
Bilkent University

Computer Science Department

# Object Oriented Software Engineering Project

*STARS League*

## Design Report

Metehan Kaya, Babanazar Gutlygeldiyev, Yılmaz Berkay Beken, Kerem Ayöz

Course Instructor: Eray TÜZÜN

April 17, 2018

This report is submitted to the GitHub in partial fulfillment of the requirements of the Object Oriented Software Engineering Project, course CS319.

# Contents


**1. Introduction**

1.1 Purpose of the system……..3
1.2 Design goals……...…3
1.2.1 Usability……..…..3
1.2.2 Reliability……..4
1.2.3 Performance……4
1.2.4 Portability…..…….…….4
1.2.5 Extensibility…..…….…….…..4
1.3 Trade Offs…….5
1.3.1 Performance vs Memory……..5
1.3.2 Usability vs Functionality……..5
1.3.3 Efficiency vs Reusability……5
1.4 Definitions, acronyms, and abbreviations……..5

**2. Software architecture**

2.1 Subsystem decomposition……6
2.2 Hardware/software mapping……7
2.3 Persistent data management……7
2.4 Access control and security……..……8
2.5 Boundary conditions……8

**3. Subsystem services**

3.1 Game Model……...……..9
3.2 Game View……...……...……10
3.3 Game Controller……..……..……11

**4. Low-level design**

4.1 Design Patterns…….……...…….……12
4.2 Final object design……...…….…….13
4.3 Object design trade-offs……...…….……37
4.4 Improvement Summary..……...…….……38
4.5 Packages……...…….…….……39

**5. Glossary & references**……...……..…..……..….40

# 1. Introduction

## 1.1 Purpose of the system

"STARS League" is a football game similar to "Football Manager". It is the simplified version of that game. User plays as a manager who can control his team by changing its tactics, swapping players, etc. Also, the user has an opportunity to view groups, elimination stages, fixture, football players, man of the match, top goal scorers. Thus,, user can do changes better for his own team in order to improve his team. The user's goal is to be the champion of the "STARS League". Usability, reliability, performance, portability and extensibility are our main concerns while developing the STARS League. We will use variety of design patterns such as Singleton and Facade. These patterns enable us to implement the game in such a way that its implementation becomes simpler, coupling between classes gets decreased, consistency of the architecture is increased and design goals become more realistic to achieve. Thus, these patterns are really essential for our program.

## 1.2 Design Goals

In this part, it is shown that what we expect and aim from the game. All of the important design goals are listed below

### 1.2.1 Usability

STARS League is expected to be an 'easy to use' game, even for the people who do not know much about soccer. Since game is simplified version of its type, STARS League is not that complicated. Help Screen is ready to help people who want to learn details about the game itself, also tooltips are available where needed. In terms of usage, users only need to use mouse and keyboard as input devices and there are certain types of mnemonic keys that increase the options of users to interact with the game. Also we believe that the listening songs will increase the attractiveness of the game.

### 1.2.2 Reliability

Reliability is important for a software program since users prefer unproblematic program pleasure. Therefore, during the process of developing the game, we will give high importance to find the bugs to prevent possible crashes. Especially, user's progress in the game should be protected to avoid any data loss. Thus, we will provide an "auto-save" option to user that enables program to save the progress after a day passed in the game calendar. Additionally, user could disable that option and and he/she may save the game explicitly.

### 1.2.3 Performance

Performance, or efficiency, is another important design goal of a software program. Therefore, we aim to implement the game such that minimum FPS (frames per second) of the program will be larger than 60. Also unlike the other games, the setup time of the new game in Stars League is 2-4 seconds after team selection is done, which does not bother user to wait for it to setup a new game like other games.

### 1.2.4 Portability

Portability, or adaptability, is one of the most important aspects of a software program since a portability issue can make narrow the range of a software's usability for users. Instead of C, C++ or any other language, we are going to implement the game in Java, and thanks to the JVM which provides cross-platform portability, the game will not have such a problem.

### 1.2.5 Extensibility

Extensibility is a feature that can make a program more preferable from the developer's perspective. We tried to implement the most fundamental features of a soccer game to extend the requirements easily. For instance implementing a dynamic calendar system enables us to add many features to game such as transfer system, training system, pre-match conferences, news feed system etc. Addition of these features become easier with the calendar system otherwise game flow would only contain matches and extensions become difficult.

## 1.3 Trade Offs

### 1.3.1 Performance vs Memory

In the STARS League, we designed the database of the game in such a way that it is used only in the boundary conditions. This increases the performance of the game however, decreases the main memory usage of the game.

### 1.3.2 Development Time vs Functionality

In the limited development time of the game, we are not able to implement all the desired features of that kind of game. For instance adding a multiplayer option would take a lot of time for us to implement. Thus, in that limited time, we could only implement the fundamental features of that type of games.

### 1.3.3 Ease of Use vs Functionality

STARS League is not complicated as other similar type of games to increase the usability of the game. Many detailed features are removed from model classes because as we all experience in similar kind of games, the time-consuming details are making the game less enjoyable many times. For instance optimizing the training schedules of all players in team would be boring in that type of tournament. Thus, increasing the functionality might lower the ease of use in that game.

## 1.4 Definitions, acronyms, and abbreviations

[1] Java Virtual Machine (JVM): Java Virtual Machine which is an engine that can run Java programs. So that, operation systems that contain JVM can start Java applications.

[2] Graphical User Interface (GUI): A software that works at the point of contact (interface) between a computer and its user, and which employs graphic elements (dialog boxes, icons, menus, scroll bars, etc.) instead of text characters to let the user give commands to the computer or to manipulate what is on the screen.

[3] Cross Platform: Cross platform refers to ability of software to run in same way on different platforms such as Microsoft Windows, Linux and Mac OS X.

# 2. Software architecture

## 2.1. Subsystem Decomposition

In this section, the system will be decomposed into subsystems. During the decomposition of the system, we have decided to choose MVC (Model View Controller) as the architectural pattern which is commonly used for developing user interfaces that divides our application into three interconnected parts called Model, View, Controller.

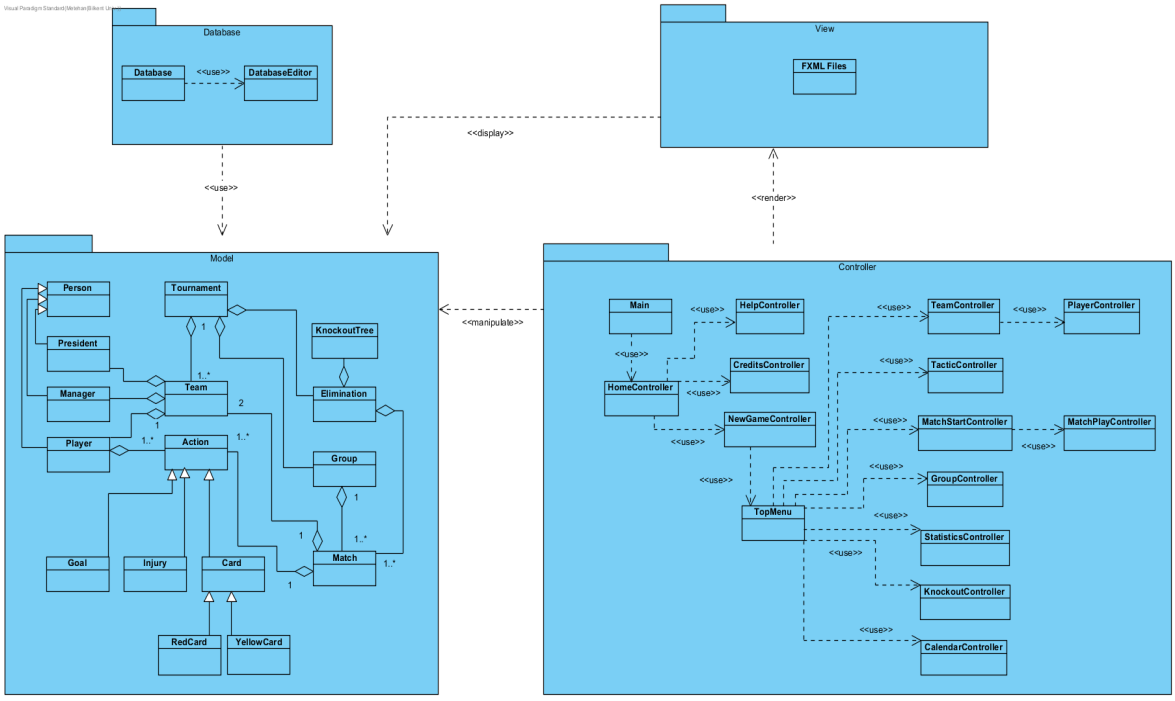



*Figure 2.1.1 Subsystem Decomposition*

Model part will include classes that manage data, logic, and rules of the game. For instance, we will have Database class that uses text file to manage data of the game.

"Game Model" subsystem consists of "Model" and "Database" packages. In "Game Model", we have classes Database, DatabaseEditor, Tournament, Team, KnockoutTree, Elimination, Group, Match, Person, President, Manager, Player, Action, Goal, Injury, Card, RedCard, and YellowCard.

We use Singleton which is a design pattern that is used to restrict instantiation of a class to one object with a single instance. Tournament class uses this design pattern. There are three reasons why singleton is used in the project. First, we avoid multiple instances of tournament class. Otherwise, there would be data inconsistency in "Model". Second reason is that this design pattern makes the usage of database easy. The most important

reason is that interaction between "Model" and "Controller" becomes simple, so that "Controller" interacts with "Model" with ease without any problem.

Another design pattern that we use is Façade that allows us to reduce dependencies between classes. Façade design pattern used in Tournament class, and by using this design pattern, coupling between "Game Model" and "Game Controller" is reduced. Instead of having many dependencies and complexity, Façade is used to avoid this problem.

