ASSIGNMENT A2

Analysis and Design Document

Student: Moldovan Rares-Sebastian

**Group: 30433**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 3

3. System Architectural Design 3

4. UML Sequence Diagrams 3

5. Class Design 3

6. Data Model 3

7. System Testing 3

8. Bibliography 3

1. Requirements Analysis

# Assignment Specification

# Assignment Specification

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 users need to provide personal information in order to log in.

# Functional Requirements

The regular user can perform the following operations:

* Log in with an email and a password.
* View the list of tournaments.
* View match information.
* Update the score of the current game (if and only if they are one of the two players in the game).
* Enroll into upcoming tournaments by paying the enrollment fee.
* View tournaments by category.
* Search tournaments by name and type (free or paid).

The administrator user can perform the following operations:

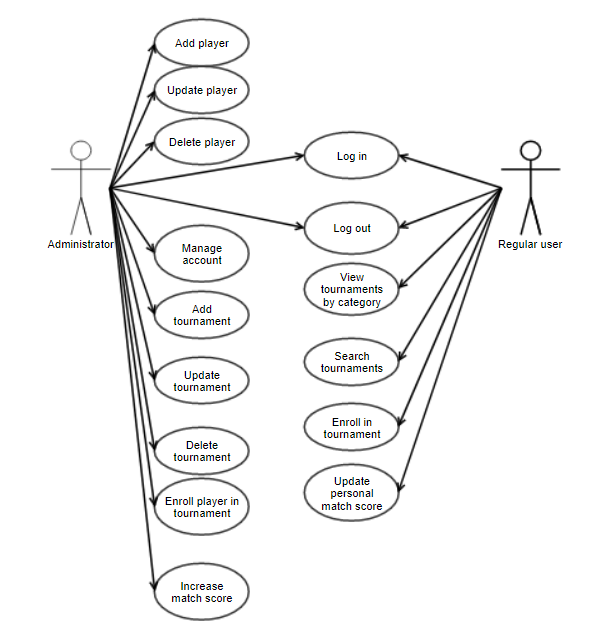
* Log in with an email and a password.
* Operations on players (users): view all players, create new player, update player, delete player, update player accounts.
* Operations on tournaments: view all tournaments, create new tournament, update tournament and delete tournament (both free and paid).
* Enroll players in tournaments manually.

# Non-functional Requirements

The non-functional requirements refer to application design and implementation constraints. The following are the requirements imposed by the assignments specification:

* The data must be stored in a database, and the information must be retrieved from it and written back at every operation.
* The application needs to be organized using the layered architectural pattern.
* The application must use a data source hybrid pattern (table module) or a domain logic pattern and a data source pure pattern (table data gateway, data access object).
* The application must be implemented using Java/C# API.
* The application must use Java FX for user interface implementation, in the case of a Java approach.

2. Use-Case Model



**Use case:** Enroll in tournament

**Level:** user-goal level

**Primary actors:** Regular user

**Main success scenario:**

* The user successfully launches the application.
* The user successfully logs in.
* The user selects a tournament form the tournament list.
* The user selects the “Enroll in tournament” command.
* The system checks for the possibility of the enrollment.
* The system successfully enrolls the player in the tournament.

**Extensions:**

* If the user does not select a tournament, the system displays and error message.
* If the user does not have enough money in his account for enrolling into the tournament, an error message is displayed.
* If the tournament cannot except any other players, the system displays an error message.

3. System Architectural Design

**3.1 Architectural Pattern Description**

The application relies on the **MVC (Model View Controller)** architectural pattern. This architecture was used because of the non-functional requirement of the assignment that stated the use of this pattern.

The main idea is to separate the functionality into three different components: model, view and controller.

1. **Model**

The model represents the business logic of the application. It contains all data, therefore it maintains the internal state of the application. It communicates with a database through a data access layer (DAL).

The model doesn’t know anything about the view, since the view has the functionality for displaying the data in the model, not alter it.

1. **View**

The view represents the graphical representation of the data stored in the model. It only cares about the presentation of the data (tables, lists, text fields, labels etc.). The view is an observer of the model and updates its graphical components with the current data in the model.

1. **Controller**

The controller binds the model to its views and takes care of the actions performed on the view (binds the buttons and all the other components that generate actions with backend functionality stored in the business model). It is aware of both the model and the view and it is the main bridge of communication between them.

