CS321 – Final Project

Soccer season simulation

Program overview

The project goal was to simulate a soccer season. A season is comprised of a number of teams that play the other twice. A game has a home and away team, both teams have to select 11 players in specific positions. A team has a roster of 26 total players. A player has a position and certain attributes. These elements are combined into a game logic that would calculate the final score. Then once all teams have played their games, the season is completed. Each team will then have a number of wins, draws, loses displayed to the user.

Team and player

A player has 6 attributes, they are mental attack, mental defense, physical attack, physical defense, technical attack and technical defense. To simplify the overall calculation, 2 additional attributes, total attack, total defense was created. They are the sum of the 3 categories. Each player also has a position, goalkeeper, defender, midfielder, attacker. A stamina attribute is provided to each player, this is used as a multiplier in the game logic. Lastly, a name and age are generated for the player.

A team is composed of a squad of 26 players. Only 11 can be selected to play a game. The starting 11 is composed of 1 goalkeeper, 4 defenders, 4 midfielders, 2 attackers. A team morale is generated, this is used as a multiplier in the game logic. A random name is generated for each team.

Game Logic and algorithm

Although soccer is a dynamic game, the choice was made to simplify it and allow each team equal opportunities. A game is 90 turns so each team would have 45 chances. There is also a home team advantage, this is to simulate a real-world equivalent.

For the attacking side, the attribute total attack was added up for the 10 players (4 defenders, 4 midfielders, 2 attackers). After each attack, those players would lose 2 stamina. For the defending side, the attribute total defense was added up for all 11 players. After the defense, these players would lose 1 stamina. The team's attack and defense totals were added to their stamina and team moral multipliers. Since the attacking team statistics can be lower, the value xG (expected goals) is the rounded absolute value of attacking team minus defending team. We then check the possibility to score a goal, using the Java nextGuassian call, it generates a random value in a normal distribution where the mean is 0.0 and standard deviation is 1.0. So if the attack is greater than defense, there is a 50% chance to score. Then if the xG is 0 or 1,

meaning that the attack and defense are relatively equal, there is a 30% chance to score. Lastly, if the defense is greater than the attack, it is required that the chance number modulus of the xG equals to 0 and the random variable has again 30% chance to score. Then once the 90 turns are up, the score will determine the outcome of the game.

Design and implementation decisions

Patterns used

For a season to create a number of teams, the factory pattern was used. Although this is not a fully incorporated into the design pattern description, allow the subclasses to choose the type of objects to create. It allows for a simple way to generate each team by providing it with a number of teams the user wants to simulate the season with.

The player is a façade pattern as the team will simply call for 26 players to be created. The player class has a subtype of goalkeeper, defender, midfielder, attacker. So it acts as a façade for the caller which can ask for a player and check its position if required.

A team calls a player factory to create the set of required players. This would a true factory pattern as the player is a façade of 4 subclasses. Each of these 4 classes have their unique positions. The players that they produce are different as their attributes are reflected to the position they play (attacker has better attacking attributes than defender, and vice versa). The usage of a factory provides a simple way to generate the players required to fill the team.

For the game logic, a chain of responsibility pattern was selected. It is used to divide the game logic code into smaller sections. This decreases the chance for the program to run into error while writing code and allows better maintenance if changes are to be added (we want to play extra time in playoff situations if both teams a tied after 90 minutes). The implementation starts at GameFlow and the subclasses GameFlowLogic, GameFlowWinner. A game is passed to logic to calculate the score of the attack and defend phase. Once completed, it is then passed to winner to determine which the outcome of both teams.

For a team to pick the 11 players for a match of 26 players, the strategy pattern was used. As there are various ways to fill out the 11 players from the roster of 26 players, this pattern is well suited for the implementation. 3 subclasses were implemented for a different strategy, BestXI, WorstXI, RandomXI. The best and worst are simply sorted by the overall attributes of each player. The Player class implements Comparable to allow the use of Collections.sort to quickly sort all of the players. Afterwards, the strategy selects the players to put in the starting 11. The usage of this pattern allows additional strategies can be put implemented at a later time (best defense, best attack). It also offers better maintainability for the current strategies in place. Currently, strategies are picked in a normal disturbed manner where 67% chance of BestXI, then 33% chance for either WorstXI or RandomXI.

Data structures

The usage of the ArrayList to store the set of teams in Season, the squad of 26 players and the starting 11 players in the Team. The choice to use the ArrayList was due to the familiarity with it. It also has various build in methods that would be used such as the iterator for use in the for each loop.

Otherwise, a simple array was used to generate the permutation of games to be played in a season. It was used in Season where the function getPermutations required the use of 2 simple arrays, 1 for empty elements and another for the team. The permutations would then be called by match to select the teams that are playing games.

Multithreading

A simple multithreading of matches was implemented. It would call 2 teams from the list of generated matches in a season. The list is shuffled to allow a random order. This would mean that if a team was already in a match when called upon to play another match the thread would be locked. As such, it could either go to another match in the generated list. However, this would mean that we would have to come back to it later on. Therefore, the choice was to simply look the 2nd match and wait for the team to finish their current match and then play the next one.

