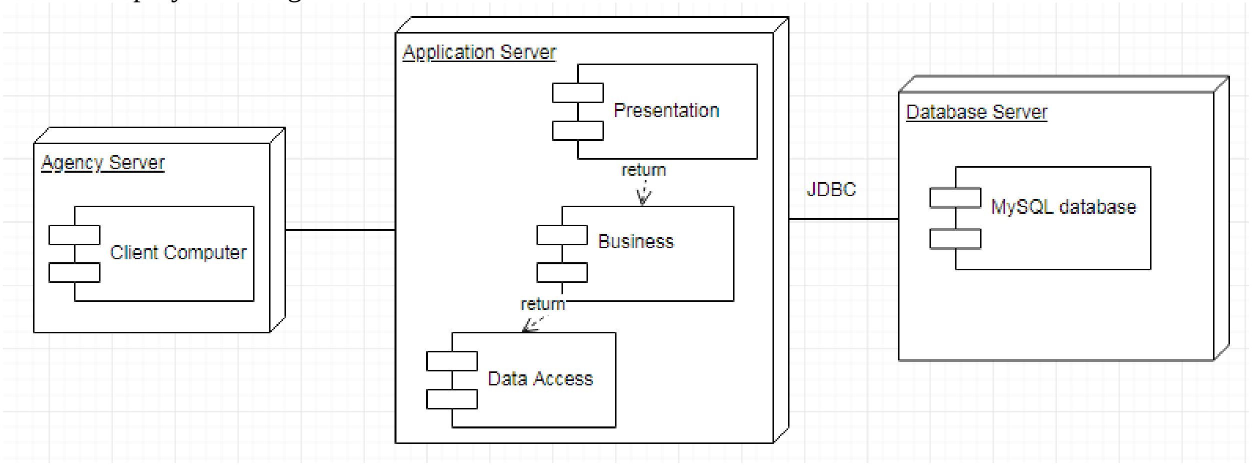


For the **Data Access Layer**, the connection to the database (kept in a local MySQL server) is done through a JDBC connector. This layer consists in three packages: connection (contains the ConnectionFactory class where the actual connection happens), model (contains four classes with getters and setters only, each one corresponding to a table in the database) and repository (where the methods for select, insert, update and delete queries are implemented separately for each table, and for their specific interfaces).

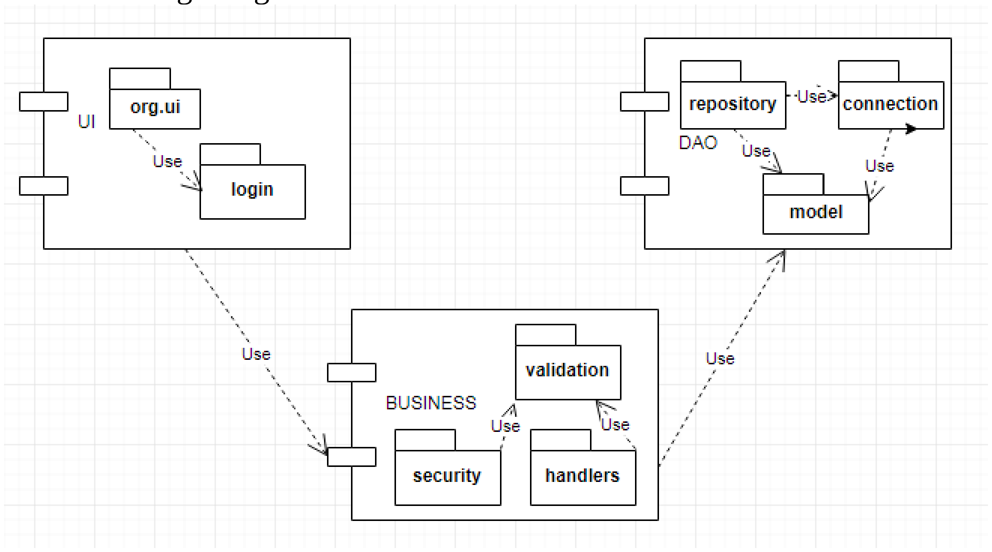
The **Business Layer**, the liaison between the database and the user interface, contains validators (for example: checking if the mail address has the right pattern, if the credentials are correct, if a game or match has ended), an Authentication class where the password is checked for strength (security package), and handlers for each table, where the methods for dealing with the user's requests are implemented in order to access and modify data from the database.