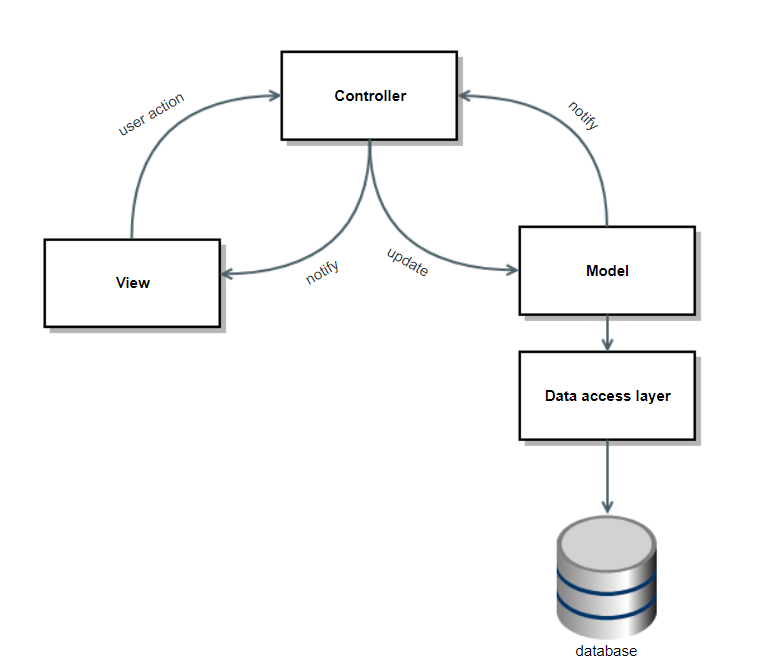
The **data access layer** provides the functionality for the interaction with the database, together with the main entities implied in the application such as player, tournament, match and set. This layer sets up the connection with the database and constructs the SQL statements for retrieving and updating data from/to the database.

In the case of this application, the access layer provides a factory for getting the data access objects. The DAOs are implemented via two different approaches: a classical approach with a JDBC connector and an approach that uses the Hibernate framework.

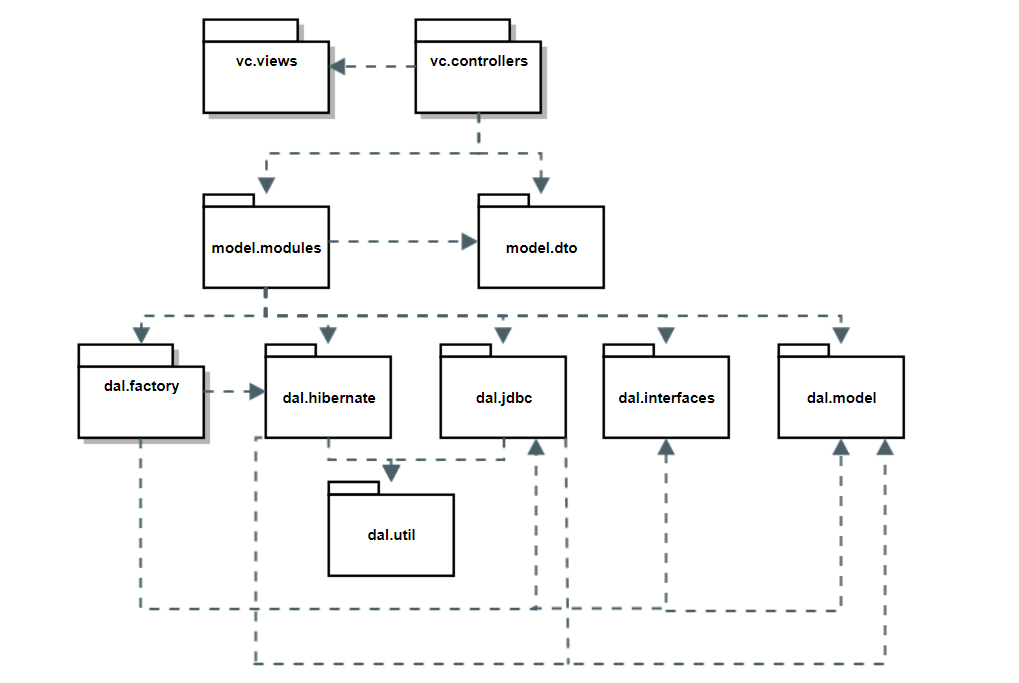
DAL has no dependency on other application layers.

