Instituto Superior Técnico

Mestrado Integrado em Engenharia Mecânica

OBJECT-ORIENTED PROGRAMMING

AND DATABASES

Project Presentation

Bestifute, a Football Scouting Database

Group 1

Bruno Cotrim, 87159

Duarte Pereira, 87180

José Trigueiro, 87225

Professor: Alexandra Moutinho

2020 – 2021

4th of January 2021

Table of Contents

1. Introduction

In the past few years, football as a business has experienced a huge growth, with well-known, experienced players being transferred for amounts surpassing the hundred-million-euro mark. The increasing inflation in the player transfer market has generated interest from clubs in scouting and developing players from lesser-known leagues and clubs with the objective of finding the next big star for cheaper prices. This new reality has created demand for greater amounts of data collection, allowing for a direct, numerical comparison between a larger pool of players. Clubs have also been focusing on developing a unique playing identity, so finding players that are not only talented but also fit specific needs of the team (for example, above average passing ability or high defensive work rate) becomes even more important. This project proposes a solution that would help football clubs in achieving these goals.

2. Project Description

At first, the database should allow agents and scouts to create player profiles, where they fill their biographic details (age, nationality, position, dominant foot, current club, height, weight, number of international caps). It should then allow for club scouts to insert a player's performance data, divided into physical, mental, or technical attributes, with a special category for goalkeeper attributes. It should also allow player agents to insert financial information, more specifically, current and demanded wage, current and demanded contract length and demanded transfer fee by the selling club, signing fee by the player and agent fee. When agents provide information about a player, they may provide their contact information in case there is need for further negotiation.

The most important performance attributes that were considered for players are:

- **Technical attributes:** shooting, dribbling, passing, and tackling.
- **Mental attributes:** leadership, positioning, and stamina.
- **Physical attributes:** strength and speed.
- **Goalkeeper attributes:** handling, reflexes.

From the perspective of the team's recruitment personnel, both sporting director and coach should be able to search for players, either by showing all players that fit specific key attributes or performance and financial indicators, or directly by their name. If no player can be found with the wanted attributes, a request may be left to the scouting team by the coach with the requirements for players to be identified and inserted in the database. Furthermore, the coach should also be able to flag specific players as recommendations that the sporting director should then be able to check.

3. Use Case Diagram

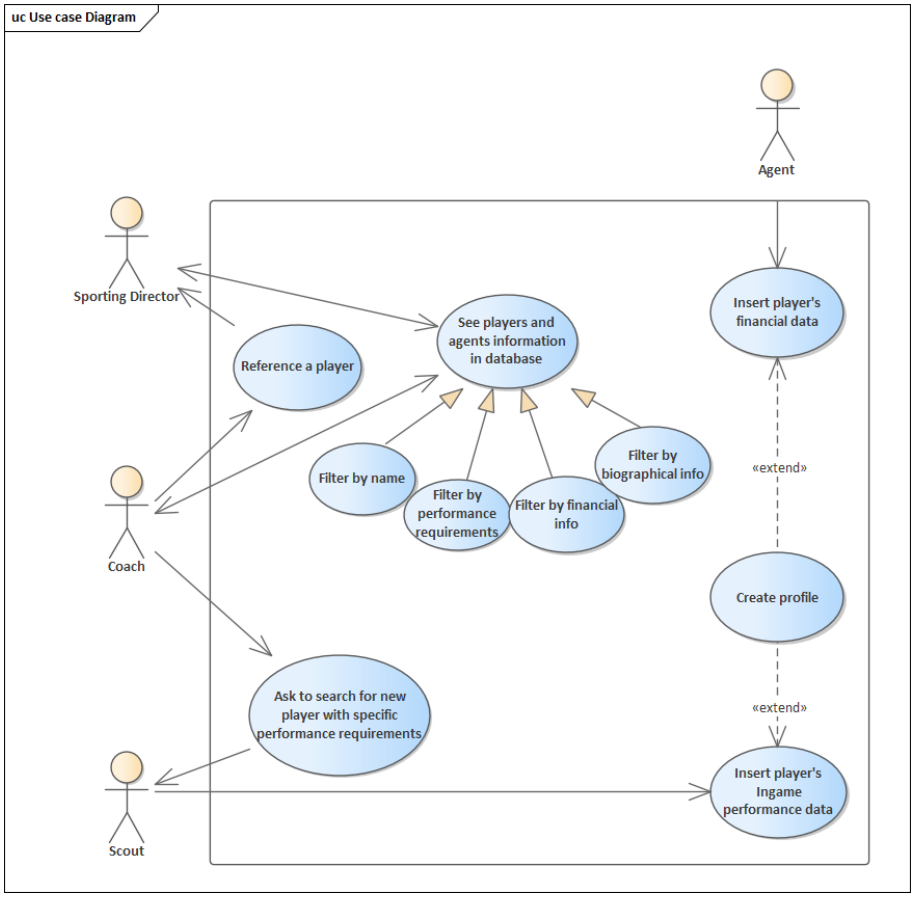

Figure 1: Use Case Diagram

Figure 1 shows the Use Case Diagram based on the project description presented in section 2, implemented using the Enterprise Architect software. Through this Diagram, the multiple described interactions available for each type of user described in the previous section are schematically organized in a summarized way.

4. Entity-Relationship Diagram

Comentado [JCMT1]: Biographical não biological

The entity-relationship (E-R) diagram shows the entities that make up the database, the relationships between them and the attributes of both. The E-R diagram from which this project was implemented is shown in Figure 2.

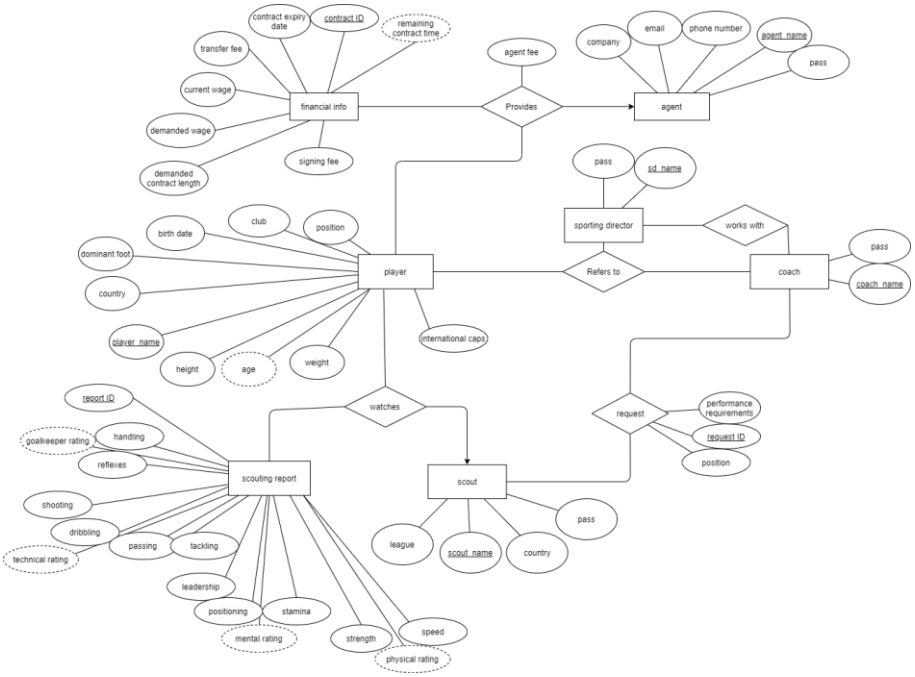

Figure 2: Entity-relation diagram

Here is provided a list of all entities, relations and attributes, and a brief explanation of each of them.

4.1. Entities

- **Player –** Refers to the profile of a football player in the database. It has the following attributes:
    - **Player name (primary key) –** contains the player's name and serves as a primary key. If two players have the same first and last name, a middle name may be added, and it was deemed very unlikely that two professional players have the same full name.

- **Position –** refers to the position the player plays on the pitch. It is identified by a two or three letter acronym (for example, LB – left back or CDM – central defensive midfielder).
- **Club –** the football club the player currently plays at.
- **Birth date –** the date of birth of the player. From it the **derived attribute age** of the player may be calculated.
- **Country –** nationality of the player.
- **International caps –** number of international (national team) games the player has participated in. It is a good indicator of the experience of the player at a high level.
- **Dominant foot –** the foot with which the player is more proficient with, a player may be right-footed, left-footed or either-footed.
- **Height –** the height of the player in centimetres.
- **Weight –** the weight of the player in kilograms.