View part consists of FXML files. We will utilize JavaFX platform to visualize these components of "STARS League". These will be discussed further later.

Controller part is the part which provides coherence along the software. Controller accomplishes that by administering communication among classes and objects in "STARS League". For example, this part will accept, and validate if needed, inputs from the user and forward them to related objects.

"Game Controller" subsystem consists of only "Controller" package. In "Game Controller", we have classes Main, HomeController, HelpController, CreditsController, NewGameController, TopMenu, TeamController, PlayerController, TacticController, MatchStartController, MatchPlayController, GroupController, StatisticsController, KnockoutController, and CalendarController.

## 2.2 Hardware/software mapping

The "STARS League" will be implemented using Java programming language. So the user's device should support Java and should have Java Virtual Machine. Moreover, game utilizes keyboard and mouse as input device. Keyboard will be used for text input and mnemonics while mouse is the main device to play the game. For output, screen and speaker are the devices needed to play the game. Screen should have at least 1400x900 resolution to play the game. Also listening music feature would require a speaker or headphone.

## 2.3 Persistent data management

We did not feel the need to utilize complex data management systems like SQL or similar databases because STARS League is an offline single player game. We will store the current progress data of the user in text files. To accomplish that we used Serializable interface of Java that helps us to convert Java objects into binary text files.

ObjectInputStream and ObjectOutputStream libraries, which implements Serializable interface, let us easily implement the text based database system.

Using Singleton design pattern makes implementation simpler since only the Singleton object-which is Tournament object in Stars League- is written to database, it contains all the information that is needed to load the past progress of the user.

Also we decided to access database only in the beginning and end of the game. After reading the previous game data, we modify it in the main memory without writing it to the text file immediately. When user decides to save game, program will write the game data to the non-volatile memory. This idea avoids the speed deficit of the text based database system. Thus, offline gameplay, ease of implementation and choice of design patterns are the main reasons of using the text based database in the STARS League.

## 2.4 Access control and security

Since "STARS League" will not use any databases other than text files and will not connect to internet, there is no need to apply security measures. Moreover, since there will be only one user we also do not need any access to controls. This results that the user cannot continue the saved game in another computer.

## 2.5 Boundary conditions

"STARS League" can be run from executable exe file to the computer as well as can be directly launched from jar file. Then, if it is run from exe file it will require permission from operating system. The game can be terminated by clicking the "X" button or clicking exit button from the home screen or top menu. After that it will ask for confirmation. If confirmation is given it will save the game by writing to the text file. If there is any exceptional error occurs it will try to ignore that error and launch with that error. For example, if it cannot find an I/O, it will launch without terminating the application. However, an exceptional error related to database occurs it prompts an error message and requires the exception to be handled. For example, when the user tries to load a game while there is no saved game yet, it will prompt a frame with information about an error.

# 3. Subsystem services

In our software, we choose to apply MVC(Model-View-Controller) design pattern so our software contains these subsystems. Also these subsystems contains different packages.

## 3.1 Game Model Subsystem

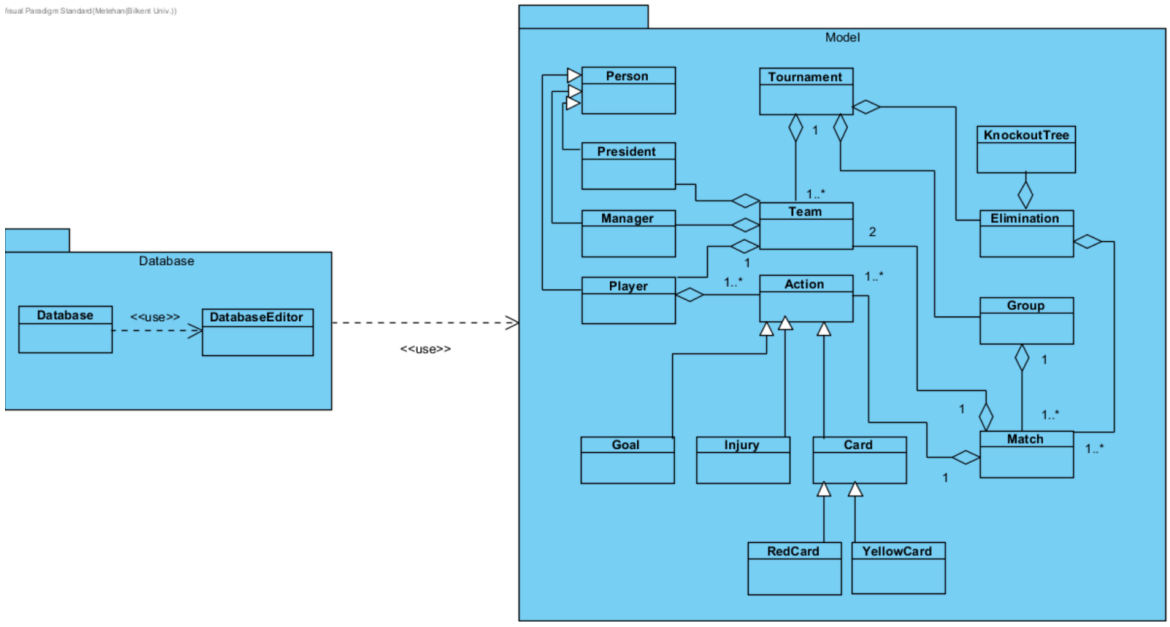



*Figure 3.1.1 Model Subsystem*

By this diagram it is easy to understand the designing of our STARS League game. Game Model subsystem consists of the model classes of the game and database connection classes. At the model class, there are many classes for processing the game. Firstly at team class it gets players, manager and president from the person class. Tournament class consists of the team class, group stage class and elimination stage class. Thus,, according to our project tournament class is a Facade class since we are using subsystems to processing the game. The information of the all teams, all goals, all assists, all cards and all statistics are stored in the database package. These informations are using by group class, elimination class, team class and person classes. Thus, the processing of the game is providing by the subsystems. After every match database is updating by the game model subsystem if the user chooses open the auto save feature. In addition, Tournament class is used as a Singleton class since it gives opportunities for using informations which are taking places in the tournament class, since the control classes avoiding the existence

of multiple instances of a tournament object. Action class consists of goal class, injury class and card class. The actions which happened during the match determining the match results of the STARS League. In addition according to match results group stage and elimination stages are determined. On the other hand with actions happened player statistics are changing. To sum up, with model classes, information which are stored in the database are used for processing the game. With the database subsystem the informations can be also loaded by the system. Thus, the game can continue from the last saved document. It also help the update the statistics and standings after every match.

## 3.2 Game View Subsystem

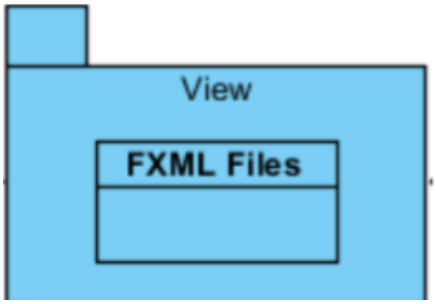



*Figure 3.2.1 View Subsystem*

Game View subsystem is provided by just .fxml files. Game View Subsystem provides a layout for GUI elements. These GUI elements consist of buttons, combo boxes, radio buttons, links and the other GUI elements. According to selection of controller classes, layouts would be updated. In game view subsystem, there are just .fxml files so it is just for the layouts. In addition, there are also images and pictures for the layouts of the game and they can be displayed on view subsystem as we done it for our game logo. In the control package, there are classes for frames to changing the states of the views, interact with the user by taking inputs. These classes also communicates with the View subsystem to change the views properly.

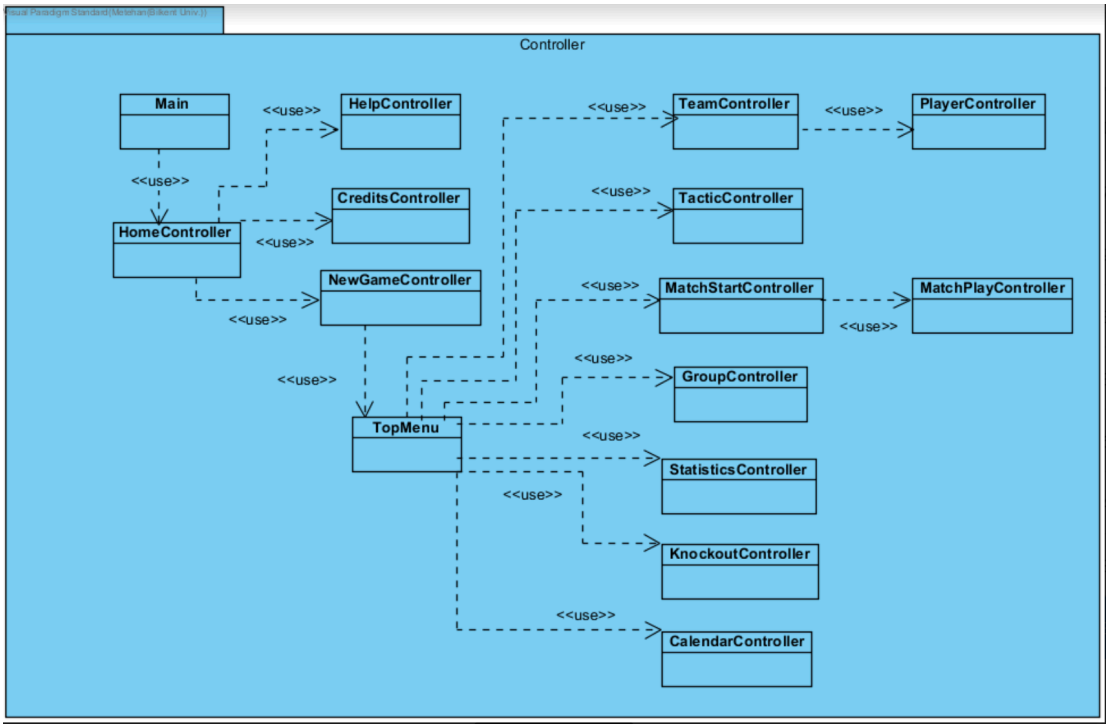



*Figure 3.3.1 Controller Subsystem*

## 3.3 Game Controller Subsystem

Game Controller subsystem consist of two packages. These packages are main and controller packages. Main package is being used for connecting the controller classes to other subsystems. Thus, Game Controller Subsystem is using as an interface in our system. In the control package, there are classes for changing the states of the frames and layouts by interacting with the user by taking inputs to these controller classes. These classes also communicates with the View subsystem to change the views properly. To sum up, according to user's selections at the controller classes the system displays the view of the game to the user.