**3.2 Diagrams**

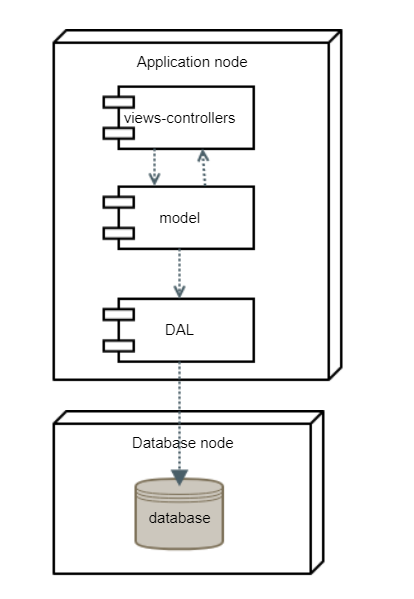
**3.2.1 Conceptual architecture**



**3.2.2. Package diagram**

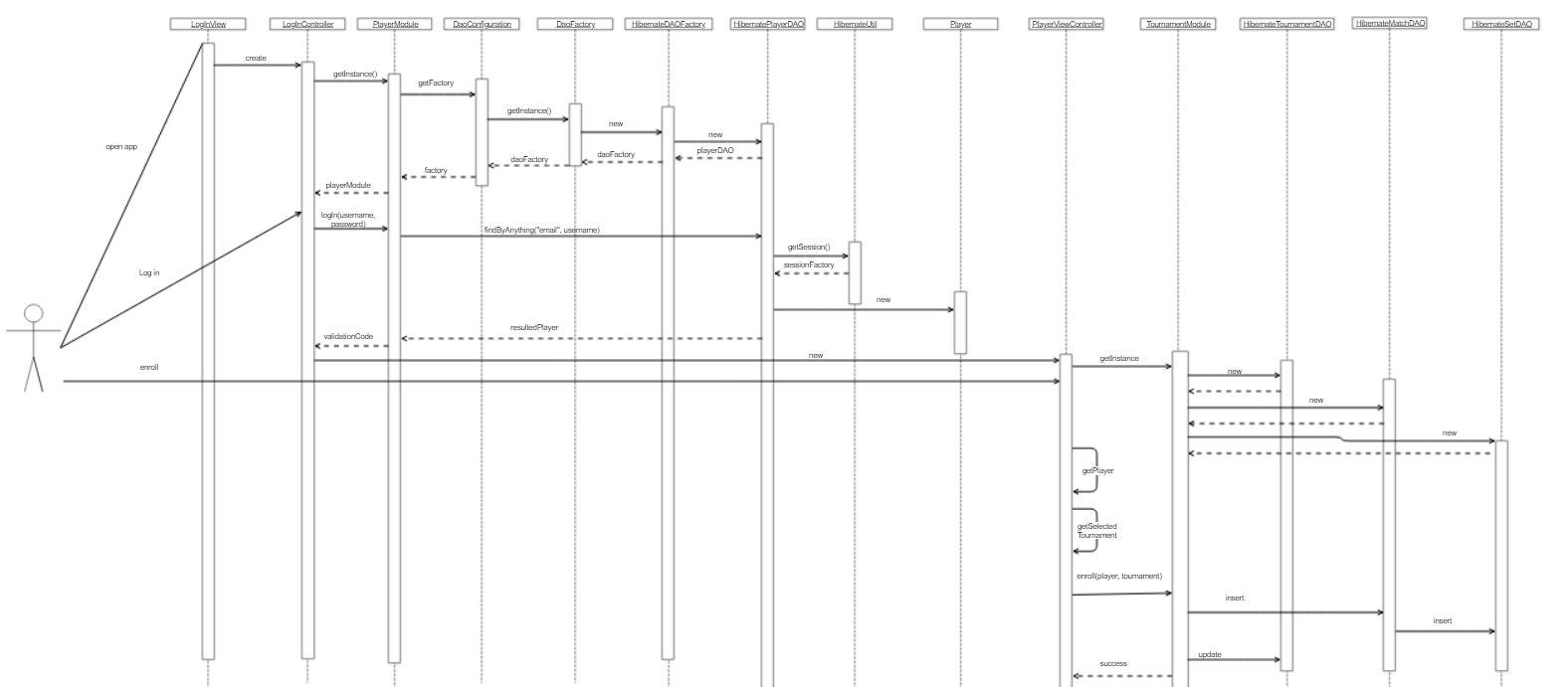


**3.2.3. Deployment diagram**



1. UML Sequence Diagrams

Sequence diagram for ‘Enroll player’ successfully, new player added in a new match



5. Class Design

**5.1 Design Patterns Description**

The design patterns that I used for the application were imposed by the non-functional requirements and by the MVC architecture.