- **Scouting Report –** information sheet provided by a scout about a player's sporting performance. Every rating is given from 0 to 10.
  - **Report id (primary key) –** unique identifying number for a scouting report.
  - **Handling –** the capacity of a goalkeeper to catch, handle and throw the ball.
  - **Reflexes –** the reaction speed of a goalkeeper when making a save.
  - **Goalkeeper rating (derived) –** average of handling and reflexes, measure of how good a goalkeeper is.
  - **Shooting –** combination of shot power, technique, and ability to aim shots at the goal of a player.
  - **Dribbling –** ability of a player to control the ball at their feet and protect it from opponent tackles.
  - **Passing –** passing accuracy, power, and distance.
  - **Tackling –** measures the ability to steal the ball from opposition players.
  - **Technical rating (derived) –** average of the four previous attributes, measuring how technically complete a player is in football's most basic actions.
  - **Leadership –** ability of the player to give orders to fellow teammates and organize the team inside the pitch.
  - **Positioning –** capacity to follow a tactic and hold a certain position as required (for example, for a striker it would be the ability to find open space behind the defence while avoiding marking and offsides).
  - **Stamina –** measures the capacity of the player to perform for an entire game, without running out of energy.

    - **Mental rating (derived) –** an average of the three previous attributes, measures how strong mentally a player is.
    - **Strength –** the physical strength of the player, affects his ability to keep the ball and withstand opposition tackles.
    - **Speed –** how fast a player can run with or without the ball.
    - **Physical rating (derived) –** an average of the two previous attributes, measures the physical ability of the player.
- **Financial Info –** financial report provided by an agent.
    - **Contract ID (primary key) –** unique identifying number for a contract provided by an agent.
    - **Contract expiry date –** date (day, month, and year) when a contract at the current club of a player expires.
    - **Transfer fee –** the fee a club must pay to a player's current club if they want to buy them.
    - **Current wage –** the wage a player currently earns at his current club.
    - **Demanded wage –** the wage the agent wants for a player if he is going to change club.
    - **Demanded contract length –** length of the contract, in years, a player demands when changing club.
    - **Signing fee –** fee that is paid to a player for the signing of a contract with a new club.
- **Scout**
    - **Scout name (primary key) –** usually first and last name of a scout, but more names can be added to differentiate since they must be unique.
    - **League** – the league a scout specializes in.
    - **Country –** scout's nationality.
    - **Pass –** password the scout uses to access the database.
- **Coach**
    - **Coach name (primary key) –** usually first and last name of a coach, but more names can be added to differentiate since they must be unique.
    - **Pass –** password the coach uses to access the database.
- **Sporting Director**
    - **SD name (primary key) –** usually first and last name of a sporting director, but more names can be added to differentiate since they must be unique.
    - **Pass –** password the sporting director uses to access the database.

- **Agent**
    - **Agent name –** usually first and last name of an agent, but more names can be added to differentiate since they must be unique.
    - **Pass –** password the agent uses to access the database.
    - **Phone number –** contact phone number of the agent.
    - **Email –** contact email of the agent.
    - **Company –** agency company the agent works at.

4.2. Relations

- **Provides –** ternary relation between **player**, **agent,** and **financial info.** An agent may provide financial info for multiple players, but a player only has a single financial information report and one agent.
    - **Agent fee –** fee the agent charges for transferring the player.
- **Watches –** ternary relation between **player**, **scout,** and **scouting report.** A scout may provide scouting reports for multiple players, but a player only has a single scouting report and is followed by one scout.
- **Request –** relation between **coach** and **scout**. It serves as a way for the coach to save a request for a scout to find a certain type of player.
    - **Position –** position that the player the coach wants must play in.
    - **Performance requirements –** a small text describing the type of player that the coach wants.
- **Refers to –** ternary relation between **player, coach,** and **sporting director.** It provides a way for the coach to reference certain players in the database that he wants for his team.
- **Works with –** relation between **coach** and **sporting director**. Since a club usually only has one of each, and a coach wants to recommend players to his club's sporting director, information about what coaches and sporting directors work in the same club must be saved.

5. Class Diagram

The class diagram is another schematical form of representing this system and is particularly relevant since once it is implemented, software Enterprise Architect can export the MySQL code that would structure the Database. The Class Diagram can be obtained considering that each Entity and Relation (from the Entity-Relation Diagram) form a table and consequently a class in the Class Diagram. In this diagram, all attributes described before are represented, with primary keys being identified with “PK”. Note that all relation tables take the primary keys of the classes they connect as foreign (primary) keys denoted “pfK”. Links between tables contain information about the equivalence between primary keys of an entity table and foreign keys of a relation table. This diagram also further specifies the structure from the entity relationship, since it contains information about attribute type: CHAR/VARCHAR for text strings, INT for integer numbers and DATE for dates. An initialization size for the attributes is also specified.

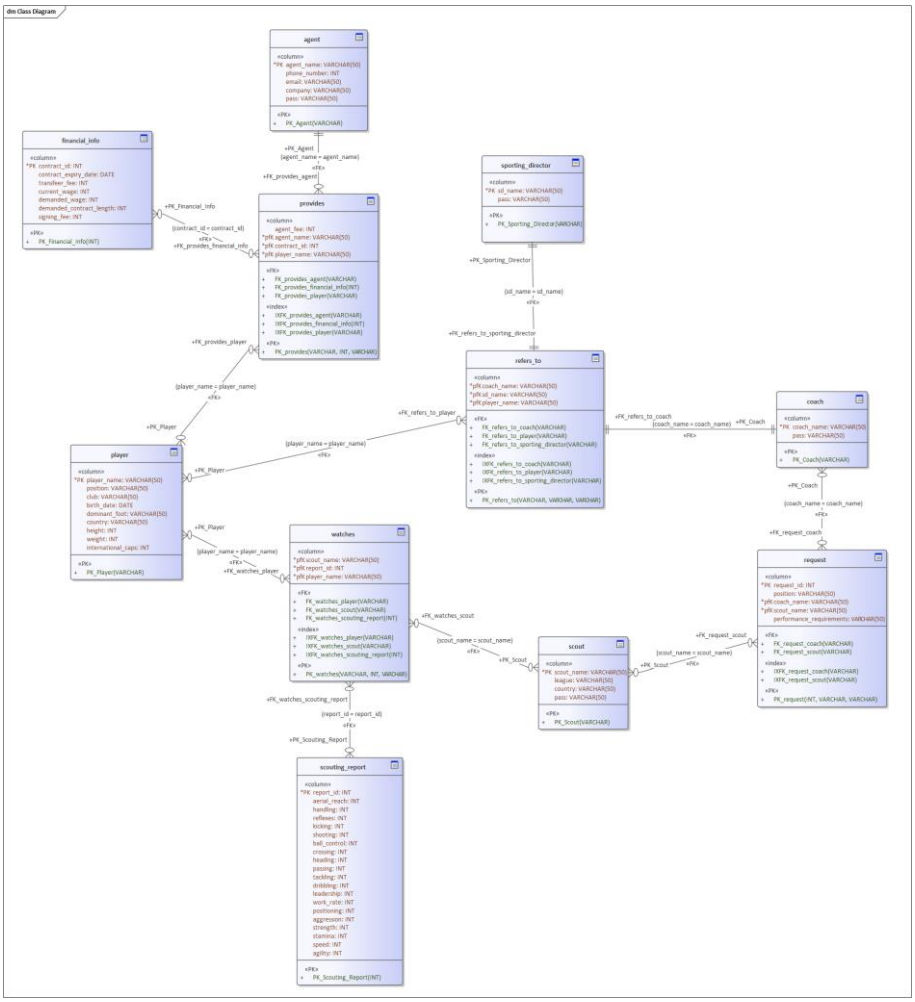

Figure 3: Class diagram

6. Database Implementation and Queries

6.1. Database Implementation

As mentioned before, the Database is initialized by importing to MYSQL the code that was generated by the Enterprise Architect class diagram. There are some changes that were made manually since the initial code had some features and extra code that were not desired.

A header was also added that allows the Database to be hosted on the IST server, deleting the preceding data that would be stored there first. This allowed the continuous update of the Database without conflicts. This portion of code is presented below.

List 1:

```
1.  DROP DATABASE IF EXISTS ist187225;
2.  CREATE SCHEMA IF NOT EXISTS ist187225;
3.  USE ist187225;
```

Tables are then created declaring their attributes, primary keys and the different types of variables associated with each one.