# 4. Low-level design

## 4.1 Design Patterns

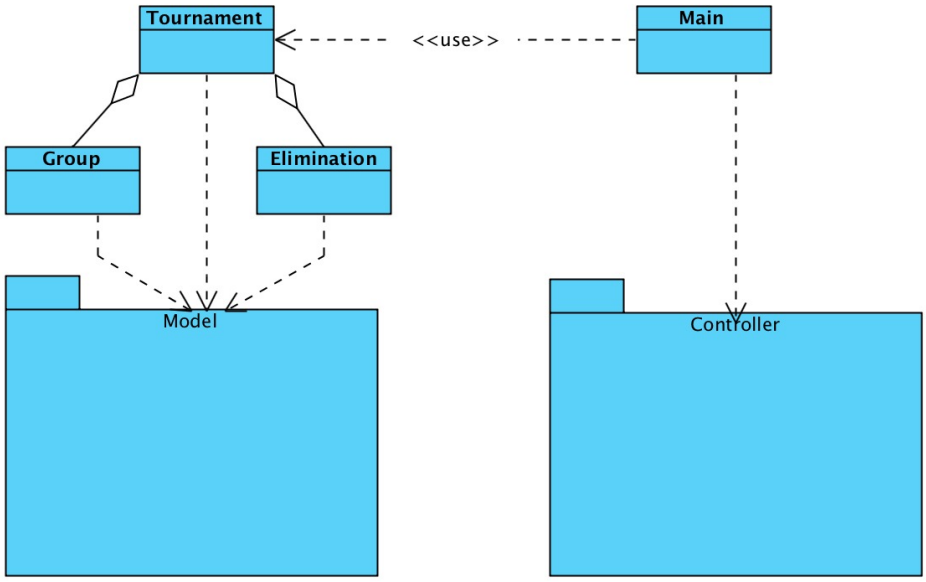



*Figure 4.1.1 Façade Design Pattern Demonstration*

Façade and Singleton are used as design patterns. Tournament is a Singleton class that provides data consistency. If Singleton was not used, we could have faced with some data inconsistency problems because having more than one object of Tournament class may lead to update of objects owned by different Tournament objects. This kind of updates can cause a problem during a query process.

Main, Tournament, Group and Elimination classes are Façade classes that reduce the complexity between different subsystems. This relation is between the "GameModel" and "GameController" subsystems. Instead of making many connections between the "GameController" subsystem and all classes of the "GameModel" subsystem, three classes which are Tournament, Group, and Elimination used as Façade classes. Tournament class handles with most of the connections in this manner. In addition to Tournament class, Group and Elimination classes take responsibility to reduce the complexity. Also, Main is another Façade class that prevents coupling with the "GameController" subsystem. It's a Façade class that is related to GUI.

# 4.2. Final Object Design

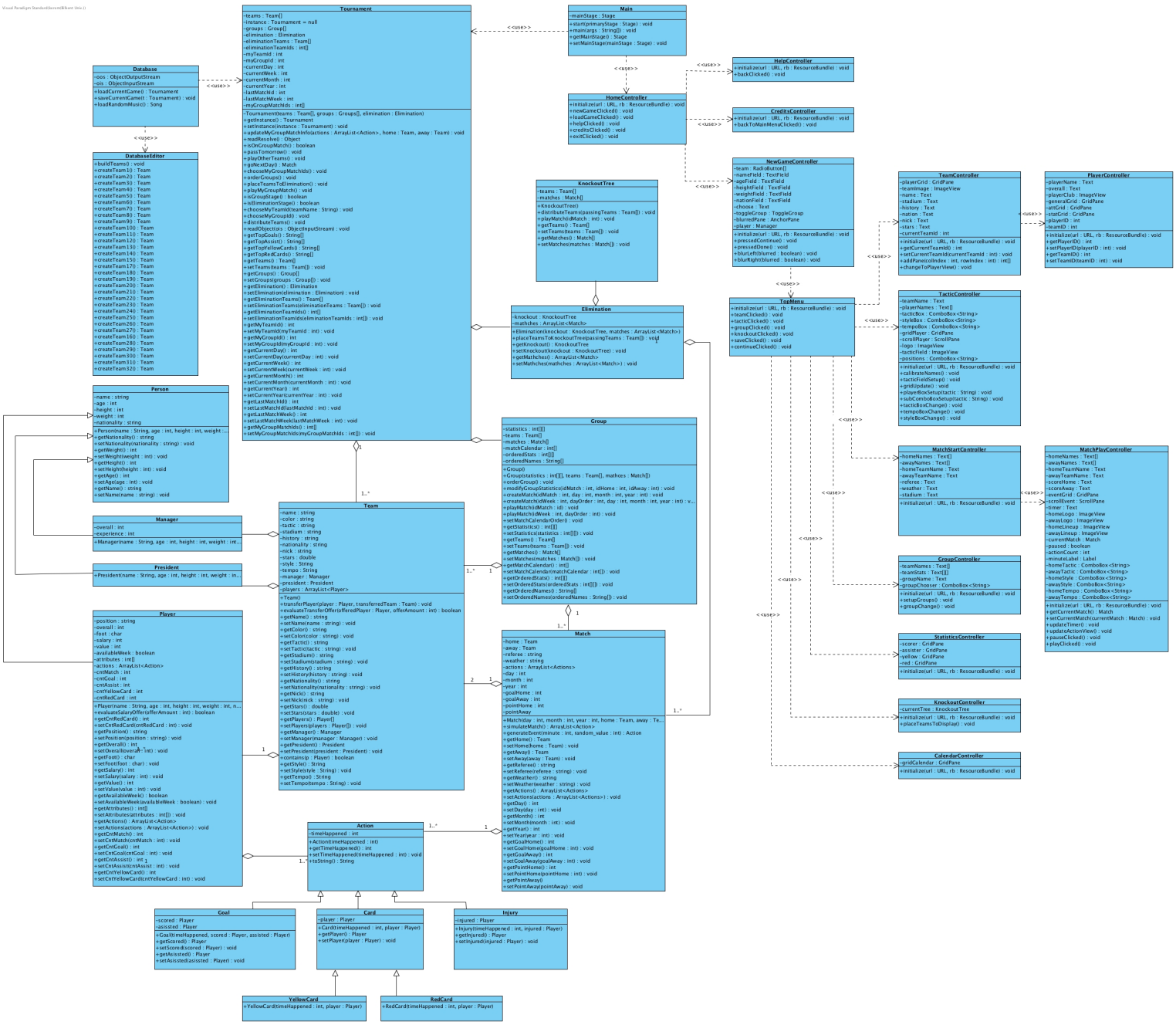



*Figure 4.2 Complete UML Class Diagram*

## Tournament Class

Tournament class represents the STARS League itself. It contains the 2 stages of the STARS League. These stages are group and elimination. By the starting of the game, there would be a draw for groups and teams will go to groups randomly. After that, in the group stage, there are 8 groups with 4 teams per one group and two teams who finished their groups in top 2 will qualify from their groups and will participate to elimination stage of the tournament. The qualified teams will be drawn by the system for every stage of the elimination. Semi Final draw will be last draw. During the elimination stage teams will play with their opponent 2 times. These matches would be in home and away for every team. If a team loses against his/her opponent the game will be over for this team and this manager will have chance to start a new game. If a team wins all elimination matches, his/her team will win the STARS League cup! Thus,, game will over.