The **Presentation Layer** offers the possibility to log in or sign up, to view matches and tournaments, and other functionalities that are specific to each type of user. Depending on the credentials entered, either a player window or admin window will be displayed. A player can view all the tournaments and matches, change their password or address, and update their own score in a game they're playing. The admin can view the tournaments, matches, and players, delete tournaments and players, create tournaments, and change a tournament's status or location. All these actions have to pass first through the business layer, even for something that doesn't require access to the database, such as verifying a password's strength.

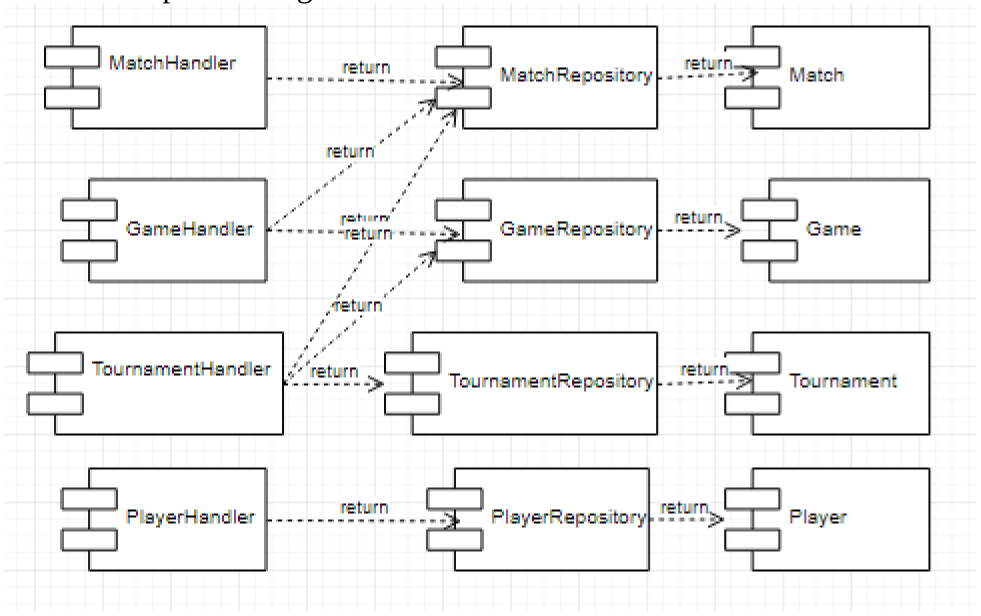
Deployment diagram:



Package diagram:

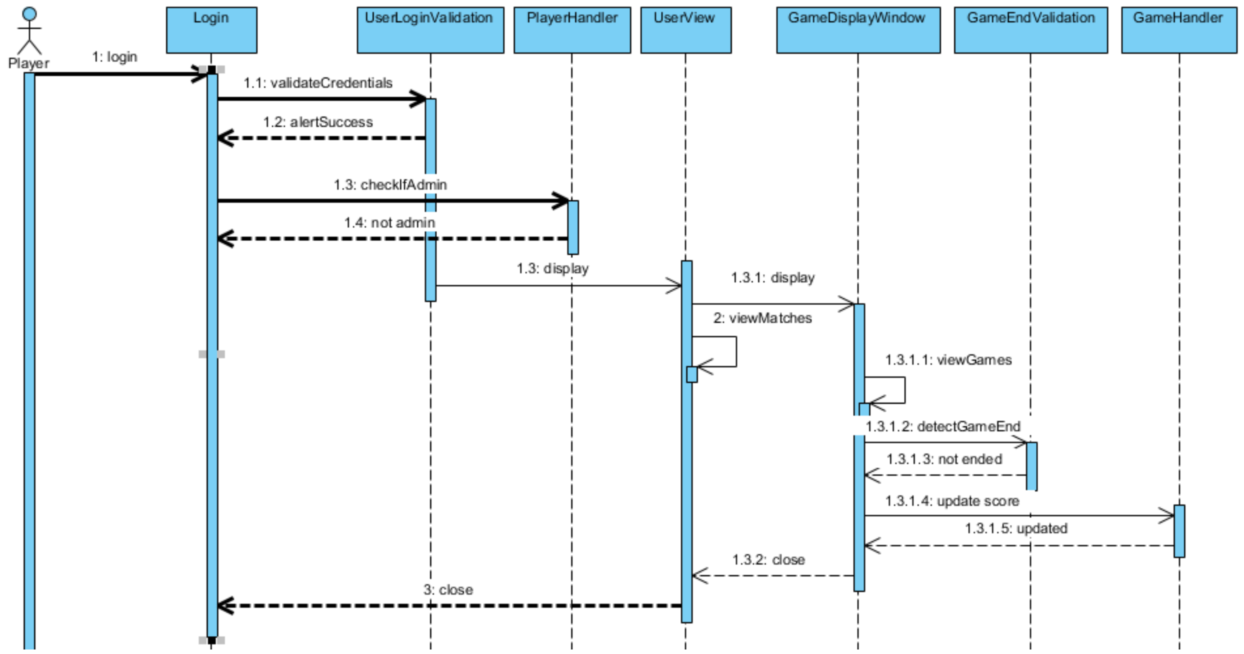


Component diagram:



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]