List 2:

```
1.  CREATE TABLE `agent`
2.  (`agent_name` VARCHAR(50) NOT NULL,
3.      `phone_number` INT NULL,
4.      `email` VARCHAR(50) NULL,
5.      `company` VARCHAR(50) NULL,
6.      `pass` VARCHAR(50) NULL,
7.      PRIMARY KEY (`agent_name` ASC) );
8.
9.  CREATE TABLE `coach`
10. (`coach_name` VARCHAR(50) NOT NULL,
11.     `pass` VARCHAR(50) NULL,
12.     PRIMARY KEY (`coach_name` ASC) );
13.
14. CREATE TABLE `financial_info`
15. (`contract_id` INT NOT NULL,
16.     `contract_expiry_date` DATE NULL,
17.     `transfer_fee` INT NULL,
18.     `current_wage` INT NULL,
19.     `demanded_wage` INT NULL,
20.     `demanded_contract_length` INT NULL,
21.     `signing_fee` INT NULL,
22.     PRIMARY KEY (`contract_id` ASC) );
23.
24. CREATE TABLE `player`
25. (`player_name` VARCHAR(50) NOT NULL,
26.     `position` VARCHAR(50) NULL,
27.     `club` VARCHAR(50) NULL,
28.     `birth_date` DATE NULL,
29.     `dominant_foot` VARCHAR(50) NULL,
30.     `country` VARCHAR(50) NULL,
31.     `height` INT NULL,
32.     `weight` INT NULL,
33.     `international_caps` INT NULL,
34.     PRIMARY KEY (`player_name` ASC) );
35.
36.
```

```
CREATE TABLE `provides`
(`agent_name` VARCHAR(50) NOT NULL,
    `contract_id` INT NOT NULL,
    `player_name` VARCHAR(50) NOT NULL,
    `agent_fee` INT NULL,
    PRIMARY KEY (`agent_name` ASC, `contract_id` ASC, `player_name` ASC) );

CREATE TABLE `refers_to`
(`coach_name` VARCHAR(50) NOT NULL,
    `sd_name` VARCHAR(50) NOT NULL,
    `player_name` VARCHAR(50) NOT NULL,
    PRIMARY KEY (`coach_name` ASC, `sd_name` ASC, `player_name` ASC) );

CREATE TABLE `request`
(`request_id` INT NOT NULL,
    `position` VARCHAR(50) NULL,
    `coach_name` VARCHAR(50) NOT NULL,
    `scout_name` VARCHAR(50) NOT NULL,
    `p_requirements` VARCHAR(200) NULL,
    PRIMARY KEY (`request_id` ASC, `coach_name` ASC, `scout_name` ASC) );

CREATE TABLE `scout`
(`scout_name` VARCHAR(50) NOT NULL,
    `league` VARCHAR(50) NULL,
    `country` VARCHAR(50) NULL,
    `pass` VARCHAR(50) NULL,
    PRIMARY KEY (`scout_name` ASC) );

CREATE TABLE `scouting_report`
(`report_id` INT NOT NULL,
    `handling` INT NULL,
    `reflexes` INT NULL,
    `shooting` INT NULL,
    `passing` INT NULL,
    `tackling` INT NULL,
    `dribbling` INT NULL,
    `leadership` INT NULL,
    `positioning` INT NULL,
    `strength` INT NULL,
    `stamina` INT NULL,
    `speed` INT NULL,
    PRIMARY KEY (`report_id` ASC) );

CREATE TABLE `sporting_director`
(`sd_name` VARCHAR(50) NOT NULL,
    `pass` VARCHAR(50) NULL,
    PRIMARY KEY (`sd_name` ASC) );

CREATE TABLE `watches`
(`scout_name` VARCHAR(50) NOT NULL,
    `report_id` INT NOT NULL,
    `player_name` VARCHAR(50) NOT NULL,
    PRIMARY KEY (`scout_name` ASC, `report_id` ASC, `player_name` ASC) );

CREATE TABLE `works_with`
(`coach_name` VARCHAR(50) NOT NULL,
    `sd_name` VARCHAR(50) NOT NULL,
    PRIMARY KEY (`coach_name` ASC, `sd_name` ASC) );
```

Having created the tables, only the Foreign Keys need to be added, which guarantee the connection between the different tables, ensuring the untroubled functioning of the relations.

Foreign Keys Declaration:

```
ALTER TABLE `provides`
ADD FOREIGN KEY (`agent_name`) REFERENCES `agent` (`agent_name`)
ON DELETE CASCADE;
ALTER TABLE `provides`
ADD   FOREIGN KEY (`contract_id`) REFERENCES `financial_info` (`contract_id`)
ON DELETE CASCADE;
ALTER TABLE `provides`
 ADD  FOREIGN KEY (`player_name`) REFERENCES `player` (`player_name`)
ON DELETE CASCADE;
ALTER TABLE `refers_to`
 ADD  FOREIGN KEY (`coach_name`) REFERENCES `coach` (`coach_name`)
ON DELETE CASCADE;
ALTER TABLE `refers_to`
 ADD  FOREIGN KEY (`player_name`) REFERENCES `player` (`player_name`)
ON DELETE CASCADE;
ALTER TABLE `refers_to`
 ADD  FOREIGN KEY (`sd_name`) REFERENCES `sporting_director` (`sd_name`)
ON DELETE CASCADE;
ALTER TABLE `request`
 ADD  FOREIGN KEY (`coach_name`) REFERENCES `coach` (`coach_name`)
ON DELETE CASCADE;
ALTER TABLE `request`
 ADD  FOREIGN KEY (`scout_name`) REFERENCES `scout` (`scout_name`)
ON DELETE CASCADE;
ALTER TABLE `watches`
 ADD  FOREIGN KEY (`player_name`) REFERENCES `player` (`player_name`)
ON DELETE CASCADE;
ALTER TABLE `watches`
 ADD  FOREIGN KEY (`scout_name`) REFERENCES `scout` (`scout_name`)
ON DELETE CASCADE;
ALTER TABLE `watches`
 ADD  FOREIGN KEY (`report_id`) REFERENCES `scouting_report` (`report_id`)
ON DELETE CASCADE;
ALTER TABLE `works_with`
 ADD  FOREIGN KEY (`coach_name`) REFERENCES `coach` (`coach_name`)
ON DELETE CASCADE;
ALTER TABLE `works_with`
 ADD  FOREIGN KEY (`sd_name`) REFERENCES `sporting_director` (`sd_name`)
ON DELETE CASCADE;
```

After the database was implemented, .csv files were created with the information to be stored. Presented below is an example of both the .csv file and the SQL code that loads it into the appropriate table this step (other tables follow the same format).

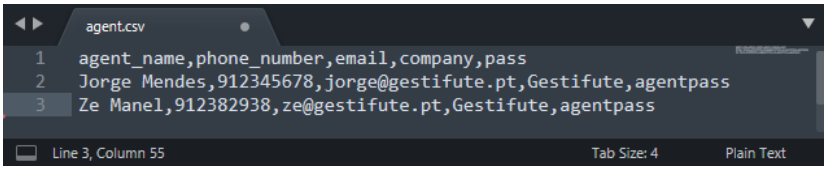

Figure 4: Example agent.csv file

CSV Loading code:

```
LOAD DATA LOCAL INFILE 'C:\\Users\\duart\\Documents\\IST\\POBD\\BEStifute\\data\\agent.csv'
INTO TABLE agent
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\r\n'
IGNORE 1 ROWS;
```

6.2. Queries

To implement queries without later having problems using them in the C ++ Application, it is important to bear in mind that the maximum limit per query is 256 characters. To avoid this type of implementation problems, *Stored Procedures* were created for some of the queries and stored in the Server. Using *Stored Procedures* is also considered to be safer against threats like, *SQL injection,* and allows a more modular and organized code.

In the following points of the report will be shown, both the *Stored Procedure calling sentence* (for the queries that were implemented this way) and the Query itself.

Get Player Info

Description**:** This query was created in a modular way to allow the user to customize their search parameters. In its most basic form, the query has the following SQL code:

Relational Algebra:

$$\pi_{\text{player\_name,club,agent\_name,transfer\_fee,demanded\_wage}}(\text{player} \bowtie \text{provides} \bowtie \text{financial\_info} \bowtie \text{agent} \bowtie \text{scouting\_report} \bowtie \text{watches})$$

SQL:

Get Player Info Query:

```
1. SELECT P.player_name, P.club, A.agent_name, FI.transfer_fee, FI.demanded_wage
2. FROM player AS P,agent AS A, provides AS PR, financial_info AS FI, watches AS W, sc
   outing_report AS SR
3. WHERE P.player_name = PR.player_name AND PR.contract_id = FI.contract_id AND A.agen
   t_name = PR.agent_name AND P.player_name = W.player_name AND W.report_id = SR.repor
   t_id
```

This code displays all important information about all players in the database. Note that this query ends in a series of AND statements that clear all the useless information produced by the cartesian product of tables *player*, *agent*, *provides*, *financial_info, watches,* and *scouting report*. The lack of semicolon at the end of the query is deliberate and is necessary to add further constraints to the query. Since the program must allow the user to customize which requirements, they want the players to fulfil, the following types of statements may be appended to the end of this query:

- **Performance Constraints:**

These types of constraints relate to minimum value that the player must have in one of the performance parameters, and consists in adding the following to the query, where **requirement** is the type of requirement the player must fit (any of the non-primary or derived

attributes of table scouting report) and **min_value** is the minimum value that requirement should have:

Relational Algebra:

$$\pi_{\dots}\left(\sigma_{scouting\_report.requirement \geq \text{min\_value}}(\text{player} \bowtie \text{provides} \bowtie \cdots)\right)$$

SQL:

Performance Requirement constraint:

```
AND SR.requirement >= min_value
```

- **Financial Constraints:**

This type of constraints concerns a maximum value that may be payed for a player. The structure of the query is the following, where the **fee** parameter may be either the transfer fee, demanded wage or the total fee (defined, as seen below, by the sum of transfer fee paid to the player's club, signing fee paid to the player and agent fee paid to the agent) and **max_value** the maximum allowed value for that fee.

