ASSIGNMENT A1

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 4

3. System Architectural Design 5

4. UML Sequence Diagrams 8

5. Class Design 9

6. Data Model 10

7. System Testing 11

8. Bibliography 12

1. Requirements Analysis

# 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).

The administrator user can perform the following operations:

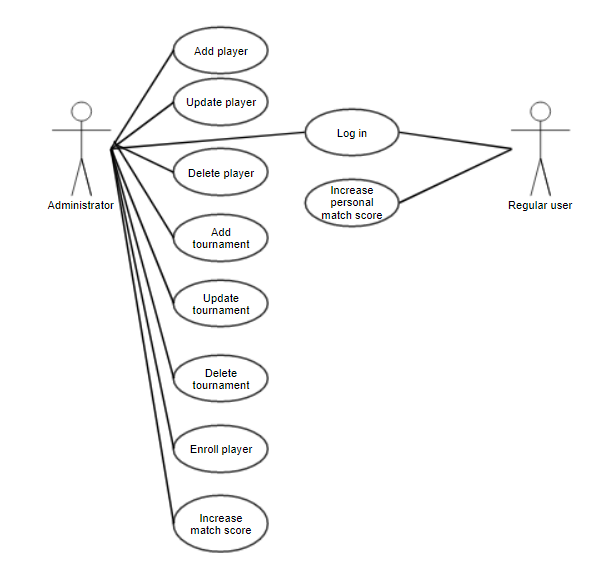
* Log in with an email and a password.
* Operations on players (users): view all players, create new player, update player and delete player.
* Operations on tournaments: view all tournaments, create new tournament, update tournament and delete tournament.
* Enroll players in tournaments manually.

# 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 player in match

**Level:** user-goal level

**Primary actors:** Administrator

**Main success scenario:**

* The administrator successfully launches the application.
* The administrator successfully logs in.
* The administrator selects a tournament form the tournament list and a player from the player list and selects the “Enroll” command.
* The player is enrolled in the tournament and automatically added in the next available match.

**Extensions:**

* If the administrator does not select a tournament, the system displays an error message.
* If the administrator does not select a player, the system displays an error message.
* If the player is already enrolled in the tournament, the system displays an error message.
* If the number of players has reached maximum (8), the system displays an error message.

**Use case:** Update personal match score

**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 and selects the “Open tournament” command.
* The user selects a match in which he takes part and hits the “Update” command.
* The user hits the “Increase” button and the score is updated (by adding one point for the player that the Increase button points to).

**Extensions:**

* If the user does not select a tournament, the system displays and error message.
* If the match is already over, the system displays a warning message.
* If the user selects a match in which he does not take part, the “Increase” commands will be blocked and he can only see the state of the match.

3. System Architectural Design

**3.1 Architectural Pattern Description**

The used architectural pattern is the **Layered Architecture** (or **Three-tier Architecture**). This architecture was used because of the non-functional requirement of the assignment that stated the use of this pattern.

The main idea behind this architecture is to separate the functionality into distinct layers, each having its own responsibility. The main constraint is that each layer has a dependency only on the next-level layer. Therefore, no dependencies from higher-level layers to lower-level ones are allowed. In other words, layer *i* can depend only on layer *i+1*, if we were to label the layers with numbers in increasing order.

The architecture of this application is Three-tier because it has exactly 3 layers:

* The user interface layer: ui
* The business layer: business
* The data access layer: dao

The user interface layer provides functionality for the interaction with the user and makes

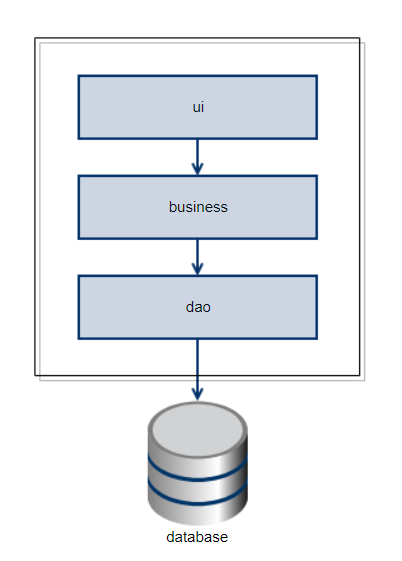
use of the business layer to respond to user commands (ui -> business).