**5.1.1. Abstract factory**

The abstract factory design pattern allows a class to get an instance of another class (interface) without knowing which subclass it selects. In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

I used the Abstract factory DP for switching between the JDBC and Hibernate implementations of the DAO design pattern by providing classes: DaoFactory, HibernateDaoFactory and JDBCDaoFactory. The static method getInstance of the abstract class select between the DAO implementations.

**5.1.2 DAO design pattern**

The DAO (Data access object) design pattern focuses on communicating with the database. Each entity in the domain model has associated a data access object which deals with the basic operations on that table only. The basic operation performed in a DAO class are insert, find, update and delete.

Moreover, the layer that is communicating with the database through these objects has access to an interface DAO Interface that provides only the functionality needed for the business and any other methods that may be public in the actual DAO object, but used only in the dao layer, are hidden from the business layer.

**5.1.3. DTO design pattern**

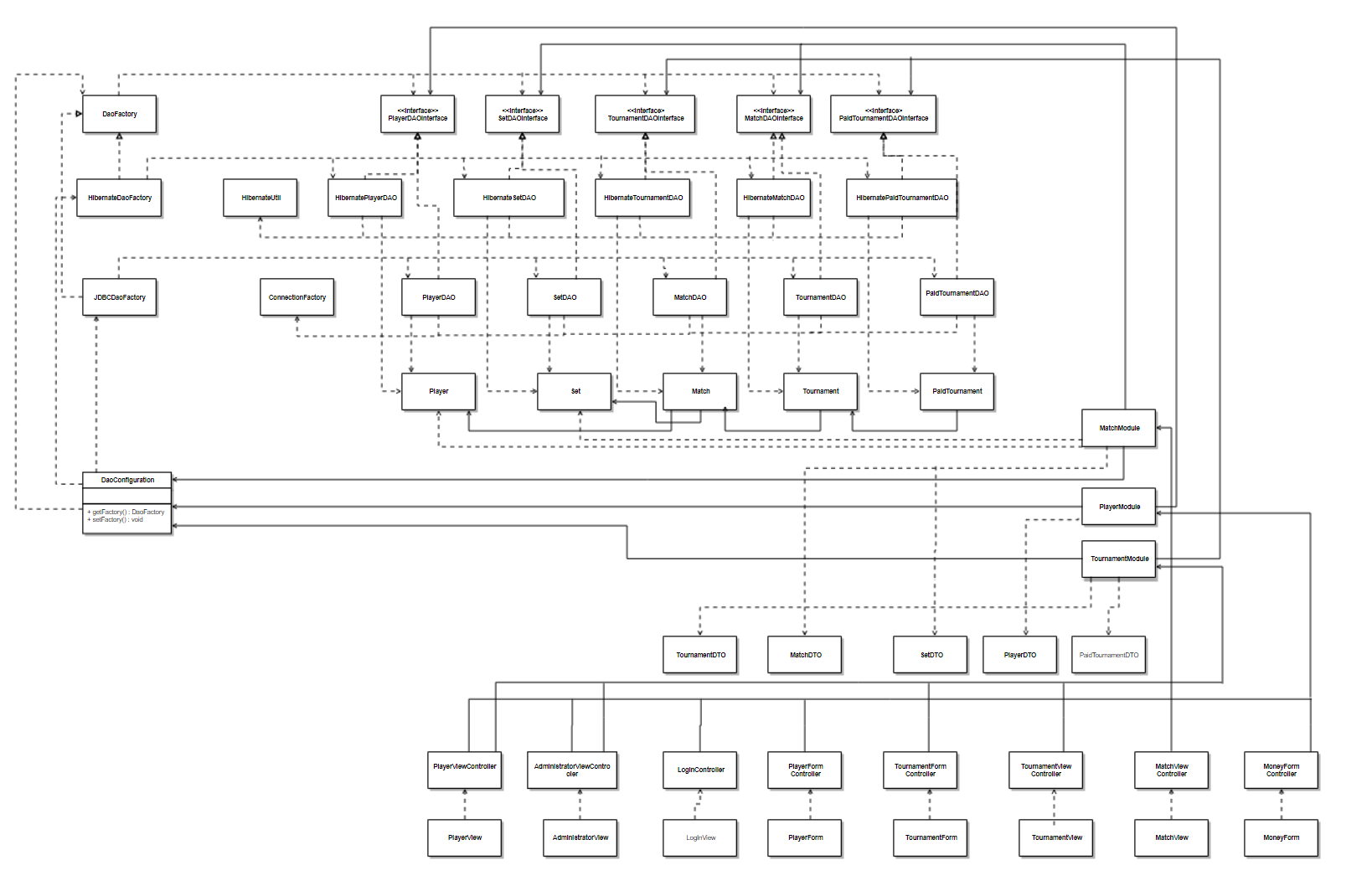
The DTO design pattern (Data transfer object) is a pattern that allows communication between more components because it groups information in objects. The presentation layer does not need to know anything about the data model (entities), but it often requires complex information that needs to be packed and provided. This pattern allows the information to be packed and sent between presentation and business without the need for the presentation to understand what it is receiving.