Relational Algebra:

$$\pi_{\dots}\left(\sigma_{\text{fee} < \text{max\_value}}(\text{player} \bowtie \text{provides} \bowtie \cdots)\right)$$

SQL:

Financial Requirement constraint:

```
AND FI.fee <= max_value
AND (FI.transfer_fee + FI.signing_fee + PR.agent_fee) < max_value
```

- **Biographic Constraints:**

These constraints deal with the problem where the user searches for a player with specific biographic constraints (name, position, or dominant foot). It is written as below, where the **requirement** represents either *player_name*, *position,* or *foot*, and **inserted_value** the value the user inserts for that requirement (for example, a particular player's name or "L" for a left footed player).

Relational Algebra:

$$\pi_{\dots}\left(\sigma_{\text{requirement} = \text{inserted\_value}}(\text{player} \bowtie \text{provides} \bowtie \cdots)\right)$$

SQL:

Biographic Requirement constraint:

```
AND P.requirement = inserted_value
```

- **Age Constraints:**

Another important parameter to select players is the maximum age, since after a certain age a player may not be as attractive for a club due to his approaching retirement. However, due to memory constraints and robustness, age is a derived attribute, calculated from current day and from the player's birth date. In MySQL, this may be done using functions CURDATE, which gives the current data, and TIMESTAMPDIFF, giving the difference between two dates, in this case the difference in years between birth date and current date. The constraint is implemented as follows, with **max_age** being the inserted desired maximum age for the players:

SQL:

Age Requirement constraint:

```
1.  AND TIMESTAMPDIFF(YEAR,P.birth_date,CURDATE()) <= max_age
```

In relational algebra the time difference functions are not available, so assuming age is directly available the query is reduced to the same format as the financial requirement constraints.

Result:

As a test, selecting only the basic query, Figure 5 is obtained.

player_name	club	agent_name	transfer_fee	demanded_wage
Cristiano Ronaldo	Juventus	Jorge Mendes	100	700
Ze Antonio	Tondela	Jorge Mendes	100	700

Figure 5 – Result from the basic query

Adding a constraint, for age lower than 27, which player Ze Antonio fits but player Cristiano Ronaldo does not, the result is what is shown on Figure 6.

player_name	club	agent_name	transfer_fee	demanded_wage
Ze Antonio	Tondela	Jorge Mendes	100	700

Figure 6 – Result from the added constraint

Check login

Description**:** This query has the goal of returning the login data of the user. It was coded to be able to be used by the four type of different users. The inputs that go in the query are the type of user, **logintype** (agent, scout, sporting director or coach), and the name of the user, **inputedname**. The outputs consist in two columns, with the username and the password of that specific user.

Relational Algebra:

$$\pi_{logintype\_name,pass}\left(\sigma_{agent\_name=inputedname}(logintype\_name)\right)$$

SQL:

Check Login Query:

```
1.  SELECT logintype_name,pass FROM logintype
2.  WHERE logintype.logintype_name = inputedname
```

Result:

For testing purposes, the inputs are agent as **logintype** and 'Jorge Mendes' as **inputedname**, with the results shown in Figure 7.

	agent_name	pass
▶	Jorge Mendes	agentpass

Figure 7 - Result of Query CheckLogin

Check rating (goalkeeper, technical, mental, physical)

Description**:** These four ratings are derived attributes that can be calculating the average value of other, more specific, ratings. The input for these four queries is the **player_name_in,** from which the users intend to know the ratings.

Relational Algebra: This type of Queries involves algebraic operations between columns. Relational Algebra is designed to perform operations between values in the same column, therefore these four queries were only implemented in SQL.

SQL:

Goalkeeper Rating Query:

```
1.  SELECT (sdr.handling+sdr.reflexes)/2 AS goalkeeper_rating
2.  FROM scouting_report AS sdr, player AS P, watches AS W
3.  WHERE P.player_name=player_name_in AND P.player_name=W.player_name
4.  AND W.report_id=sdr.report_id;
```

Goalkeeper Rating Query calling sentence:

```
1.  CALL gk_rating(player_name_in)
```

Technical Rating Query:

```
1.  SELECT (sdr.shooting+sdr.dribbling+sdr.passing+sdr.tackling)/4 AS technical_rating
2.  FROM scouting_report AS sdr, player AS P, watches AS W
3.  WHERE P.player_name=player_name_in AND P.player_name=W.player_name
4.  AND W.report_id=sdr.report_id;
```

Technical Rating Query calling sentence:

```
1.  CALL tech_rating(player_name_in)
```

Mental Rating Query:

```
1.  SELECT (sdr.leadership+sdr.positioning+sdr.stamina)/3 AS mental_rating
2.  FROM scouting_report AS sdr, player AS P, watches AS W
3.  WHERE P.player_name=player_name_in AND P.player_name=W.player_name
4.  AND W.report_id=sdr.report_id;
```

Mental Rating Query calling sentence:

```
1.  CALL ment_rating(player_name_in)
```

Physical Rating Query:

```
1.  SELECT (sdr.strength+sdr.speed)/2 AS physical_rating
2.  FROM scouting_report AS sdr, player AS P, watches AS W
3.  WHERE P.player_name=player_name_in AND P.player_name=W.player_name
4.  AND W.report_id=sdr.report_id;
```

Physical Rating Query calling sentence:

```
1.  CALL phys_rating(player_name_in)
```

Result:

For testing purposes, the input is ‘Cristiano Ronaldo’ as **player_name_in**, with the resulting table being shown in Figure 8.

goalkeeper_rating	technical_rating	mental_rating	physical_rating
0.0000	7.7500	8.3333	9.0000

Figure 8 - Check Rating Results for 'Cristiano Ronaldo'

Get Scouting Report Info

Description: This query is structured similarly to the precious one however it returns all attributes of the scouting report (except primary and derived). The input for this query is the **player_name_in,** identifying the player from which the user intends to know the ratings.

Relational Algebra:

$$\pi_{handling,reflexes,shooting,passing,tackling,dribbling,leadership,positioning,strength,stamina,speed}$$

$$(\sigma_{player\_name=player\_name\_in}\ (watches \bowtie scouting\_report \bowtie player))$$

SQL:

Get Scouting Report Info Query:

```
1. SELECT SCR.handling, SCR.reflexes, SCR.shooting, SCR.passing, SCR.tackling,SCR.dribbling, SCR.leadership, SCR.positioning, SCR.strength, SCR.stamina, SCR.speed
2. FROM player AS P, watches AS W, scouting_report AS SCR
3. WHERE P.player_name=W.player_name AND SCR.report_id=W.report_id AND P.player_name=player_name_in;
```

Get Scouting Report Info Query calling sentence:

```
1. CALL sc_report_info_rating(player_name_in)
```

Results:

Applying ‘Cristiano Ronaldo’ as the **player_name_in** input, the result is shown in Figure 9.

	handling	reflexes	shooting	passing	tackling	dribbling	leadership	positioning	strength	stamina	speed
▶	0	0	10	8	8	5	7	10	10	8	8

Figure 9 - Check Rating Results for 'Cristiano Ronaldo'

Get Financial Info

Description**:** This Query returns the financial information attributes. The input for this query is the **player_name_in,** from which the user wants to see said attributes.

Relational Algebra:

$$\pi_{transfer\_fee,signing\_fee,current\_wage,contract\_expiry\_date,demanded\_wage,demanded\_contract\_length}$$
$$(\sigma_{player\_name\ =\boldsymbol{player\_name\_in}}(provides \bowtie financial\_info \bowtie player))$$

SQL:

Get Financial Info Query:

```
1.  SELECT F.transfer_fee, F.signing_fee, F.current_wage, F.contract_expiry_date,
2.  F.demanded_wage, F.demanded_contract_length
3.  FROM player AS P, provides AS PR, financial_info AS F
4.  WHERE P.player_name=player_name_in AND P.player_name=PR.player_name
5.  AND PR.contract_id=F.contract_id;
```

Get Financial Info Query calling sentence:

```
1.  CALL getfinancial_info(player_name_in)
```

Results:

Again, 'Cristiano Ronaldo' as **player_name_in** for testing, with the output represented in Figure 10.

	transfer_fee	signing_fee	current_wage	contract_expiry_date	demanded_wage	demanded_contract_length
▶	100	20	500	2023-06-30	700	3

Figure 10 - Get Financial Info Results for 'Cristiano Ronaldo'

Get Agent Info

Description**:** This Query intends to return the personal information of an agent representing a specific player. The input for this query is the **player_name_in,** since every player has an associated agent, whose information is returned.