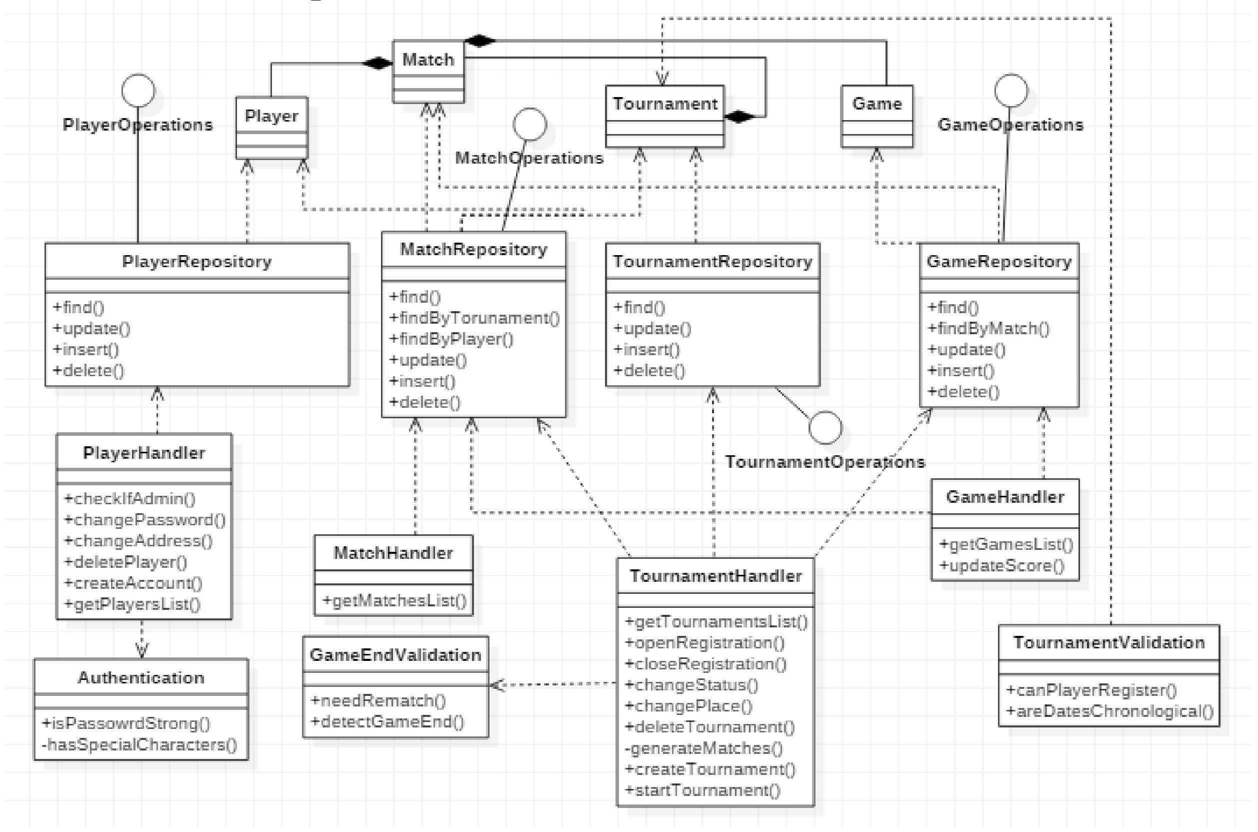
5.1.2 Table Module

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.

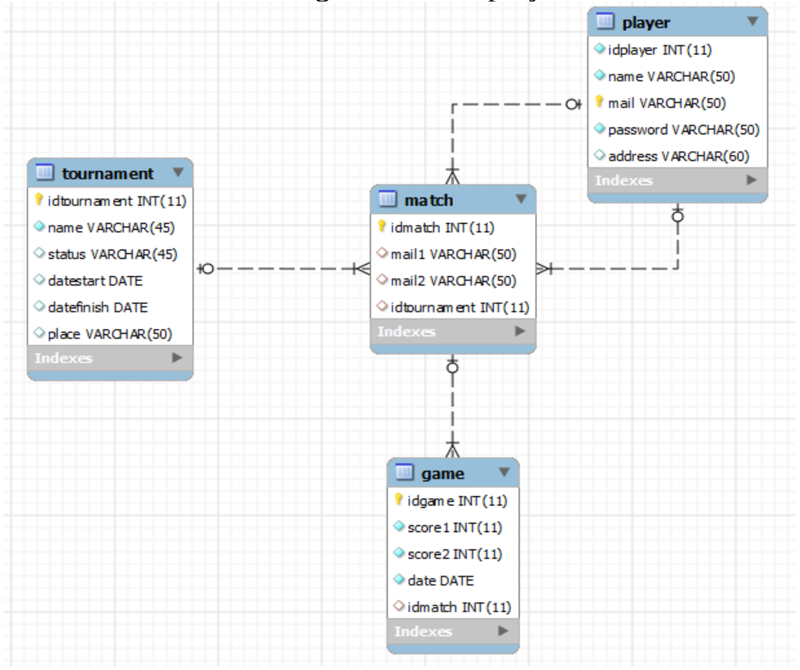
Usually you use *Table Module* with a backing data structure that's table oriented. The tabular data is normally the result of a SQL call and is held in a *Record Set* that mimics a SQL table. The *Table Module* gives you an explicit method-based interface that acts on that data. Grouping the behavior 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.

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.