Current weaknesses and Further improvements

The simulation cannot handle double digit teams in a season. This is due to the way which the program generates the matches for those teams. A 2 character string 12 or 56 is used to represent the home team of the left and away team on the right. There is a constrain put in place which only allows an even number of teams which is why currently it can only complete a simulation with 2,4,6,8 teams in a season. In order to fix this issue, a change would have to be made to 4 characters when generating the matches, this way 0209 or 1220 would indicate that team 2 and 12 are home teams and team 9 and 20 are away teams.

The usage of the skeleton code is minimized and as such, inputs that are past the total number of matches in a season will just display that all matches are played. Further changes can be put in place either by terminating the program as it had all the time it needed to finish the simulation. Or it can be used to calculate the total number of actions in the simulation and then generate additional seasons for a given number of teams or generate a larger number of teams to allow for more games in a season.

A better user interface would allow more interaction for the user. The user could pick a team they would like to create, manage their team by selecting their players or using a defined strategy. They could then mix up their strategy before each game to ensure better results. The

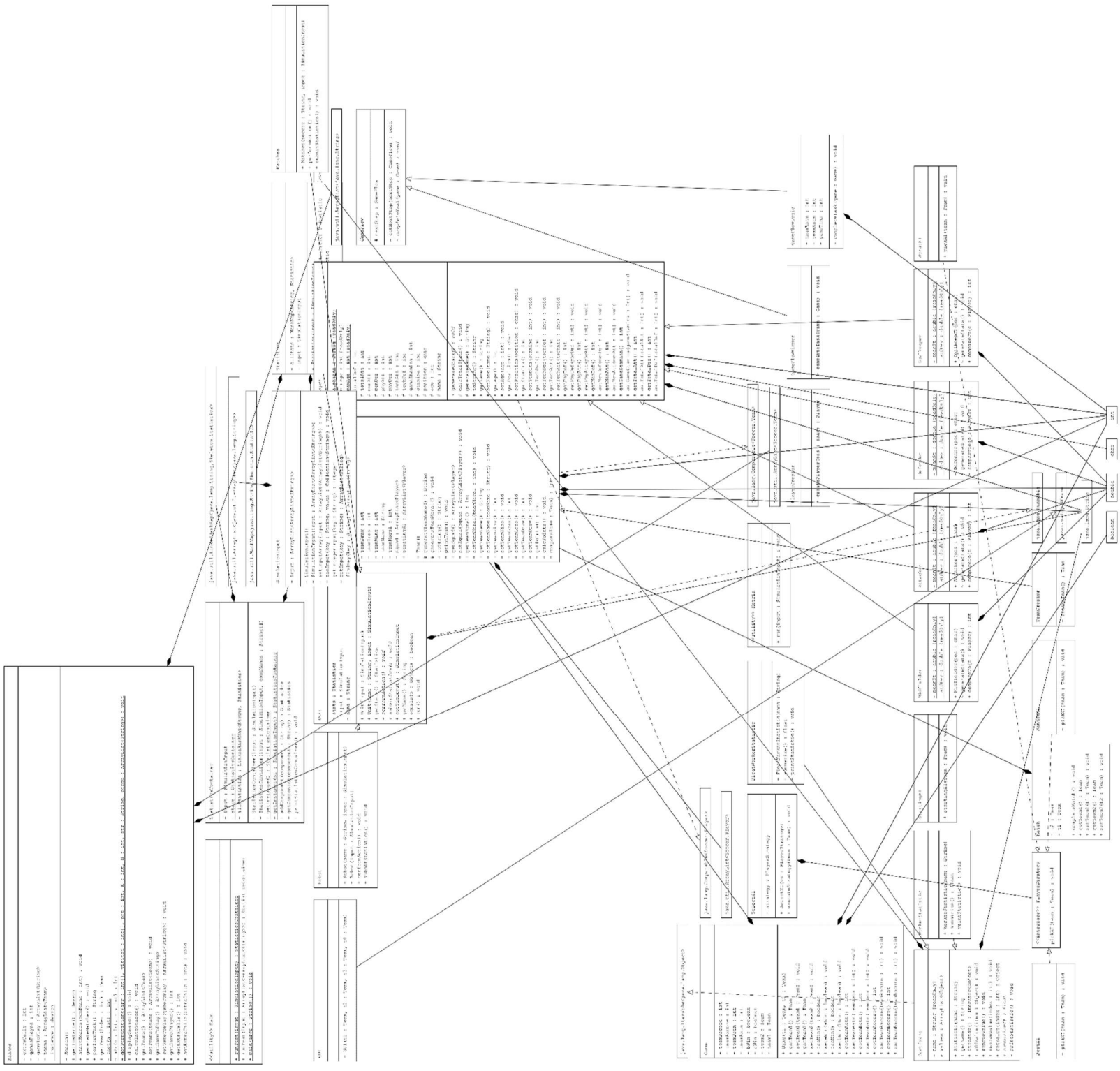
user would also be able to check on each player and look at their attributes to decide who to put in the matches.