Relational Algebra:

$$\pi_{agent\_name,phone\_number,email,company,agent\_fee}$$
$$(\sigma_{player\_name\ =player\_name\_in}(provides \bowtie agent \bowtie player))$$

SQL:

Get Agent Info Query:

```
1.  SELECT A.agent_name, A.phone_number, A.email, A.company, PR.agent_fee
2.  FROM player AS P, provides AS PR, agent AS A
3.  WHERE P.player_name=player_name_in AND P.player_name=PR.player_name
4.  AND PR.agent_name=A.agent_name;
```

Get agent Info Query calling sentence:

```
2.  CALL agent_info(player_name_in)
```

Results:

The output for input ‘Cristiano Ronaldo’ as **player_name_in** is shown in Figure 11.

agent_name	phone_number	email	company	agent_fee
Jorge Mendes	912345678	jorge@gestifute.pt	Gestifute	3

Figure 11 - Get Agent Info Results for ‘Cristiano Ronaldo’

Get Bio Info

Description**:** This query returns information regarding the biographical aspects of the player to inspect, with input **player_name_in,** containing the name of the player of interest.

Relational Algebra:

$$\pi_{position,club,birth\_date,dominant\_foot,country,height,weight.international\_caps}$$
$$(\sigma_{player\_name\ =player\_name\_in})$$

SQL:

Get Bio Info Query:

```
1. SELECT P.position, P.club, P.birth_date, P.dominant_foot, P.country, P.height,
   P.weight, P.international_caps
2. FROM player AS P
3. WHERE P.player_name=player_name_in;
```

Get Bio Info Query calling sentence:

```
1. CALL agent_info(player_name_in)
```

Results:

Testing for the input 'Cristiano Ronaldo' as **player_name_in**, the result is shown in Figure 12.

	position	club	birth_date	dominant_foot	country	height	weight	international_caps
▶	EE	Juventus	1985-02-05	R	Portugal	187	83	170

Figure 12 - Get Bio Info Results for 'Cristiano Ronaldo'

Get scouting report/contract ID

Description**:** These queries are done when adding a player to the database or when editing an existing one, to access the last scouting report/contract to then increment (when adding a new player) or the scouting report / financial report ID associated with a player (when editing an existing player, with **player_name_in** as input).

Relational Algebra:

Get last scouting report id:

$$g_{\max(report\_id)}(works)$$

Get last contract id:

$$g_{\max(contract\_id)}(provides)$$

Get player's scouting report id:

$$\pi_{report\_id}(\sigma_{player\_name=player\_name\_in}(watches))$$

Get player's contract id:

$$\pi_{contract\_id}(\sigma_{player\_name=player\_name\_in}(provides))$$

SQL:

Get last scouting report id:

```
1.  SELECT MAX(report_id)
2.  FROM watches
```

Get last contract id:

```
1.  SELECT MAX(contract_id)
2.  FROM provides
```

Get player's scouting report id:

```
1.  SELECT report_id
2.  FROM watches
3.  WHERE player_name = player_name_in
```

Get player's contract id:

```
1.  SELECT contract_id
2.  FROM provides
3.  WHERE player_name = player_name_in
```

Results:

Results from each of the queries for when the database has only two players (getting the max id) with when the players name given is "Cristiano Ronaldo" are shown in Figure 13.

MAX(report_id)
2

MAX(contract_id)
2

report_id
1

contract_id
1

Figure 13: Results from the queries related to the IDs

Refer Player

Description**:** This query is used by a coach to add a certain player as a reference to the sporting director that works with him. In the real database, to facilitate interaction with the MFC application and promote modularity, this query was divided in two, one to get the sporting director name and another to reference the player to said sporting director. For compactness, the queries are shown here combined, where **current_coach** is the name of the coach referring the player and **ref_player** is the name of the referenced player.

Relational Algebra:

$$\pi_{coach\_name,sd\_name,player\_name}(\sigma_{player\_name=ref\_player}(\pi_{coach\_name,sd\_name}$$

$$\left(\sigma_{coach\_name=current\_coach}(\text{coach} \bowtie \text{works}_{\text{with}})\right) * player))$$

SQL:

Refer Player Query:

```
1.  INSERT INTO refers_to
2.  SELECT curr_coach, temp.sd_name, ref_player
3.  FROM
4.      (SELECT sd_name FROM coach AS C, works_with AS W
5.       WHERE C.coach_name = W.coach_name AND W.coach_name = curr_coach) AS temp
```

Result:

When setting **curr_coach** to “Jorge Juses” and **ref_player** to “Cristiano Ronaldo”, the line in Figure 14 is added to the table (keep in mind that table *works_with* has the information that sporting director “André Martins” works with coach “Jorge Juses”).

	coach_name	sd_name	player_name
▶	Jorge Juses	Andre Martins	Cristiano Ronaldo

Figure 14 – Result from referred player query

Check Referred Players

Description**:** This query is used when a sporting director wants to check which players were recommended to him by the coach. The input to this query is **sd_name_in**, the name of the sporting director that is checking for recommended players.

Relational Algebra:

$$\pi_{\text{player\_name,club,agent\_name,transfer\_fee,demanded\_wage}}(player \bowtie provides \bowtie financial\_info \bowtie agent \bowtie \sigma_{sd\_name=sd_{name_{in}}}(refers\_to))$$

SQL:

Check Referred Player Query:

```
1.  SELECT P.player_name, P.club, A.agent_name, F.transfer_fee, F.demanded_wage
2.  FROM player AS P, agent AS A,provides AS PR, financial_info AS F, refers_to AS R
3.  WHERE P.player_name=PR.player_name AND A.agent_name=PR.agent_name AND F.contract_id
    =PR.contract_id AND P.player_name=R.player_name AND R.sd_name=sd_name_in;
```

Check Referred Player Query calling sentence:

```
2.  CALL reffromcoach(sd_name_in)
```

Result:

When setting **sd_name_in** to “André Martins”, two players are shown in Figure 15 as recommended by “Jorge Juses”.

	player_name	club	agent_name	transfer_fee	demanded_wage
▶	Cristiano Ronaldo	Juventus	Jorge Mendes	100	700
	Ze Antonio	Tondela	Jorge Mendes	100	700

Figure 15 – Result from Check Referred Player Query

Create Scouting Request

Description**:** This query is used when a coach wants to leave a message to the scout to ask him to scout an additional player. The coach provides a player position, **position_in**, and performance requirements in text, **p_requirements_in**, in addition to his name **coach_name_in** and the name of the scout to fulfil the request, **scout_name_in**.

Relational Algebra:

$$request \longleftarrow request \cup ((g_{\max(request\_id)}(request) + 1) \bowtie$$

$$\{(position\_in, coach\_name\_in, scout\_name\_in, p\_requirements\_in)\})$$

SQL:

In SQL, this query must consider two cases, one where a request is being added for the first time to the database, when the *report_id* is initialized as 1, and when additional requests are added, incrementing the *report_id* instead.

Create Scouting Request Query:

```
IF (SELECT request.request_id FROM request) THEN
INSERT INTO request (request_id,position,coach_name,scout_name,p_requirements)
SELECT MAX(request.request_id) + 1, position_in, coach_name_in, scout_name_in,
p_requirements_in FROM request;
ELSE
INSERT INTO request (request_id,position,coach_name,scout_name,p_requirements)
SELECT 1, position_in, coach_name_in, scout_name_in, p_requirements_in
FROM request;
END IF;
```

Create Scouting Request calling sentence:

```
CALL addnewrequest (position_in, coach_name_in, scout_name_in, p_requirements_in)
```

Result:

By calling the function with **position_in** “MC”, **coach_name_in “**Joao Silva**”, scout_name_in** “Manuel Silva” and **p_requirements_in** “Big player with defensive capabilities”, the result from

	coach_name	position	p_requirements
▶	Joao Silva	MC	Big player with defensive capabilities

Figure 16 – Result of query that inserts a new request in the database

Get Requests from Coach

Description**:** This query is done whenever the logged scout wants to see all the requests from the coach. The input of this query is the scout name, **scout_name_in**.

Relational Algebra:

$$\pi_{coach\_name,position,p\_requirements}(\sigma_{scout\_name=scout\_name\_in}(requests))$$

SQL:

Get Requests from Coach Query:

```
1.  SELECT R.coach_name, R.position, R.p_requirements
2.  FROM request AS R
3.  WHERE R.scout_name=scout_name_in;
```

Get Requests from Coach calling sentence:

```
1.  CALL getrequests (scout_name_in)
```

Results:

Calling the results with input **scout_name "**Joao Silva**",** the line inserted in the previous query is shown, as seen in Figure 17.

	coach_name	position	p_requirements
▶	Joao Silva	MC	Big player with defensive capabilities

Figure 17: Result of the query that gets requests from coach

Add New rows

Description: The remaining queries described will be related to change the database. The first one allows adding new rows with information in the existing tables, for example, when a new player is added. As the table that would receive the new row of information is represented through **table_in.**

Relational Algebra:

$$table\_in \leftarrow table\_in \cup \{(new\_value_{row1}, new\_value_{row2}, ..)\}$$

SQL:

Add New rows Query:

```
1. INSERT INTO table_in
2. VALUES (new_value_row1, new_value_row2, ...);
```

Add new rows Query calling sentence:

```
1. CALL addnewtable_in (new_value_row1, new_value_row2)
```

Results:

As example considering the table, *player,* adding the player *David Michael*¸ and its attributes is provided below, with Figure 18 representing the *player* table before and Figure 19 after the addition of the player.

player_name	position	club	birth_date	dominant_foot	country	height	weight	international_caps
Cristiano Ronaldo	EE	Juventus	1985-02-05	R	Portugal	187	83	170
Ze Antonio	MC	Tondela	1995-02-05	R	Brazil	187	83	1

Figure 18: Table *player* before adding David Michael

player_name	position	club	birth_date	dominant_foot	country	height	weight	international_caps
Cristiano Ronaldo	EE	Juventus	1985-02-05	R	Portugal	187	83	170
David Michael	CB	Arsenal	1998-06-30	L	England	183	75	3
Ze Antonio	MC	Tondela	1995-02-05	R	Brazil	187	83	1

Figure 19:Table player after adding David Michael

Edit rows

Description: The next query is related to edit existing tables. The table that would receive the updated row of information is represented through **table_in** and the row that would suffer this update in this case will be considered the one with the attribute, ***row_to_update***.