**5.1.4. Observable design pattern**

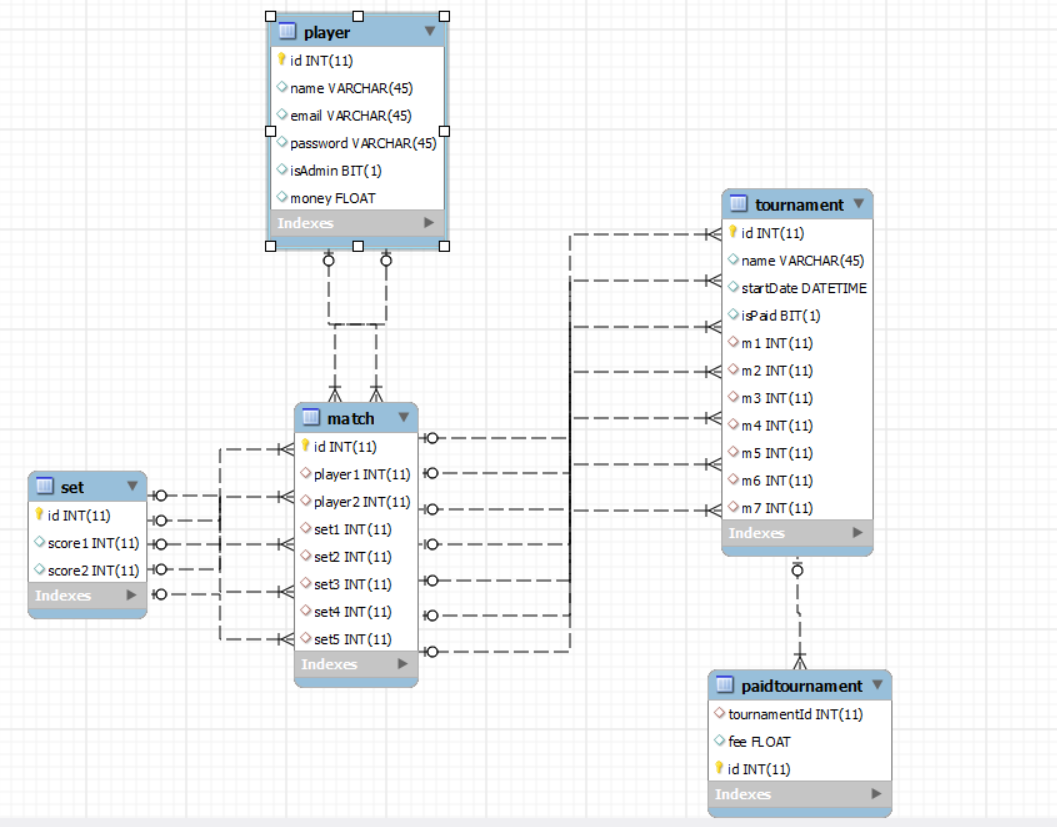
The observable design pattern was induced by the MVC architecture. The view is an observer of the model, the model being observable. This design pattern allows notification of one object that another has changed. Any observer can add observable objects to an internal list.

Whenever the observable object is changed, the observer will perform an action of update. In Java, this is done via interface java.util.Observer (with method update) and the superclass java.util.Observable (with methods notifyObservers and setChanged).