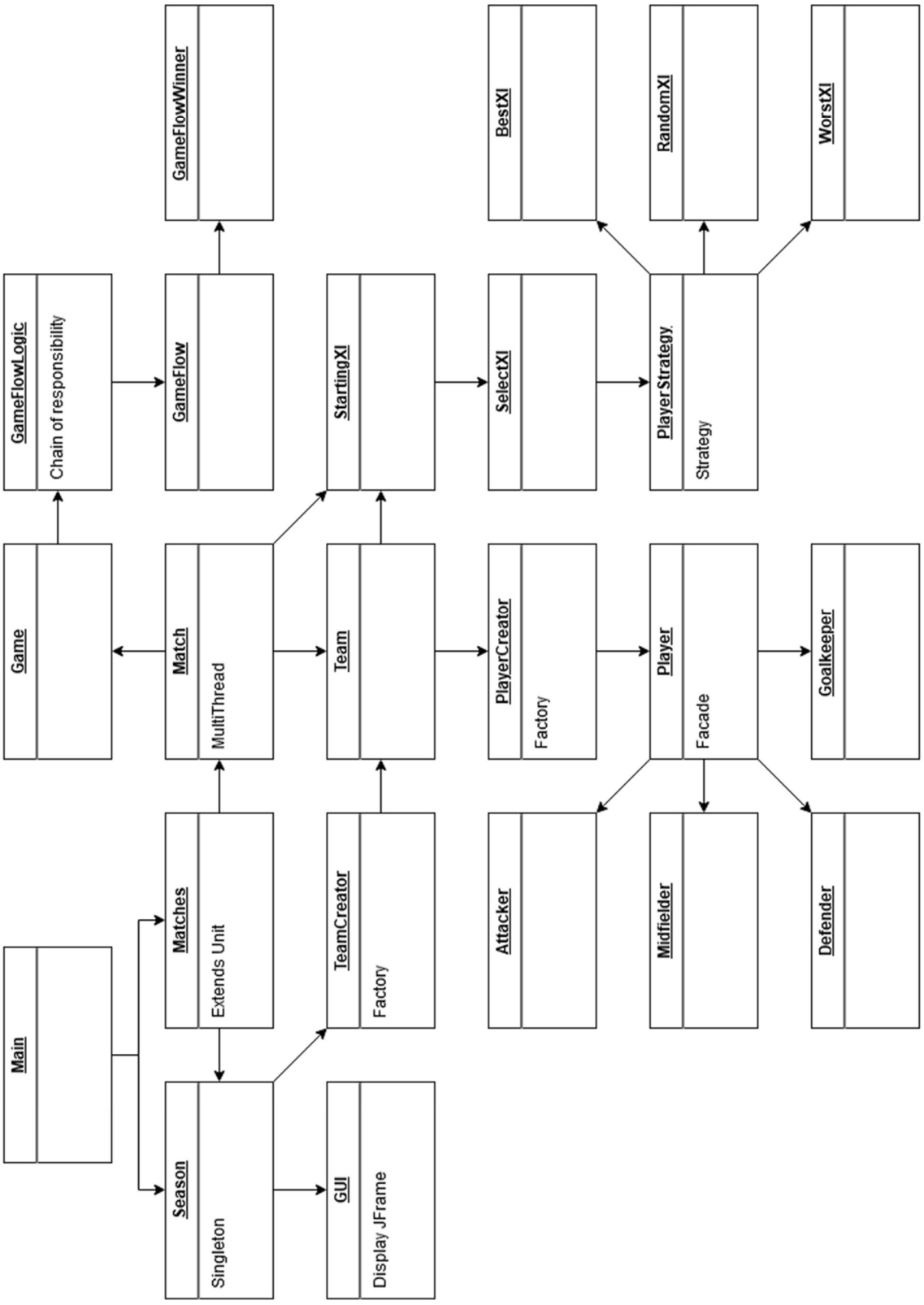
Finally, as the program grows larger, the use of a database to store the teams and the players would be required for better performance. It would allow for expanded season, leagues and other various items that can be added to this program.

Additional notes

2 diagrams are provided, 1 is a UML diagram of all of the code from a plugin. The other is a simplified diagram where the arrows are used to point in the direction of each relation. Each box is a class which calls another class as per the arrow direction. It can also be used to check the way the simulation will progress from one class to another.

Lastly, this program was inspired by a game, “Football manager 2021”. After looking at the overall function of the program, the current code reflects the minimum viable product that would fulfill the requirements of this project. Items such as the player attributes are a simplification of similar items that are found in game. It also has a very interactive UI which would allow the user to select players. The game also moves forward in time so players on a team can improve their attributes. New players are also created as well. Many other factors such as team staff are also used. However, many of these items were discarded as the implementation would have required much more time.




Main
Matches
Extends Unit
TeamCreator
Factory
Season
Singleton
GUI
Display JFrame
Game
Match
MultiThread
Team
PlayerCreator
Factory
Attacker
Player
Facade
Midfielder
Defender
Goalkeeper
GameFlowLogic
Chain of responsibility
GameFlow
GameFlowWinner
StartingXI
SelectXI
PlayerStrategy
Strategy
BestXI
RandomXI
WorstXI