Relational Algebra:

$$table\_in \leftarrow \pi_{new\_value_{row1}, new\_value_{row2}, \ldots} (\sigma_{row=row\_to\_update}(table\_in))$$

SQL:

Add New rows Query:

```
1. UPDATE table_in
2. SET value_row1=new_value_row1, value_row2=new_value_row2,…
3. WHERE row=row_to_update;
```

Add new rows Query calling sentence:

```
2. CALL edittable_in (row_to_update,new_value_row1, new_value_row2)
```

Results:

As example considering the table, *player,* editing the player *David Michael's* nationality the result is:

player_name	position	club	birth_date	dominant_foot	country	height	weight	international_caps
Cristiano Ronaldo	EE	Juventus	1985-02-05	R	Portugal	187	83	170
David Michael	CB	Arsenal	1998-06-30	L	England	183	75	3
Ze Antonio	MC	Tondela	1995-02-05	R	Brazil	187	83	1

Figure 21:Table player before editing David Michael's country

player_name	position	club	birth_date	dominant_foot	country	height	weight	international_caps
Cristiano Ronaldo	EE	Juventus	1985-02-05	R	Portugal	187	83	170
David Michael	CB	Arsenal	1998-06-30	L	Scotland	183	75	3
Ze Antonio	MC	Tondela	1995-02-05	R	Brazil	187	83	1

Figure 20: Table player after editing David Michael's country

Delete Rows

Description: At last, the final query is related to removing rows from the database, for example, when removing a player that retired. As the table that would see one of its rows deleted, is considered as general for this example as **role** (agent, coach, scout or sporting director) and the row that would be deleted in this case would be the one associated with the user referred as, ***inserted_name.***

Relational Algebra:

$$role \longleftarrow role - \left(\sigma_{role\_name=inserted\_name}(role)\right)$$

SQL:

Delete Rows Query:

```
4.  DELETE FROM role
5.  WHERE role_name = inserted_name;
```

Delete Rows Query calling sentence:

```
3.  CALL delete_from_role (inserted_name)
```

In SQL there is also the problem of the dependency of foreign keys from the primary keys that are deleted. The solution is to force deletion of entries in the relation tables where the foreign key entry is equal to the deleted primary key, by adding the **ON DELETE CASCADE** command to the foreign key creation. An example is shown below.

Delete Foreign Key command:

```
1.  ALTER TABLE `relation`
2.  ADD FOREIGN KEY (`role_name`) REFERENCES `role` (`role_name`)
3.  ON DELETE CASCADE;
```

Results:

As example considering the table, *player,* editing the player *David Michael's* nationality the result is:

7. C++ App User Manual

The application developed can be used by four different entities: coaches, sporting directors, scouts, and agents. When starting the app, the user is presented four options, where he should select the one that relates to him, as shown in Figure 22.

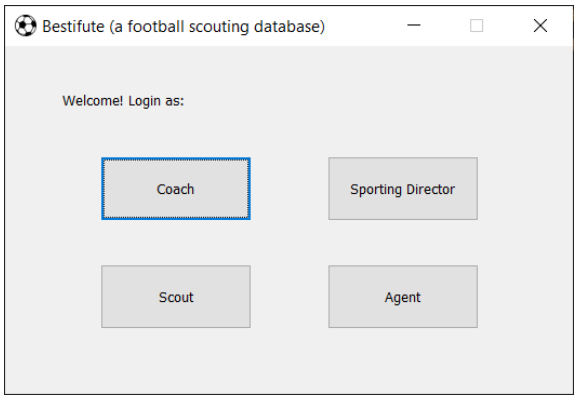

Figure 22: Initial menu

7.1. Login Window

Upon selecting an option, a window (Figure 23) pops in which the user must insert his credentials (*username* and *password*, stored in the data base) to access his menu and the query *Check login*

is done. New users can only be added by the administrator of the database in MySQL, as well as deleting accounts or changing the password.

If the name or the password do not match, a warning is presented accordingly (where the *password* field is cleared).

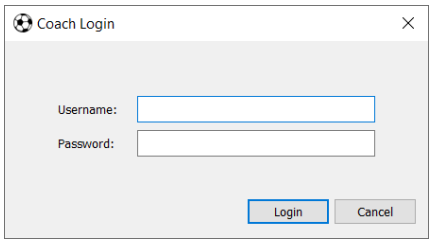

Figure 23: Login window

7.2. Coach and Sporting Director Manual

In this section the types of user covered will be the coach and the sporting director, as their main functionality is very similar, which is the ability to search the database. Menus for both types of user are shown in Figure 24.

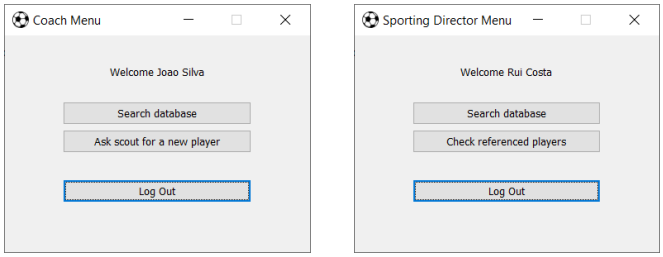

Figure 24: Coach and sporting director menus

When selecting *Search database*, a window is presented to the user (Figure 25) where he can search players according to some desired constraints, namely a minimum value for a certain performance parameter, a maximum value for the age or for monetary sums that must be paid to transfer a player or directly by name, dominant foot, or position. The C++ MySQL connector implemented has a maximum query size of 256 characters, which at 8 bits per character gives a maximum size of 256 bytes for the query string, meaning only approximately 10 options can be checked at the same time, depending on the length of input parameters and the individual queries themselves. A warning pops up if the user defines a larger query than the connector allows (an alternative could have been implementing two queries, the first regarding nine check boxes and the other one regarding the other 9 check boxes, and merging both results in the same table, but the previously described solution was chosen to make the implementation simpler).

When searching for dominant foot, the user should insert **R** for right-footed, **L** for left-footed and **B** (both) for either-footed players. In terms of search by position, below are the most commonly used abbreviations used in this database, even though new positions are constantly being developed:

- **GK –** goalkeeper;
- **CB –** centre-back;
- **RB –** right back;
- **LB –** left back;
- **CDM –** centre defensive midfielder;
- **MF –** centre midfielder;
- **CAM –** centre advanced midfielder;

- **RW –** right winger;
- **LW –** left winger;
- **ST –** striker.

After selecting *Find Players*, a query is done to get all the players that have at least the specified attributes (getting all the players in the database if no attributes are specified), alongside with the respective club, agent, transfer fee and demanded wage.

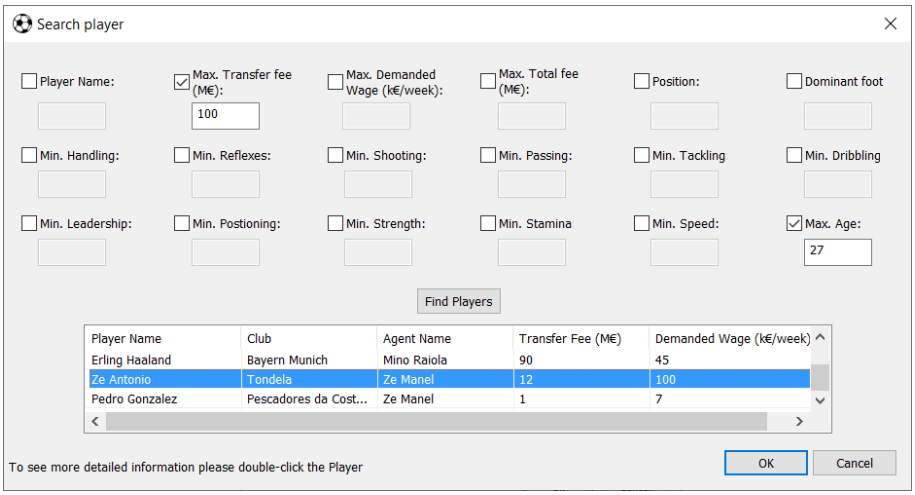

Figure 25: Search player window

Double clicking any line in the table will open another window (Figure 26), where the queries *Get Bio Info*

, *Get Scouting Report Info*

, *Check rating* **(goalkeeper, technical, mental, physical)**

and *Get Financial Info*

are done to get the player's personal information, scouting report (including the averages for each category of attributes) and financial information, as well as their agent's information. The coach may also then refer the player to the sporting director if he judges him to be a suitable transfer target, by clicking the *Refer Player to Sporting Director* button (invisible to the sporting director), that does the query *Refer Player*

, which updates the relation *refers_to*, adding a new row with the player referred, the coach that referred the player and the sporting director. As a repeated referral would lead to

a repetition in primary keys in table *refers_to*, in that case the query is invalid and duplicate rows with the same coach, sporting director and player will not be created.