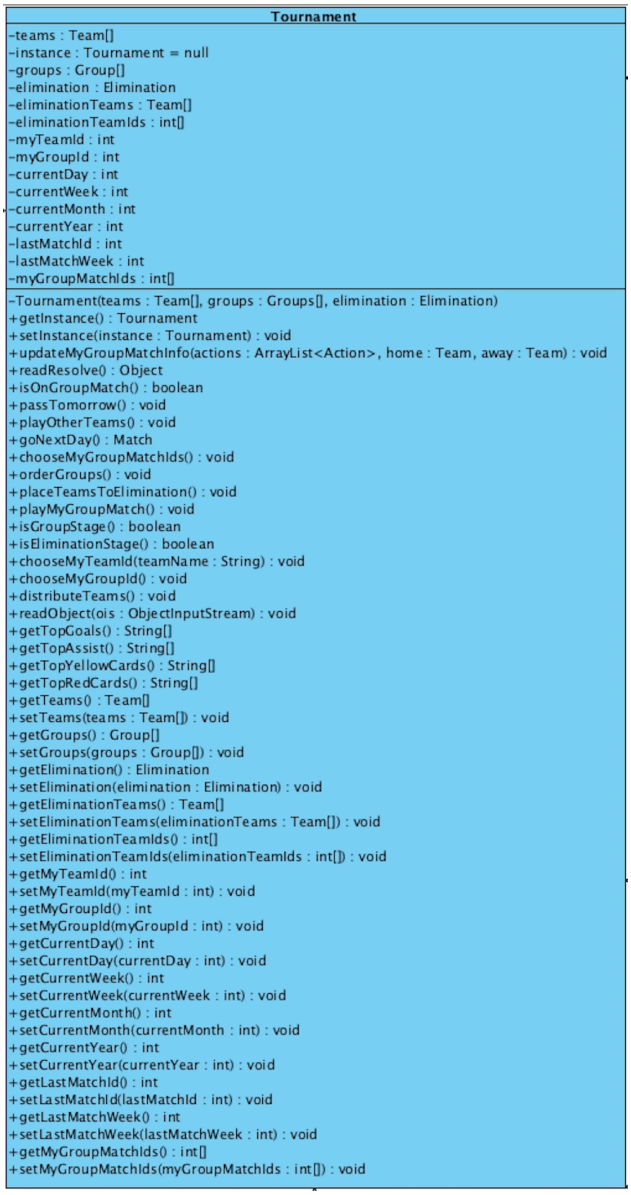



*Figure 4.2.1 Tournament class*

After the matches the date will processing for next week and next match. That's why, tournament class has current day, current week, current month and current year. These variables also provide the schedule. Schedule will be displayed according to these variables. With "lastMatchId", "myGroupMatchIds" variables, scores and statistics of the matches will be stored at group standing table, elimination tree and statistics. Thus,, user will have chance to get very important information from group standing table, elimination tree and statistics. These variables will be updated after all matches. After the matches and scores are obtained, schedule will also be updated with matches' scores. Also, it has the list of teams that are in the tournament.

Having these instances of objects enables the Tournament class to be a 'Facade Class' in the Facade design pattern. It is the interface of model for interacting with the controller and view subsystems. Also, all the information about tournament could be accessed from the Tournament object, so storing only Tournament object in database is enough for our program to load the previous game that is saved.

Also, all the other controller classes interact with the Tournament instance, therefore there should be only one Tournament instance during the program is running. Therefore, we decided to apply 'Singleton' design pattern for tournament. This Singleton instance is created at the beginning of the program-either loaded from database or created newly- and used by other controller classes.

**Methods:**

- *getInstance()*: This method is called when the user clicks on "New Game" or "Load Game". It fills the variables with corresponding values by using the txt file.
- *isOnGroupMatch()*: Checks if there is a match of the user's team on the current day.
- *updateMyGroupMatchInfo()*: Players' statistics are updated according to actions happened during the user's team's group match.
- *passTomorrow():* Handles with overflows. For instance, if the current day is the last day of the current month, new date is the first day of the next month.
- *playOtherTeams()*: Matches between the teams where none of them is the user's team are played, and results of them are updated.
- *goNextDay()*: It skips to next day. If there is any match, they are played and results of them are updated.
- *chooseMyGroupMatchIds()*: At the beginning of the game, decides ids of all group matches which are played by user's team.
- *orderGroups()*: After a group match is played, this method should be called to order the group according to teams' points and goals.
- *placeTeamsToElimination()*: After group stage is finished, this method is called to put top 16 teams into elimination (knockout) stage. From each group, top 2 teams are passed to elimination stage while the others are eliminated from the tournament.
- *playMyGroupMatch()*: Next group match of the user's team is played. At the end of the match, necessary updates are handled.
- *isGroupStage()*: Returns true if the game is in group stage. It checks the date to understand this.
- *isEliminationStage()*: Returns true if the game is in elimination (knockout) stage. It checks the date to understand this.

- *chooseMyTeamId()*: Looks for all teams of the STARS League, and returns the id of the team that user choose at the beginning of the game.
- *chooseMyGroupId()*: Looks for all groups of the STARS League, and returns the id of the group that user's team is in.
- *distributeTeams()*: At the beginning of the game, gets all of the teams, shuffles their order, and then puts them into 8 groups where each group has 4 teams.
- *getTopGoals()*: The game enables the user to see top 5 scorers. When the user wants to see corresponding panel, this method is called to get top 5 scorers. This method looks for all players of the STARS League. Then it modifies the top 5 table by doing some simple comparisons. In the end, it returns string version of the top 5 table.
- *getTopAssists()*: The game enables the user to see top 5 assisters. When the user wants to see corresponding panel, this method is called to get top 5 assisters. This method works similarly to getTopGoals().
- *getTopYellowCards()*: The game enables the user to see top 5 players with most yellow cards. When the user wants to see corresponding panel, this method is called to get top 5 players. This method works similarly to getTopGoals().
- *getTopRedCards()*: The game enables the user to see top 5 players with most red cards. When the user wants to see corresponding panel, this method is called to get top 5 players. This method works similarly to getTopGoals().

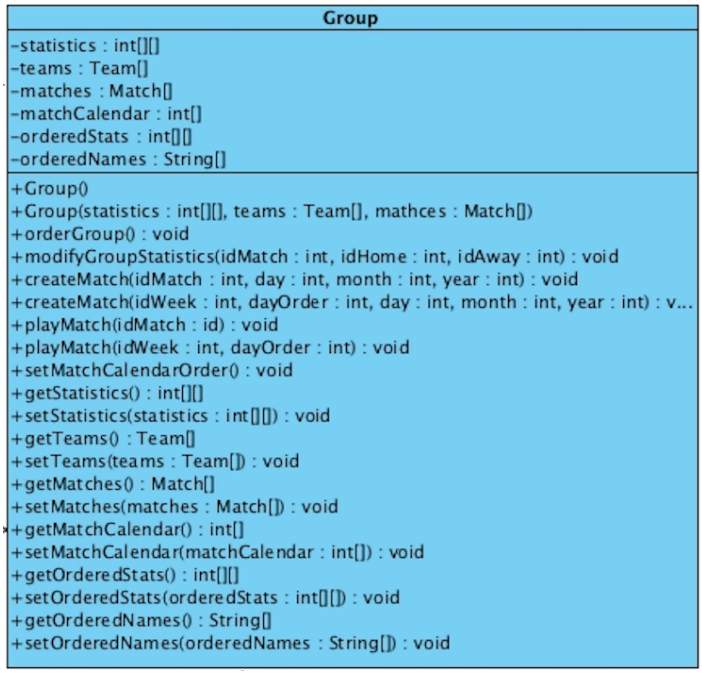



*Figure 4.2.2 Group Class*

## Group Class

The "Group" class stores the statistics of the tournament. The statistics consist of the number of matches that is currently played, goals scored in these matches, goals conceded in these matches and points gained from these matches for every team in that group. When a group is displayed, teams are ordered according to their points and goals. That's why variables "orderedStats" and "orderedNames" are necessary. Groups will be consisted of 4 teams and there will be 8 groups at the tournament. 2 teams of every group which finished group stage first and second will be qualified to elimination stage. GroupStage has Match instances to store the matches that is played between that group members.

**Methods:**

- *orderGroup()*: Teams are ordered according to points they collected, and goals they scored.
- *modifyGroupStatistics()*: Responsible method to update group statistics including total matches played, total wins, total draws, total losses, total goals scored, total goals conceded, total points.
- *createMatch()*: A "Match" object is created according to the date.
- *playMatch()*: A given group match is played, then group statistics are updated.
- *setMatchCalendarOrder()*: For each week, it decides the time of matches. Since each pair of teams play with each other twice, a team plays 6 group matches. Therefore, in total, there are 12 group matches for each group.

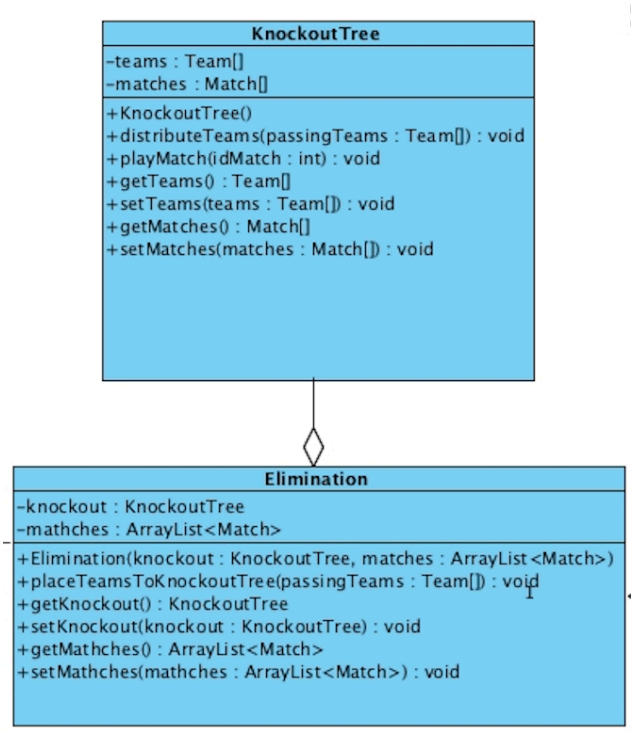



*Figure 4.2.3 Elimination and KnockoutTree Classes*

## Elimination Class

The "elimination" class stores the teams who qualified to elimination stage from group stages. For qualification to the elimination stage, teams have to be in top 2 in their own groups. This class consist of a binary tree for maintaining the tournament.

**Methods:**

- *placeTeamsToKnockoutTree()*: calls KnockoutTree's distributeTeams() method to distribute teams randomly into the tree.

## KnockoutTree Class

The "KnockoutTree" class consist of a binary tree to maintain the tournament. This tree represents the current stage of the knockout. Nodes of this tree are teams and the teams that have the same parent are rivals to each other. Also, the match results are stored in that nodes.

**Methods:**

- *distributeTeams()*: 16 teams which are qualified to elimination stage are randomly shuffled and put into tree.
- *playMatch()*: A given knockout match is played. If it's the second match that played between the teams, the winner is decided according to these 2 matches. Then this team is qualified to next stage. For instance, if it was the semi final match, then this team will qualified to final match. Some updates are done on the tree, in this case.