The business layer provides the main logic of the application and provides methods for responding to user commands (such as logging in and creates, deletes and updates). The business layer makes use of dao to communicate the changes to the database and to gather information from it. (business -> dao)

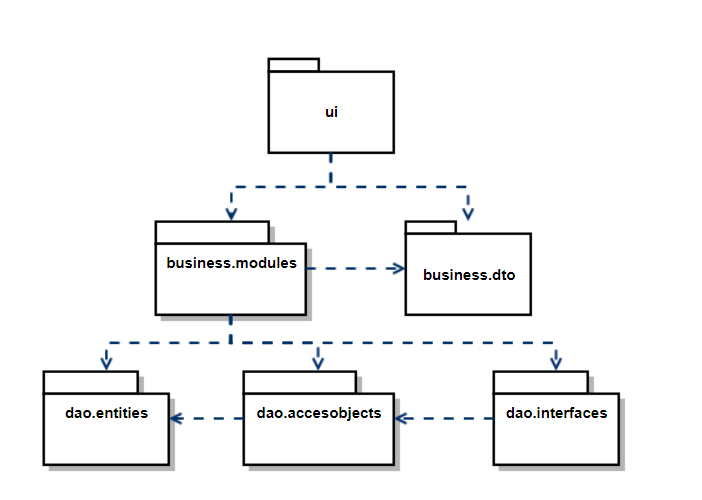
The dao layer provides the functionality for the interaction with the database, together with the main entities implied in the application such as player, tournament, match and set. This layer sets up the connection with the database and constructs the SQL statements for retrieving and updating data from/to the database. It has no dependency on other application layers.