**5.2 UML Class Diagram**



6. Data Model



Data is stored on a local database called “tabletennis”. This database organizes the information into four main tables that represent also the entities used in the application.

These four tables are:

6.1. **player**(*id: int, name: varchar, email: varchar, password: varcha*r, *is\_admin: bit*)

The player table defines the basic information about the player: the id, the name and the email and password are hold for authentication purposes. The is\_admin field stores information about the state of the user. If it is 1 (true), then the user is an administrator. Otherwise, the user is a regular one. This information is very important because the use cases of the two differ significantly.

! Whenever a player is deleted, corresponding matches are filled with null values.

6.2. **tournamen**t(*id: int, name: varchar, startDate: date, is\_paid: bit, m1: int, m2: int, m3: int, m4: int, m5: int, m6: int, m7: int*)

This table defines the basic information about a tournament. The id, name and date fields are self-explanatory. The is\_paid fields selects whether or not the tournament has a fee and needs to be stored in the paidtournament table.

The others represent foreign keys in the match table and they are allowed to be null. If a column is null, that means the match is not ready, but the tournament can obviously exist. Moreover, only the first four matches can be added from the very beginning because the last three are derived from them.

6.3. **paidtournamen**t(*id: int, fee: float, tournamentId: int*)

This table associates a fee to all the tournaments which are paid. It has an id field, the fee field which represents the money to be paid for entering the tournament and a tornamentId field which is a foreign key and is associated with the tournament table’s primary id field.

6.4. **match**(*id: int, player1: int, player2: int, set1 : int, set2 : int, set3 : int, set4 : int, set5: int)*

This table defines the information about a match. The id is necessary for selecting purpose. All the other fields, except stage, are foreign keys in the following sense:

* Player1: id of the first player participating in the match.
* Payer2: id of the second player participating in the match.
* Set1: id of the first set.
* Set2: id of the second set
* Set3: id of the third set
* Set4: id of the fourth set
* Set5: id of the fifth set.

6.5. **set**(*id: int, score\_p1: int, score\_p2: int*)

This table defines the information about a set. The id is necessary for the selecting purpose. The score\_p1 field indicates the score of the first player and the score\_p2 field indicates the score of the second player.

7. System Testing

The main testing strategy was unit testing. The focus was on the business module because it is the layer that contains the most difficult operations (such as checking for end of matches and games, automatically introducing matches after quarterfinal/semifinal matches are won etc).

Unit testing was performed using Junit and focused on both computations and database operations. The following functionalities are covered by the Junit tests:

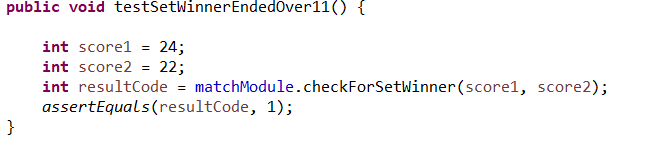
* Test for end of set detection.
* Test for end of match detection.
* Test for money deposit.
* Test for money withdraw.

The testing strategy used equivalence partitioning for checking the end of set and end of match detection. For the test for end of set, the following are the partitions:

1. score1 <11 and score2 <11
2. score1 >11 and score2<10
3. score1<10 and score2>11
4. score1>10 and score2>10 with difference of one point.
5. score1>10 and score2>10 with difference of two points.

The testing strategy for checking the end of match took the same form, but since the points of one player relative to a match belong to the discrete set {1,2,3}, there were maximum 3\*3=9 testing possibilities. A test for score1 = 3, one for score2=3 and for both different from 3 were performed.

An example of the Junit test method for set winner detection is provided in the following picture:



8. Bibliography

1. Resources

<https://github.com/buzea/SoftwareDesign2018>

1. MVC & Controller of JavaFX

<https://ro.wikipedia.org/wiki/Model-view-controller>

<https://docs.oracle.com/javase/8/javafx/api/javafx/fxml/FXMLLoader.html>

1. JavaFx

<https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm>

<https://www.youtube.com/playlist?list=PL6gx4Cwl9DGBzfXLWLSYVy8EbTdpGbUIG>

1. SQL operations

<https://www.w3schools.com/sql/>

1. Junit

<https://junit.org/junit4/>

1. Hibernate

<https://www.journaldev.com/3793/hibernate-tutorial>

<https://www.tutorialspoint.com/hibernate/index.htm>

1. Support in commonly-asked programming problems

<https://stackoverflow.com/>