## Team Class

Team class is responsible for storing the teams' features. Each team has a "name". Colors of teams are displayed, that's why there is a variable "color" for all teams. Each team has to have a "tactic" (formation) for the incoming matches. Each team has to have a "stadium" for playing their own matches at their home. There should be information about all teams' cups, championships, records at "history". Each team has a "nationality" because STARS League is an international tournament and participants of the tournament are from different countries. Each team has a "nick" which is determined by their fans (Galatasaray - Cimbom, Manchester United - Red Devils, etc.).


*Figure 4.2.4 Team class*

Team has a "manager" who controls the team's actions (transfers, matches, squad selections, substitutions). There would be information of the manager which are filled from manager at the starting of the game. Team has a "president", to show the personal information of the president of the team. Each team in STARS League has 20 players at the beginning of the game, but the number of players can change after transfers. Also the first 11 players of these 20 players will be the starting players in the next match of the team. On the other hand, user can display these teams' players' ratings, statistics, personal information to get information. Thus,, the possibility of the success of the user will be increased since there is a strong relation between the success and knowledge.

Every team would have "stars" which represents to team's powers according to their players' ratings. Evert team has a "style" for the incoming matches (ultra-attacking, defensive, normal etc.). Styles of the teams will affect their match results. For instance, if a team's style is defensive there would be less goals than ultra-attacking tactic). Each team is affecting by their previous matches results this represents at "tempo" variable and that will affect team's incoming matches.

**Methods:**

- *contains(Player p):* Checks if the team has the given player "p". Returns true, if the team has that player, or false.

### Match Class

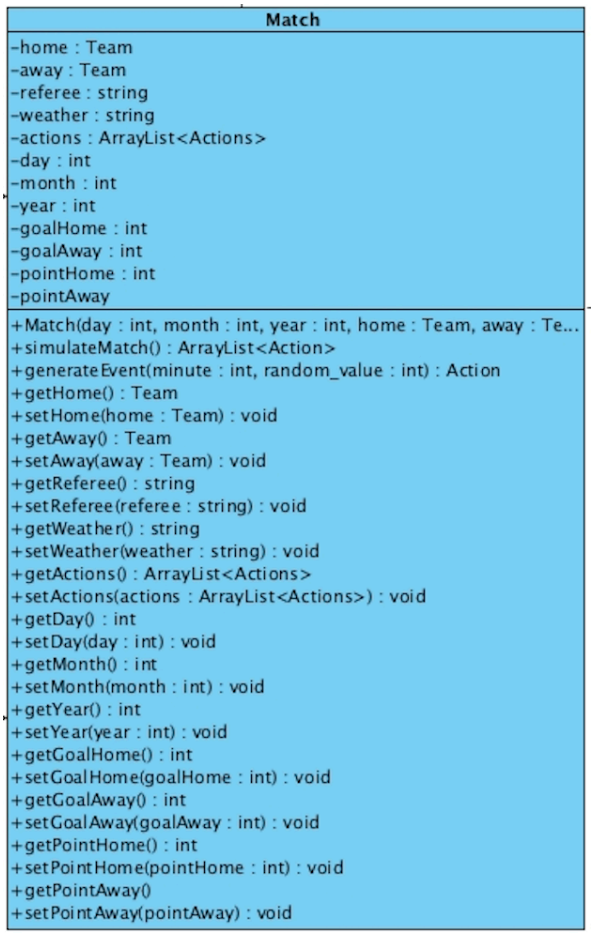



*Figure 4.2.5 Match class*

The "Match" class stores the home team by a variable "home". It also stores the away team by a variable "away". It stores the match's "referee", and "weather" of the match-day at the stadium. These are displayed to the user. Date of the match-day is important for match, and it is stored by 3 variables named as "day", "month", and "year" which are displayed in the form "mm/dd/yyyy". Date has an effect on the weather like in winter dates, the possibility of the snowing will be increased and that will also affect the matches score.

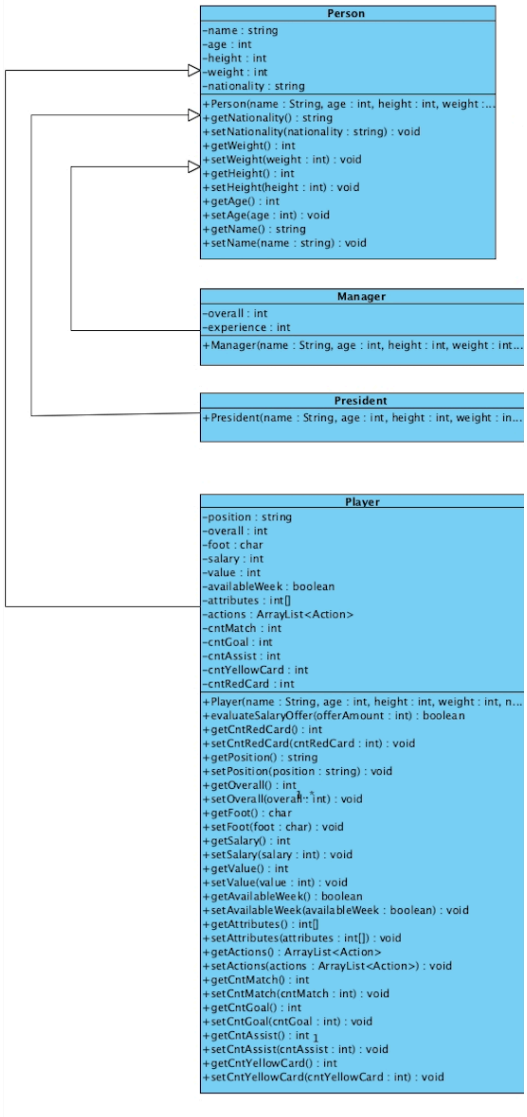
The number of goals that scored during the match, decides the values of variables named "goalHome", and "goalAway". Value of the "goalHome" is equal to the number of goals scored by home team, while value of the "goalAway" is equal to the number of goals scored by away team. These values play a part in deciding the values of "pointHome" and "pointAway". If home team scores more goals than away team, then home team wins the match, in other words away team losses the match, and "pointHome" becomes 3, where "pointAway" is 0. If away team scores more goals than home team, then away team wins the match, in other words home team losses the match, and "pointAway" becomes 3, where "pointHome" is 0. If none of the teams scores more than another one, the result of the match is draw, and both "pointHome" and "pointAway" become 1.

What "Match" class controls directly do not differ in the various stages of the tournament. The same pattern is used in both knockout matches and group stage matches. However, "Group" and "KnockoutTree" classes use matches in different ways, obviously. After a group match is played, status of corresponding group's table is updated according to "pointHome", "pointAway", "goalHome", and "goalAway". After a knockout match is played, its results are updated. If it is the second match that is played between these teams, most successful one is passed to next stage, where the other one is eliminated. Success is determined by total points collected, and goals scored by teams.

**Methods:**

- *simulateMatch()*: A match lasts for 90 minutes. At each minute, there is a possibility of generation of an action. Features of players, managers, tactics, status of recent matches of the teams playing the match currently, weather condition and random generator play a part in deciding the kind of the event. In the end of the method, players' statistics, only *"cntMatch"* which shows the number of matches played, are updated.
- *generateEvent(int minute, int value)*: "minute" shows the time when the event is happened. "value" determines the kind of the event. Possible events are goal, any kind of card, and injury. If the event is not an injury, the player's statistics are updated. If it is a goal, then also assister's statistics are updated.

## Person Class




The "Person" class stores the properties of a person. Every person has a name, age, height, weight and nationality. There are different types of person. A person can be either a player, manager or president.

## Player Class

The "Player" class stores the information of a player. Player has need to have position(s) that's why, there is a variable named "position. Player has a preferred foot by a variable name "foot". Player has a wage information which stores at a variable named "salary". Player has a value which is determined by player's "attributes" and "overall" and it's stored at a variable named "value". Player can get injured or suspended so they can't play for the incoming match, that's why, there is a variable named "availableWeek" which shows the availability of the player for incoming match.

Also, a "Player" object has some variables named as *"cntMatch", "cntGoal", "cntAssist", "cntYellowCard", "cntRedCard"* which are decided by the matches that the player plays, and actions that the he plays a part in. There is a panel that shows top 5 scorers, assisters, players that got most red and yellow cards. Existence of these variables is necessary to display this panel.

*Figure 4.2.6 Person, Player, Manager, President classes*

## Manager Class

The "Manager" class stores the manager's attributes total by "overall". That overall has a role in the team's performance which is affected by the manager significantly. A "Manager" object has a variable named "experience" which shows the number of experienced years, and it is affecting manager's overall critically.

## President Class

The "President" class does not store anything more than "Person" class stores. This class is used when information of the teams are displayed.