**3.2 Diagrams**

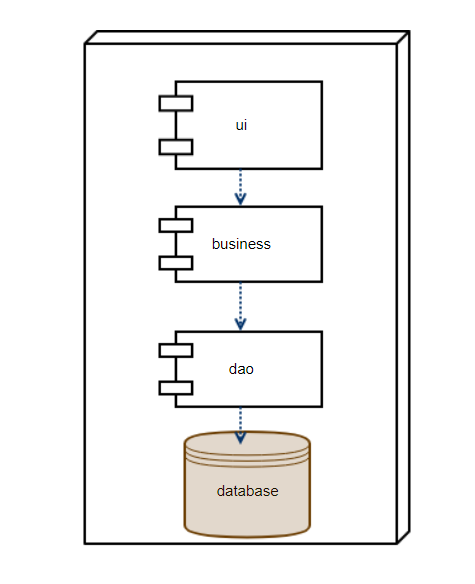
**3.2.1 Conceptual architecture**



**3.2.2. Package diagram**

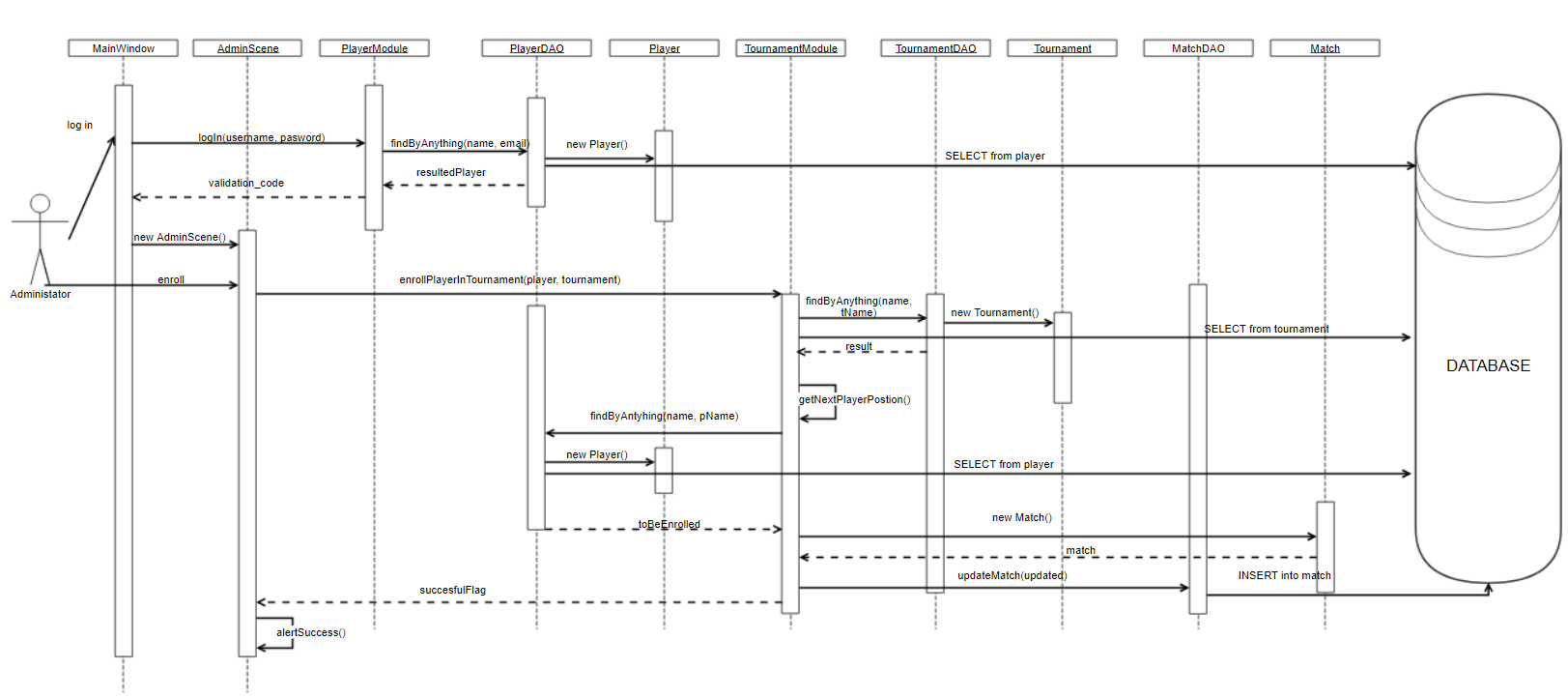


**3.2.3. Deployment diagram**



4. UML Sequence Diagrams

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



5. Class Design

**5.1 Design Patterns Description**

I used three design patterns for implementing the application, two of them imposed by the non-functional requirements and the third one for supporting the layered architecture.

**5.1.1. Transaction script**

The transaction script pattern organizes business logic by procedures where each procedure handles a single request from the presentation layer. Each module (class) handles related request (such as requests on the same table), but it often accesses more information from the database, spread among many tables.

This design pattern is very useful when the database has several relationships (dependencies) and one transaction supposes retrieving and updating data from more sources and using more data access objects.

**5.1.2 DAO design pattern**

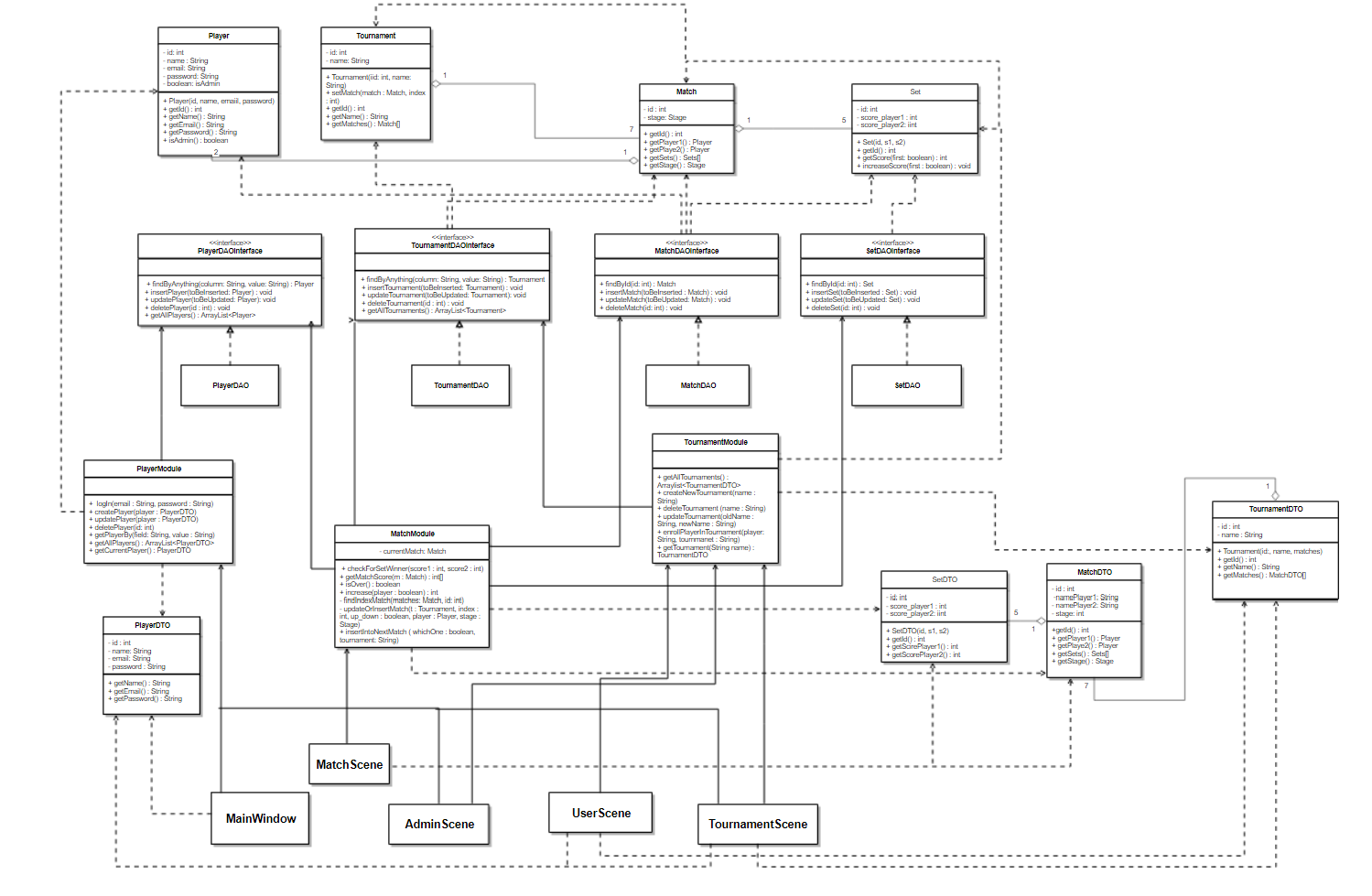
The DAO (Data access object) design pattern focuses on communicating with the database. Each entity in the domain model has associated a data access object which deals with the basic operations on that table only. The basic operation performed in a DAO class are insert, find, update and delete.