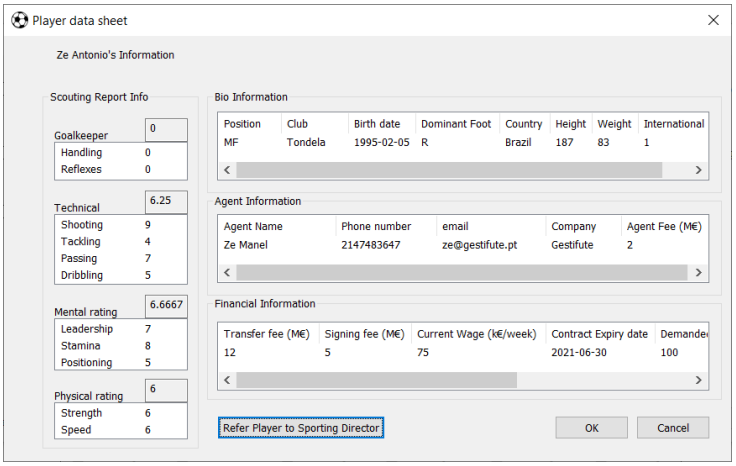

Figure 26: Player data sheet window

The sporting director can then check them by clicking *Check referenced players* in Figure 24, where the query *Check Referred Players*

is done to get all the all the players that the coach referred, as well as their agent name, demanded wage and transfer fee, shown in Figure 27. Double clicking the player name will open the same window as in the *Search player window* (Figure 26).

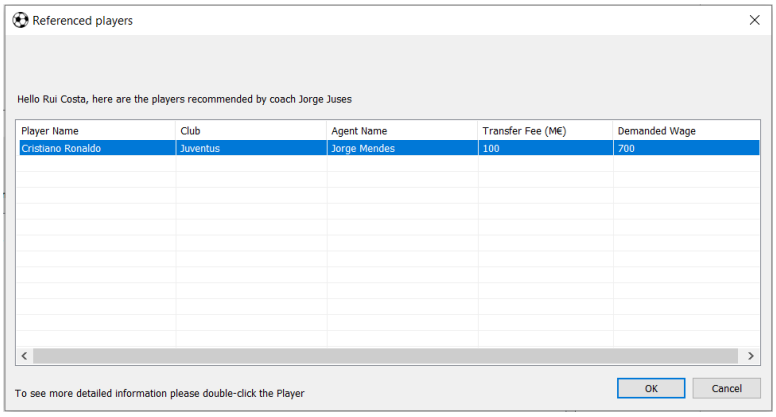

Figure 27: Referenced players window

The coach can also send a request to a scout by selecting the *Ask scout for a new player* button (Figure 24) if he does not find a player that suits his requirements in the

database. When doing so, a query is done to get all available scouts' names. The coach should then select the scout to whom he wants to send a request in the drop-down menu *Select the Scout that will receive the request* and then write the characteristics of the desired player (up to 200 characters). After clicking *Send request*, the query *Create Scouting Request*

is done, updating the relation *request*.

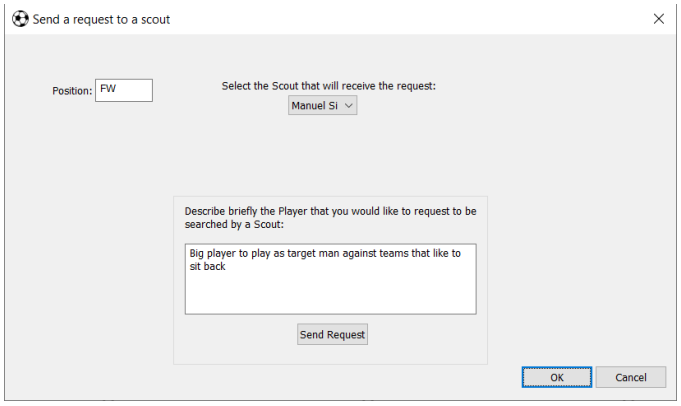

Figure 28: Request to scout creation menu

7.3. Scout/Agent Manual

In the last section, both scout and agent menus will be explained, as both users' functionalities are very similar, as both can add and edit new players, with the main exception being the fact that the scouts can also check requests from coaches. The main menus for both types of user are shown in Figure 29.

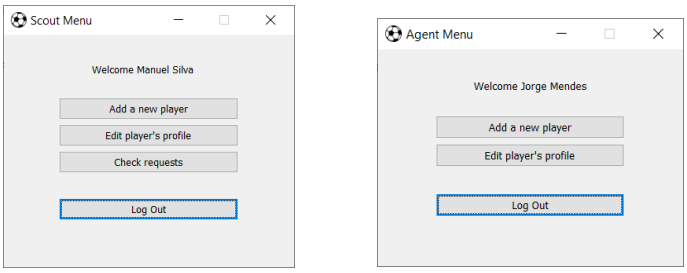

Figure 29: Scout and agent menus

When adding a new player by clicking the *Add a new player* button, the window shown in Figure 30: Window to add a new player to the database appears, where the scout/agent will fill the personal information related to the player. All fields must be filled, with *name* (unique in the database), *country* and *club* being limited to 50 characters each and *position* to 5 characters. If the player is a free agent, *club* field must be filled with 'No team'. Players must

also be at least 16 years old to be featured in the database. Position names should use the same convention as in the *Search Player* window. After clicking *Ok*, the player will be attributed to the scout/agent that added him (where a new row is added to *scout_report*/*financial_info*, depending on which one is logged, with values set at 0, apart from the *demanded_wage* and *demanded_contract_length* when the agent is adding, which are set to 1, as they cannot be 0 as it will be seen in the *Edit player* window, and a new relation is created in *watches*/*provides*), by running the queries *Add New rows*

and *Get scouting report/contract ID*

.

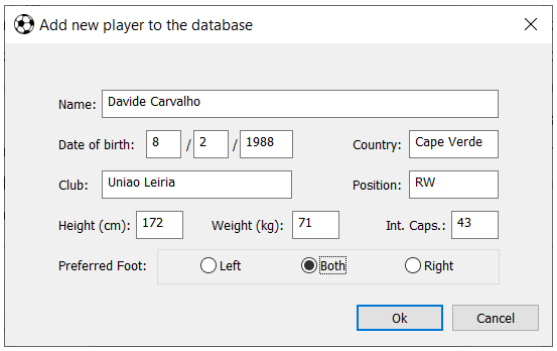

Figure 30: Window to add a new player to the database

After adding the player, the scout/agent should then choose *edit players' profile* (Figure 29) to add the player's stats/financial details, using the window in Figure 31. The scout/agent must type in the name of the player they want to edit in the field *name* and then click *Search*. After clicking search, a query is made to check if the player is related to the scout/agent (by accessing the relations *watches*/*provides*). If so, another is done get all the information (*Get Bio Info*

, *Get Scouting Report Info*

and *Get Financial Info*

) of the player and updated into the respective boxes, so the user knows what needs to be edited. The boxes related to personal information and to the scouting report/financial info (depending on who is logged on) are also enabled. If the player is a goalkeeper, only the fields *handling*, *reflexes*, *passing* and *leadership* are enabled in the scouting report, whereas if the player is a field player, only *handling* and *reflexes* are not enabled. The *name* field is also disabled so it cannot be changed (to improve robustness, as changing it would allow to edit other players or would try to edit non-existing players). It is also important to note that if

the player does not have a scout/agent attributed (because he was added by the other), the first one to search him in the *Edit player* window will be the one assigned.