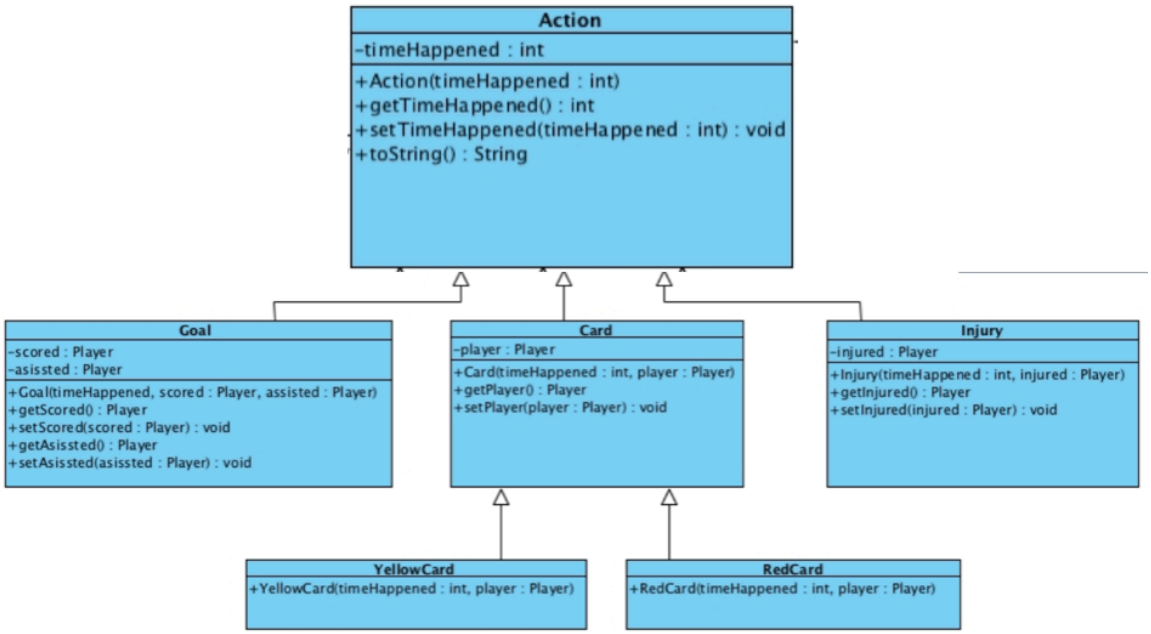



*Figure 4.2.7 Action class hierarchy*

## Action Class

An object of "Action" class in generated when an event occurs during the match. There are different kinds of action. An action can be a goal, card (red or yellow), and injury. In common, all actions have a variable named "timeHappened" which indicates when the action happened. Features of players, managers, tactics, status of recent matches of the teams playing the match currently, weather condition and random generator play a part in deciding the kind of the event.

### Goal Class

An object of "Goal" class is generated when a player scores a goal during the match. 2 players have a role in scoring a goal. These are "scored" which is the player who scored the goal, and "assisted" which is the player who gave a pass to the scorer before the goal.

### Card Class

An object of "Card" class is generated when a player gets a card during the match. Variable "player" is the player who got the card. There are 2 types of card. These are red card, and yellow card.

### RedCard Class

An object of "RedCard" class is generated when a player gets a red card during the match. Getting a red card causes a player to be dismissed from the field. A dismissed player cannot be replaced; their team is required to play the remainder of the game with one fewer player.

### YellowCard Class

An object of "YellowCard" class is generated when a player gets a yellow card during the match. Getting two yellow cards result in a red card. Therefore, the manager can decide to swap the player with another player who is not in the field.

### Injury Class

An object of "Injury" class is generated when a player gets injured. Variable "injured" is the player who got injured. If this injury is resulted from a foul, this action can be generated with another action which is a card that is given to the player who made the foul. After an injury, the player cannot play efficiently. Therefore, the manager can decide to swap the player with another player who is not in the field. Also, the player cannot play in the next week which is shown by a variable named "availableWeek" that is owned by the injured player.

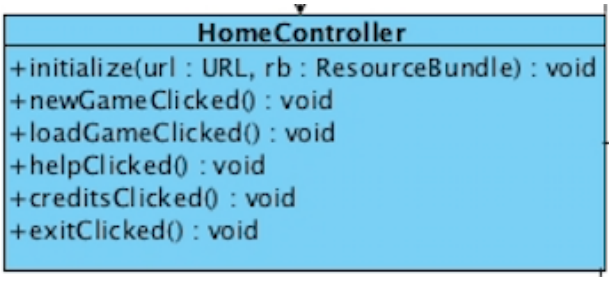



*Figure 4.2.8 HomeController class*

## HomeController

This is the controller class of the view which the user sees at first opening of the game. The application starts and goes on from this controller according to preference of the user.

### Methods:

-*newGameClicked()*: When the user prefers to start a new game, New Game button must be pressed. This click will call newGameClicked() method will be called. On the call, this method will load *NewGameView.fxml* to the parent scene and stage will be set to that scene.

-*loadGameClicked()*: This is a method initiated when the user has already saved game and wants to load that game. Since the user already has saved game and preferred team it will load his/her team's view to the parent scene.

-*helpClicked()*: If the user clicks Help button on the Home Screen, then it will require a help which means we need to provide help. To do that, the method will load *HelpView.fxml* to the parent scene.

-*creditsClicked()*: When Credits button is clicked it will call the creditsClicked() method. Then this will display *CreditsView.fxml* by loading it to the parent scene.

-*exitClicked()*: If the user clicks Exit button, then the application will call exit method of the system and exit the application.

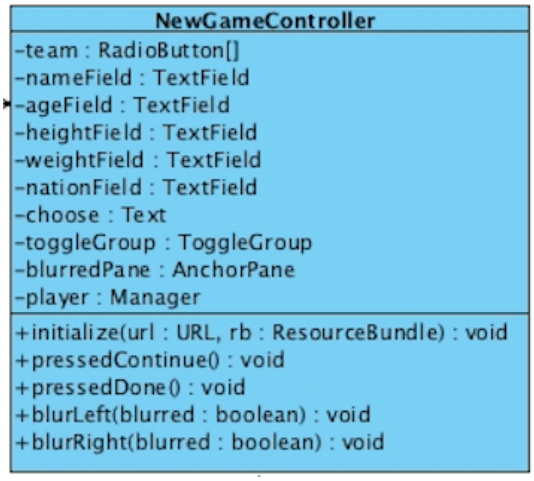



*Figure 4.2.9 NewGameController*

## NewGameController

This is the controller of the New Game View. It will be initiated if the user wants to start a new game.

### Methods:

-*initialize()*: Here, all 32 teams will toggled for the user to make a choice. The user will be asked to make choice among these teams. Otherwise it is not allowed to go on.

-*pressedContinue()*: If the user presses Continue button after filling all fields, it will be asked to choose his/her team. Manager of the team will be assigned as the user in this method.

-*pressedDone()*: This method is called after the user presses done button. Selected teams and every other informations are set in the method.
+blurLeft(): This method is in action in the *pressedContinue()* method. It will blur the color of the left side to give visual effects.

-*blurRight()*: This method is to blur right side of the screen, which is intended for the user to fill the related fields of personal information.

### Attributes:

-*team[]*: These are radio buttons for all of the teams.
-*nameField*: This the field for the name of the user.
-*ageField:* The user is supposed to write age of his/her in this field.
-*heightField*: It is for the height of the user.
-*weightField*: Weight of the user will be written here.

-*nationField*: This field id for the nationality of the user.

-*choose*: This is the text that informs the user to choose the team.

-*toggleGroup*: This is toggle group for the team names.

-*blurredPane*: This is blurred section of screen after continue button is clicked.

-*manager*: This is the manager of the team which is assigned as the user.

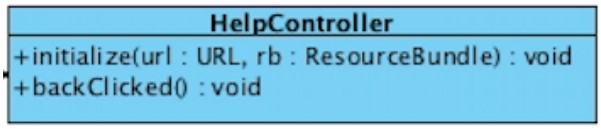



*Figure 4.2.10 HelpController class*

## HelpController

This class has view class which provides help for the users who need one. So it will be controlling *HelpView.fxml* class.

### Methods:

-*backClicked()*: This is a method for the Back button in the help view. It will take the user to the home screen by loading *HomeScreenView.fxml* to the parent stage.


*Figure 4.2.11 CreditsController class*

## CreditsController

This class controls *CreditsView.fxml* which displays credits of the game. This class will be initiated when credits view is on the view.

### Methods:

-*backToMainMenuClicked()*: When user clicks Back To Main Menu button, in the credits view, it will take the user to the home screen. To do that, this method will set parent scene as *HomeScreenView.fxml*.

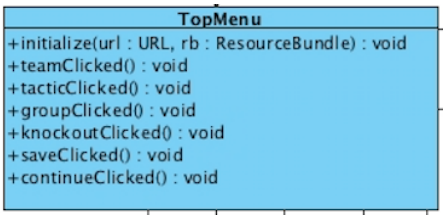



*Figure 4.2.12 TopMenu controller class*

**TopMenu:**

This method's view class, which is *TopMenu.fxml,* is placed at the top of every view class. This class controls which button is pressed at the top of each view.

**Methods:**

-*teamClicked():* This method is called whenever the user clicks Team button in the top menu. This method loads *TeamView.fxml* to the Parent scene. Then, the dimensions of this scene are set here in that method. Afterwards, main stage is set to this scene.

-*tacticClicked()*: This method waits until Tactic button is clicked. If the user clicks Tactic button, this method will load *TacticView.fxml*. This method will also assign related dimensions according to given numbers. Main stage is also changed accordingly in this method.

-*groupClicked():* This method is for *GroupView.fxml*. Thereafter, the game will proceed to the view of the groups.

-*knockoutClicked():* When the user wants to see state of the knockout, s/he needs to call this method by clicking knockout button. It will continue to the *KnockoutView.fxml*.

-*statClicked()*: This method will be called when the user clicks Statistics button. Then it will display statistics at the *StatisticsView.fxml*.

-*calendarClicked()*: When the user clicks Calendar button, this method will take them to the calendar view.

-*saveClicked()*: When the user clicks Save button, it will save the game and take the user to the Home Screen by loading *HomeScreen.fxml* to the parent scene.

-*continueClicked()*: When the user clicks Continue button, it means the game needs to go forward. It will it in Match Play View after going to this fxml class.

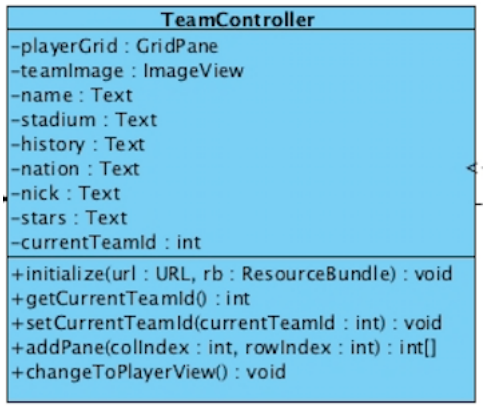



*Figure 4.2.13 TeamController class*

## TeamController

This is a controller class of the team view. This class will set team's data to the required fields.