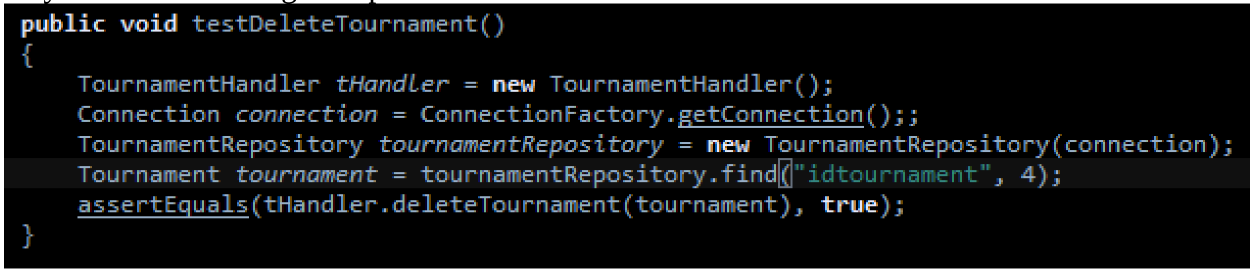
The **Match** table specifies the two players that take part in a certain match and the tournament to which it belongs. Each match consists of 3 up to 5 games, and for each game the winner is the first player to reach 11 points wins that game, however a game must be won by at least a two-point margin.

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).

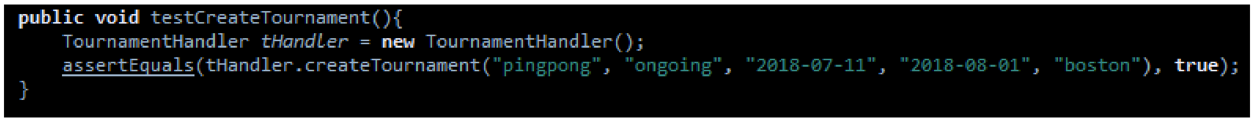
The **Tournament** table only contains data about the tournaments themselves. A tournament consists of a total of exactly 7 games (4 in quarter finals, 2 in semi-finals, 1 final).

7. System Testing

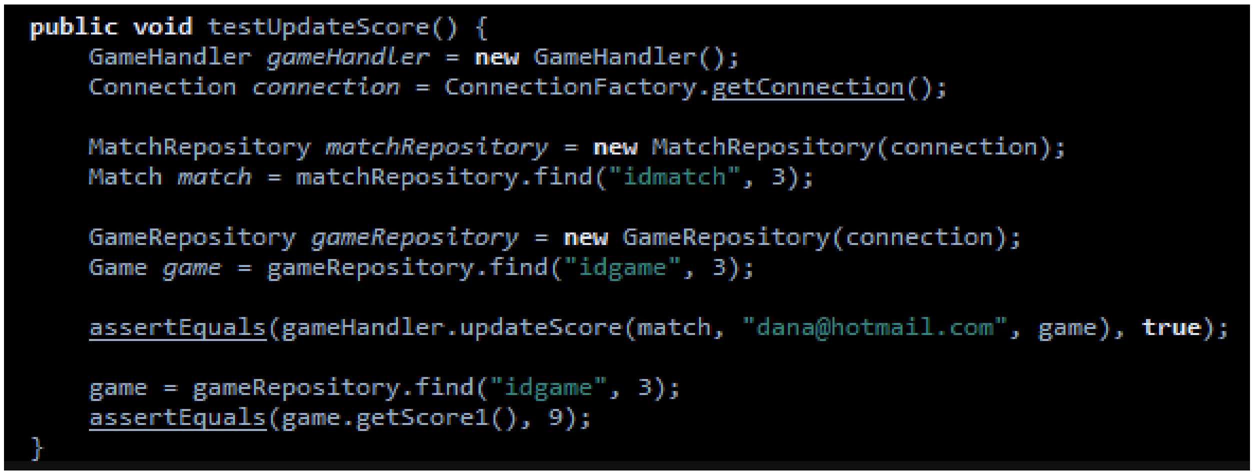
At first, testing the application was done through System.out.println(), and by checking the rows in the database to see if they had the expected values. Later on, I used Junit tests, which involved testing parts (units) of the code in the Business and Data Access layers to see whether they worked according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



```
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);
}
```



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



```
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