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.

**5.2 UML Class Diagram**

**

The class diagram yields the usage of the three patterns:

* The transaction script pattern is used with classes PlayerModule, TournamentModule and MatchModule. They make use of the DAO interfaces to communicate with the database from the business layer.
* The DAO design pattern is obvious because of DAO classes and DAO interfaces. The DAO classes correspond to the four entities in the dao layer: Player, Tournament, Match and Set. Each one implements the DAO interface which is used by the business layer.
* The DTO design pattern is observabile through classes TournamentDTO, MatchDTO, PlayerDTO and SetDTO that are used for communication between the business layer and the user interface layer.

The class diagram also reflects the data model in the sense that the relations between tables in the database are present in the class diagram as well as follows:

* Aggregation between Tournament and Match ~ indicates the “is made of” relationship between the two entities. One tournament is made of 7 matches.
* Aggregation between Match and Set ~ indicates the “is made of” relationship between the two entities. One match is made of 5 sets. For the purpose of simplicity, we consider that all matches are made of 5 sets, although some may end in 3 or 4 sets.
* Aggregation between Match and Player ~ indicates the fact that one match is made of two players. There are cases in which a match will have only one player, but that match cannot be updated in any way until the second player is submitted to the database.

6. Data Model

Data is stored on a local database called “ping-pong”. 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.

6.2. **tournamen**t(*id: int, name: varchar, 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 an name fields are self-explanator. 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. **match**(*id: int, player1\_id: int, player2\_id: int, set1\_id : int, set2\_id: int, set3\_id: int, set4\_id: int, set5\_id: int, stage: 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 : id of the first player participating in the match.
* Payer2\_id : id of the second player participating in the match.
* Set1\_id : id of the first set.
* Set2\_id : id of the second set
* Set3\_id : id of the third set
* Set4\_id : id of the fourth set
* Set5\_id : id of the fifth set.

The stage field specifies if the match is a quarterfinal (1), a semifinal (2) or a final (3).

6.4. **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:

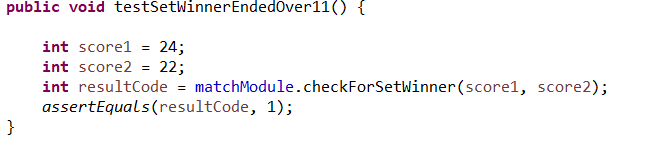
* Test for end of set detection.
* Test for end of match detection.
* Test for inserting player correctly in the database.
* Test for updating player correctly in the database.
* Test for player deletion.
* Test for inserting tournament correctly in the database.
* Test for updating tournament correctly in the database.
* Test for tournament deletion.
* Test for player enrollment correct (the enrollment is performed automatically by the system and the player gets the next available position in the tournament whenever it its enrolled).
* Test for automatically-inserted match in the right position.

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. Three-tier architecture and design patterns

Martin Fowler, Patterns of enterprise application architecture

<https://martinfowler.com/books/>

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. Support in commonly-asked programming problems

<https://stackoverflow.com/>