### Methods:

-*initialize():* This method sets all 20 players' information to the required fields. It will also set teams' own information in that view, e.g. logo, stadium, name.

-*addPane():* It is a method that provides to access players info page by clicking on their data.

-*changeToPlayerView()*: This will be called when the user clicks on the players. It will take the user to the player view.

### Attributes:

-*playerGrid:* This is a grid which contains players' information.
-*teamImage:* This is a logo of the team as an image.
-*name*: This is the name of the team.
-*stadium*: This is a string for a stadium of the team.
-*history:* This string will contain a brief text about the history of the team.
-*nation*: This is the nationalities of the players as a string.
-*stars*: This text will contain information about stars of the team.
-*currentTeamId*: This is the id of the current team as an integer.

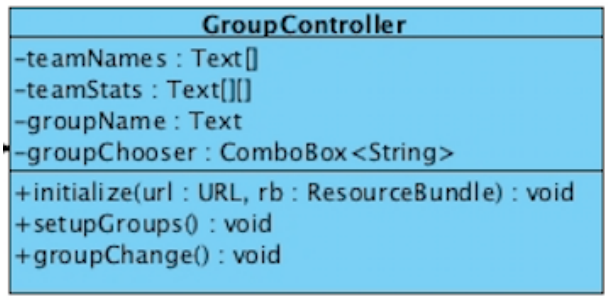



*Figure 4.2.14 GroupController class*

## GroupController

This class is the controller for the group view of the tournament. Its view is *GroupView.fxml*. This class will control all of the 8 groups. Each group will be held as an object of Group class. Then each group will have its name as an string. These names are used in Combobox of the group names.

### Methods:

-*initialize()*: This method will add all group names to *groupName* of *ComboBox<String>*. Afterwards, setup method will be called.

-*setup()*: This method will setup the group according to the given group name. Every team's name, number of wins, number of played matches, number of wins, number of draws, number of loses, number of scored goals, number of concede goals and points will be set, as a text, by its *setText()* method.

-*groupChange()*: This method will change group according to group name. It will change the group by calling *setup()* method with the group name.

### Attributes:

-*teamNames[]:* These are the names of the teams in the group.
-*teamStats[]*: This will continue the statistics about the teams of the group.
-*groupName*: It is the name of the group.
-*groupChooser*: It is a Combobox which provides options to choose among groups.

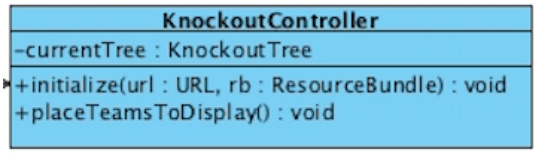



*Figure 4.2.15 KnockoutController class*

## KnockoutController

This is the controller class for the knockout view of the tournament. This class will be used after group stage is finalized. It will provide control for the *KnockoutView.fxml* class.

### Methods:

-*initialize():* This method will arrange the teams according the return data of the *placeTeamsToDisplay()* method. Winner of the pair of the teams will be highlighted with red color, while the loser team is faded with gray color.

-*placeTeamsToDisplay()*: We will utilize an object of the KnockoutTree class for this class. This method will check the score of the game and will place the teams to their places in the knockout tree, accordingly.

### Attributes:

-*currentTree*: This is an object of the KnockoutTree class. This is used to store the teams of the knockout in a tree.

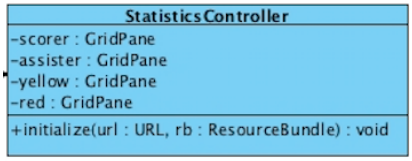



*Figure 4.2.16 StatisticsController*

## StatisticsController

This is the class for statistics view which contains statistics of the tournament. It will arrange the statistics of the tournament according its top players.

### Methods:

-*initialize()*: In this method, top players' statistics are fetched and assigned to related strings to be set to related fields later on.

**Attributes:**

-*scorer*: This is a GridPane for the top five scorers of the tournament.

-*assister*: This GridPane is for top five assisters of the tournament.

-*yellow:* This GridPane will contain top players in terms of number of yellow cards.

-red: In this GridPane will contain top players in terms of number of red cards.

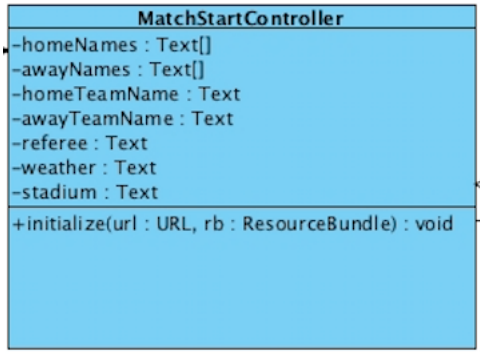



*Figure 4.2.17 MatchStart controller class*

## MatchStart

This is the controller for the view that is seen just before match starts, namely *MatchStart.fxml*. This class puts information about the teams, the stadium and the weather. Moreover, this class has buttons to kick off the match which proceeds to *MatchPlayController.*

**Methods:**

-*initialize():* This method will create a new match and will set related properties of a match, such as teams, weather, stadium, etc. It will also create new and empty ArrayList of actions to be filled later. Later on, both teams' names and their players' names are going to be set by its set methods.

**Attributes:**

-*homeNames:* These are the names of the players of the home team.

-*awayNames*: These are the names of the players of the away team.

-*homeTeamName:* This is the name of the home team.

-*awayTeamName:* This is the name of the away team.

-*referee:* It is the name of the referee.

-*weather*: This string will hold the information about the weather.

-*stadium*: This is the information about the stadium.

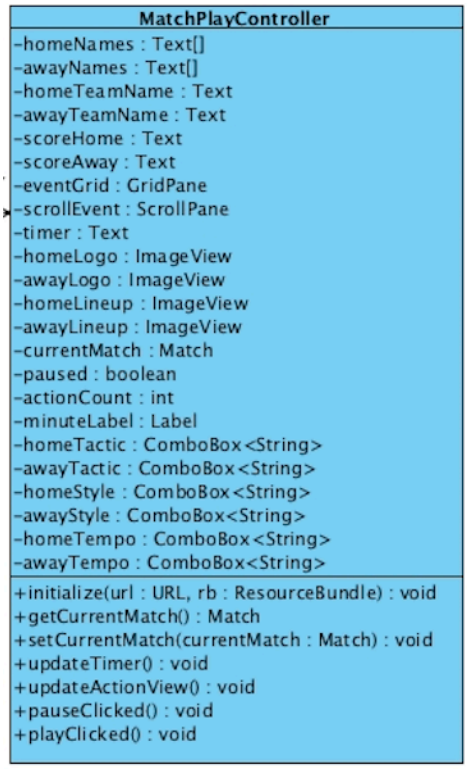



*Figure 4.2.18 MatchPlayController class*

## MatchPlayController

This is the class used with *MatchPlayView.fxml* class. It will control the matches that are played between the teams. This class will be initiated when the user clicks Continue button in the game.

### Methods

-*initialize()*: This method will get an instance from the Tournament and finds the next match in a for loop. After that, it will arrange the view of the screen according to the names and logos of the home and away teams. Thereafter, it will arrange the tactic, tempo and style of the teams with respect to preference of the user. Finally, this method will place lineups of the both teams and call *doTime()* method to forward the time of the match.

-*doTime():* In that method, Timeline class is used to manage the time of the match. It will continue for 90 seconds. At every second, if the match is not paused, one action will be created by *actionGenerator()* of the *Action* class. If there is any action it will update required slot in the view with *updateActionView()* method, accordingly.

-*updateActionView()*: It will fill the eventGrid, which is there for generated actions, with created actions in the *doTime()* method.

-*pauseClicked()*: A method to pause the game by changing value of the *paused* value. This method is called when the pause button is clicked.

-*playClicked()*: This method will play the game by changing value of the paused attribute.

**Attributes:**

-*homeName*: It is the name of the home team.
-*awayName:* It is the name of the away team.
-*scoreHome*: It is the score of the home team.
-*scoreAway:* It is the score of the away team.
-*eventGrid:* This is a GridPane to use for actions generated during the match.
-*scrollEvent*: This is a ScrollPane to be used if there are pretty many actions generated during the game.
-*timer:* This is the text to display the time of the match.
-*homeLogo:* Logo image of the home team.
-*awayLogo*: Logo image of the away team.
-*homeLineup*: Lineup image of the home team.
-*awayLineup:* Lineup image of the away team.
-*currentMatch*: This is an object of the Match class that represents the currently played match.
-*paused:* Boolean value used to pause the game.
-*actionCount*: Number actions generated in the current match.
-*seconds:* An integer for the number of the seconds played until a specific time.
-*label:* A label that seconds proceed in time in it.
-*homeTactic*: ComboBox to get the tactic of the home team from the user.
-*awayTactic*: ComboBox to get the tactic of the away team from the user.
-*homeStyle*: ComboBox to get the style of the home team from the user.
-*awayStyle*: ComboBox to get the style of the away team from the user.
-*homeTempo*: ComboBox to get the tempo of the home team from the user.
-*awayTempo:* ComboBox to get the tempo of the away team from the user.
-*actions:* ArrayList of the actions to be generated in that match.