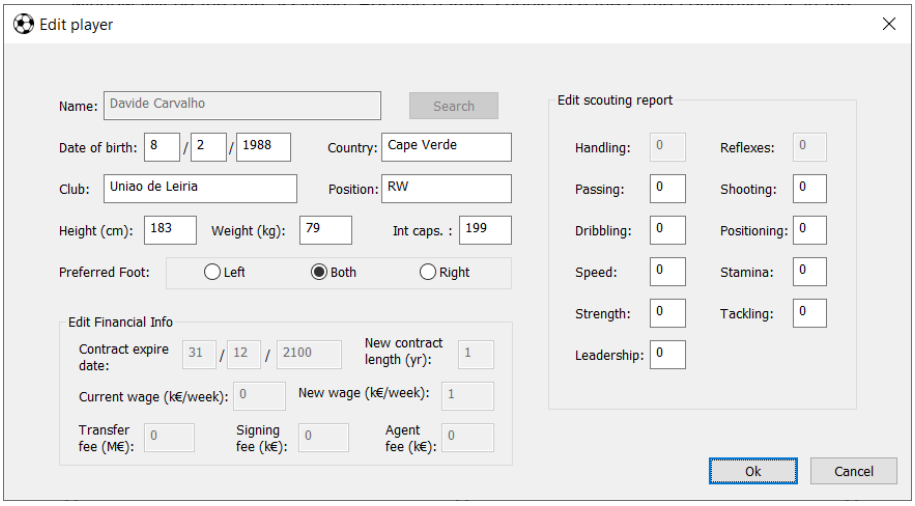

Figure 31: Window to edit an existing player to the database (via scout menu)

As for the inserted values, the fields related to the personal info function similarly to the *Add new player to the database* window. As for the scouting report, all values inserted must be integers ranging from 0 to 10. For the financial info, the player must always be demanding a certain new wage and a certain contract length, but the remaining fields can be set to 0, as they are optional. The current contract expire date must also be in the future (if the player is a free agent, today's date should be used). After clicking Ok, three procedures are called, the first to update players information, another to get the *report_id*/*contract_id* and the last one to update it.

It is also important note that, although the scout and the agent cannot search the database, they may check the complete information regarding their associated players, so the agent can check his client's performance indicators for negotiation purposes and the scout can analyse the financial viability of signing a player and decide if he wants to keep scouting the player or proceed to a new one.

The scouts also have the option to check requests in their main menu (Figure 29). When this option is chosen, a query is called to get the number of requests, presenting the scout with the resulting number (Figure 32). The scout can then click *See Requests*, where the query *Get Requests from Coach*

is done to get all requests made by coaches to the scout, presenting them, alongside the position of the desired player and the name of the coach that placed the request.

Figure 32: Window to check requests from coaches

8. Programmer Tips

8.1. MySQL (database and queries)

Stored procedures were favoured in most of the project in queries that have a clearly defined structure with only some input parameters changing with the usage (like a specific search attribute used in a WHERE statement). However, while they increase database security due to being resistant to SQL injection and deal with the problem of limited query size, they offer little versatility in changing the query structure programmatically. This means that for the highly customizable Player Search Query, it had to be implemented as a regular query, creating a weak point that may be exploited with malicious intent. The solution to this would be using a combination of Prepared Statements, another tool for database protection, but this was deemed too complex and outside of the scope of our work.

It is noted that MySQL's text encoding has difficulty dealing with letter accent (like á, õ or ç) so it is recommended that names written in languages that use them have them removed first before insertion.

Another important decision was the way to deal with the limited MySQL connector query size. This only posed a problem in the Player Search feature, since it is the only query that is both large enough to reach the maximum size and not implemented in a stored procedure (due to the required modularity of this part of the code, which would be difficult to implement in MySQL). Note that most of the query remains the same with or without adding constraint, so the common portion of it was saved in a stored procedure called *init_query* that creates a temporary table called *i*, chosen to save in character memory. All further queries are then just performed on table *i*, which needs a much lower number of characters to perform operations on. After the query, the table is dropped from the database to save memory.

Comentado [JCMT2]: Devo acrescentar codigo/imagens?

8.2. C++ (GUI)

One issue found when building MFC dialog windows is the fact that the list control boxes that were used throughout the program to display information in multiple lines and columns use as input objects from the LVITEM built-in class. When defining the function that saves the query results to a vector of objects from this class it was noted that the object saves a pointer to the data and not the data itself, meaning that when the query function is closed the program reallocates the memory to new data, turning the pointer into a dangling pointer, which leads to undefined behaviour, as seen in Figure 33.

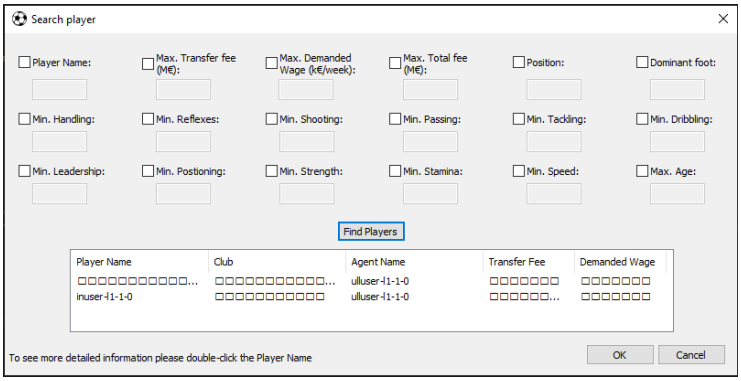

Figure 33 – Undefined behaviour caused by the dangling pointer used in the LVITEM object.

While an optimal solution to this problem was difficult to find, a workaround was developed that served the purpose of this program. A static vector is declared in the query function, which then stores all values obtained from the query, even after the function is finished. Since running the query function two times with the second result being larger than the first led to out of bounds errors on the static vector, due to it keeping its previous size, this vector must always be initialized with a large enough size as to never get a query result larger than that size. For most of our windows, a value of 1000 was deemed to be enough for the demonstration examples implemented. An example of the developed workaround is shown below.

Query for showing coach requests, using the LVITEM workaround

```
1.  vector<LVITEM> myconnectorclassDB::getrequests(CString scout_name_in)
2.  {
3.      CString query = _T("CALL getrequests('") + scout_name_in + _T("')");
4.      vector<LVITEM> requests;
5.      Query(query);
6.      int numcols = mysql_num_fields(result);
7.      static vector<CString> query_disp_value(1000);
8.      int line = 0;
9.      int value_index = 0;
10.     if (result != NULL) {
11.         while ((row = mysql_fetch_row(result)) != NULL)
12.         {
13.             for (int i = 0; i < numcols; i++) {
14.                 query_disp_value[value_index] = CPtoUnicode(row[i], 1251);
15.                 LVITEM lvi;
16.                 lvi.mask = LVIF_TEXT;
17.                 lvi.iItem = line;
18.                 lvi.iSubItem = i;
19.                 lvi.pszText = (LPTSTR)(LPCTSTR)(query_disp_value[value_index]);
20.                 requests.push_back(lvi);
21.                 value_index++;
22.             }
23.             line++;
24.         }
25.     }
26.     return requests;
```

```
    query_disp_value.clear();
}
```

The output of this function may be written in the list control table by going through its elements, using the *iItem* and *iSubItem* attributes as indexes for the line and column of the element in the table, respectively. The other issue to deal with was developing a way to open the window with detailed player information when double clicking the corresponding entry in the player list, since MFC does not have a pre-built function to detect the specific entry where the user double clicks, only when he double clicks the list itself. After some research, a solution was found by overriding the double click handler function and adding a command to generate an extra “virtual mouse click” when the user double clicks that is then possible to detect using some built-in functions of the list control class. Below the code that implemented this solution is shown, where m_PlayerList is the list control variable, PlayerName_vec is a vector storing the player names in the list, and Player_Info_Window is the new window that opens when double clicking.

Handler for the double click action

```
void SearchPlayer::OnNMDblclkPlayerlist(NMHDR* pNMHDR, LRESULT* pResult) {
    LPNMITEMACTIVATE pNMItemActivate = reinterpret_cast<LPNMITEMACTIVATE>(pNMHDR);
    DWORD dwPos = ::GetMessagePos();
    CPoint point((int)LOWORD(dwPos), (int)HIWORD(dwPos));
    m_PlayerList.ScreenToClient(&point);
    if ((int nIndex = m_PlayerList.HitTest(point)) != -1) {
        CPlayer_Info_Table Player_Info_Window;
        Player_Info_Window.m_PlayerName = PlayerName_vec[nIndex];
        Player_Info_Window.DoModal();
        return;}
    *pResult = 0; }
```

9. General Appreciation

After developing this project, we conclude the general objective was achieved: not only we were able to successfully create a database, but also an intuitive GUI that would allow users not specialized in databases to easily access it.

Due to time constraints, some ideas that were initially planned for the database had to be abandoned, such, for example, searching players by club or agent, search for players with less than 6 months left on their contract (since they may be hired without a transfer fee), automatically associate a scout to a player based on the club and league he is playing in (when added by an agent) or total cost of hiring a new player (by adding all fees plus the salary multiplied by the contract length). A graphic interface included in the application for the database administrators to more easily delete or add scouts, agents, sporting directors and coacher and delete players from the database without accessing MySQL would also be a positive addition to the project.

That said, we can say that with this project we got a good level of understanding on manipulating and creating databases in SQL language and developing related diagrams, as well as an introduction to C++ programming and MFC graphical user interfaces.

10. References

[1] Pinto, J. R. Caldas, *POBD Course notes*. IST, 2011

[2] stackoverflow.com [online] Available at: https://pt.stackoverflow.com/ [last accessed January 3rd, 2021]

[3] w3schools.com [online] Available at: https://www.w3schools.com/ [last accessed January 3rd, 2021]