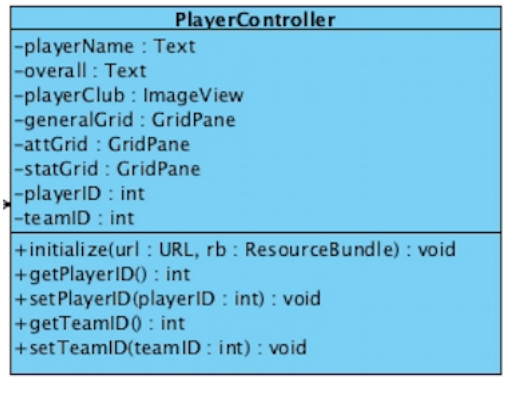



*Figure 4.2.19 PlayerController class*

## PlayerController

This is the controller class of the players of the teams. This is used in *PlayerView.fxml* to fill the places each field according to the players' information.

### Methods:

-*initialize()*: We have only one player view class and this needs to be filled correctly according to each players data. This is accomplished by that method. Players all information, such as name, age, image, etc. is fetched according to his id, namely *playerId*. Henceforth, these data is set to its related place by its set method.

### Attributes:

-*playerName*: A field for the name of the player.
-*overall*: Overall rating of the player is calculated and kept as a text in that field.
-*playerClub*: It is a logo of the players club as an image file.
-*generalGrid*: Informations of the player is arranged as cells of the GridPane.
-*attGrid*: This GridPane is for the attributes of the player.
-*statGrid*: This the GridPane which will contain statistics of the player e.g. goals scored.
-*playerID*: This is the unique id of the player.
-*teamID*: This is an integer for the id of the team.

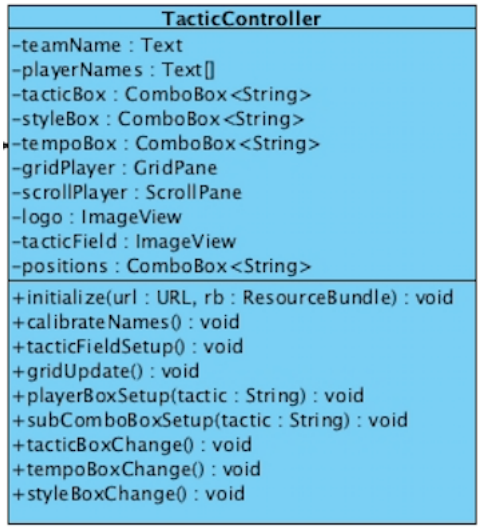



*Figure 4.2.20 TacticController class*

## TacticController

In this view, the user will see tactic of his/her team's lineup on the right and names of players on the left with the their positions. Moreover, the user can change team's tempo, style and tactic here. This is accomplished by using comboboxes.

### Methods:

-*initialize()*: This method will setup team's lineup according to its tactics by using *tacticFieldSetup()* method. Afterwards, names of each player will be put under their positions, accordingly. It will be accomplished by using *calibrateNames()* methods.

-*calibrateNames*(): This method will put each players name under its own position according to the tactic of the team.

-*tacticFieldSetup()*: This method will setup team's tactic view image according to its tactic.

-*playerBoxSetup*(): This method sets each players positions to the combobox next to his name. By doing that we are enabling player change in the tactic view. These positions are set according to team's tactics.

-*comboBoxSetup*(): In this method, comboBoxes next to players' names are assigned according to the tactics.

-*tacticBoxChange*(): This method will change the tactic box of each team according to its tactic.

-*tempoBoxChange*(): In this method, tempo box of the team will be changed according to its tempo.

-*styleBoxChange*(): Style box of the team will be changed according style set by the user.

**Attribute:**

-*teamName*: It is the name of the user's team.
-*playerNames*[]: These are the names of all players of the users team.
-*tacticBox*: This is the combo box for the user's team. to hold its tactic.
-*styleBox*: The style of the team's will be hold here.
-*tempoBox*: This is the combo box for the tempo of the team.
-*gridPlayer*: This is the grid which will contain the players.
-*logo*: This is an image which will contain logo of the team.
-*tacticField*: This is an image field of the according to its tactic.
-*positions*: This is a combo box for the positions of the team.

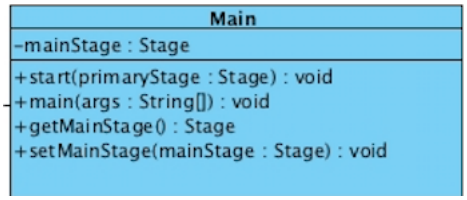



*Figure 4.2.21 Main class*

## Main

This is the main class of the "STARS League". This class is initiated when the user opens the application. It will create stage and set the scenes accordingly with its methods. It contains the main method to execute the program. Also it is the Façade class for the controller classes of program. It organizes the control of the controller classes.

**Methods:**

-*start*(): This method is used to create stage and set scenes to that stage. *mainStage*'s properties, like title, scene, etc. are set in that method. After all, this is set as *primaryStage*.

***Attributes:***

-*mainStage:* It is the Stage that is used for changing the scene in the program. Its features are set before the execution of program, fullscreen, exit-hint etc, and used throughout the program.

## 4.3 Object design trade-offs

### 4.3.1 Database Design vs Object Design

In the application, we implemented the Singleton design pattern in the Tournament class. Because we read the previous game data from our database just after the game is opened, then we modify these data in the main memory. Before exiting from the game, if user selects the option "Save Game", then the current modified data is written to our database. So our program will use database at most twice every time the game is played. Also there is no need for complex database design for that structure.

However, to implement that idea, we need to use the same instance throughout the different controller classes. Therefore we use Singleton design pattern. However, this design may bring some problems such that we need to think where this Singleton object should be created and how it will be accessed in program. So decreasing database complexity brings increased object design complexity.

### 4.3.2 Object Design vs Time Complexity

In STARS League, we used Façade design pattern for the implementation of Group, Elimination and Tournament classes in model package. Group and Elimination classes put other model classes such as Team, Player etc to use their features accordingly. Also the Tournament class brings Group and Elimination objects to

construct the real life tournament structure. Thus, using Façade design pattern increases code readability and fastens the development process.

However this brings problems in terms of the access times. For instance, when we use facade pattern, to access and manipulate an attribute of a player in a match that is played in group stage of tournament, we need to access the Group object some extra times because we build the Tournament class without Group class.

On the other hand there would be no need to access the Group class additionally, if we could access to the related object of it after following some conventions and this would increase the access time. However, using Façade design pattern helps us to organize the code and helps us in development process while decreasing the access times to certain objects.

## 4.4 Improvement Summary

For determining a better game we added transfers to our game. Thus, clubs will be able to do transfers in our game. Thus, effect of the user on team's success has increased. In addition, our game is now more enjoyable than older version. We added a help button to our game menu to show the beginners how to play STARS League. There are screenshots from our game view with their explanations at the help view. Thus, even beginners can play easily by our user-friendly interface. We made the game full-screen for the all platforms. We removed usability vs functionality tradeoff and added development time vs functionality tradeoff for implementing more enjoyable game. We also added ease of use vs functionality trade off for making game user-friendly. We added requirement for running to game have 1400x900 resolution.

We changed design of the tactic screen, the color of the jerseys on the tactic screen will be displayed according to teams' colors. We also added music to our game for making it more enjoyable. On the other hand, this feature will require a speaker or headphone to user to listen it. We used singleton design pattern to make implementation simple. We will also access the game database at the beginning of the game or in the end of the game. We also changed the game model subsystem. We connected tournament with group and elimination and connected actions with match because match is consist of actions and its score is determining by the actions. We also connected game model with database. We explained the methods of the classes to make our project more understanding. We changed the final object design. We added new objects and showed the variables at the tournament class.

We also added methods which have explained before to the tournament class for processing of the tournament. We changed the methods at database class and added loadCurrentGame(), saveCurrentGame() and loadRandomMusic methods which explained before to database class. We also added Database Editor class which stores the 32 teams which are participating to STARS League, information. These informations includes, team's players, manager and president. We added parameters to president object. We added variables to player class for calculating the statistics. cntMatch, cntGoal, cntAssist, cntYellowCard, cntRedCard added for counting players statistics. We also added methods to player class according to count variables for statistics of the players. We added a method for calling first eleven players for starting to the match. We added creditsClicked() method to provide credits screen to the user.

We added helpers for controlling subsystems. We added newGame controller for getting manager information, selected team and groups. Thus, game will start after getting these informations. At the team controller class we added informations of teams (name, stadium, history, nation, nick, stars, currentTeamID). We added some combo boxes to tactic controller for creating user-friendly interface. With these combo boxes, user can do changes at the tactic screen easily. We added information about match day as homeTeamName, awayTeamName, referee, weather, stadium. We added many features for match controller as score home and away, timer, home and away logo etc. We specialized statistics by scorer, assister, yellow, red.

## 4.5 Packages

In total, the system has 3 subsystems called "Game Controller ", "GameView" and "Game Model".

- "Game Controller" consists of 2 packages called "Controller" package which has almost all classes related to controller of the system and "Main" package which has only "Main" class.
- "Game Model" consists of 2 packages called "Model" package which has almost all classes related to model, and "Database" package which has only "DatabaseConnection" class.
- "Game View" has FXML files.

# 5. Glossary & references


[1] https://www.w3schools.in/java-tutorial/java-virtual-machine/

[2] http://www.businessdictionary.com/definition/graphical-user-interface-GUI.html

[3] https://www.webopedia.com/TERM/C/cross_platform.html

[4] http://www.klabs.org/history/history_docs/sp-8070/ch4/4p1_design_tradeoffs.htm

[5] https://wwwbruegge.in.tum.de/lehrstuhl_1/component/content/article/43-books/243-oose-template-systemdesigndocument

[6] https://www.youtube.com/watch?v=hUE_j6q0LTQ&t=1088s

[7] https://www.youtube.com/watch?v=K4FkHVO5iac&t=515s

[8] https://www.pluralsight.com/courses/design-patterns-java-creational