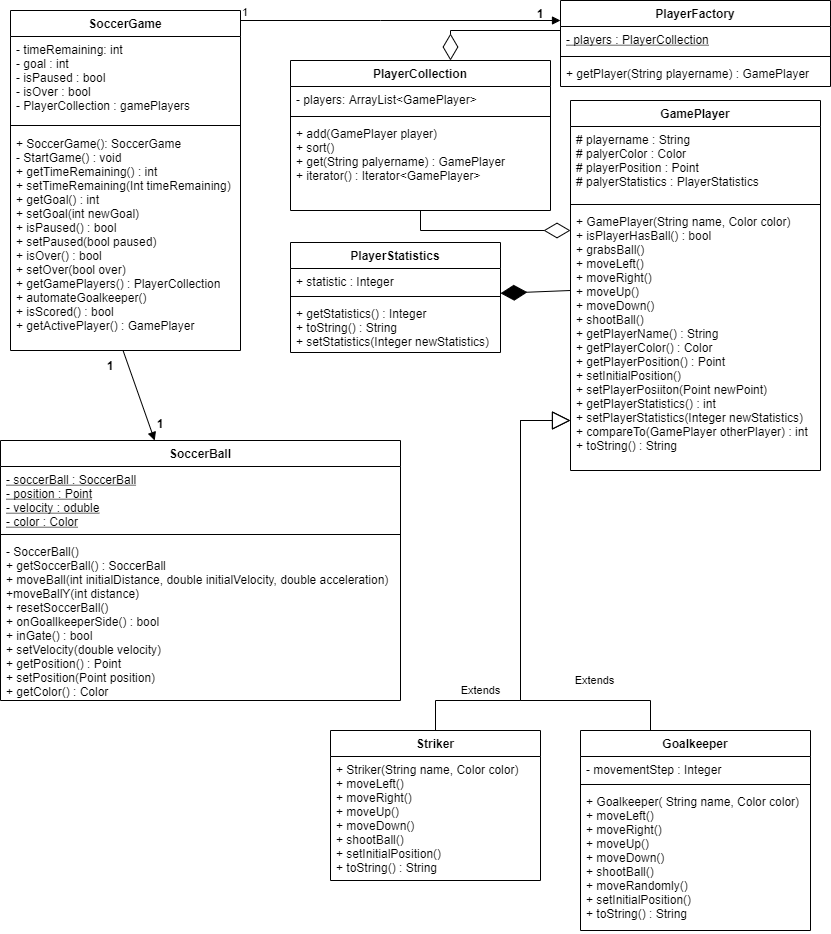
**Introduction:**

The goal of this project is to make a mini soccer game in which there is a striker and a goal keeper. The goal keeper is controlled by the computer while the player controls the striker. The striker attempts to score a goal while the goal keeper moves in a random fashion. In this project we must implement the right classes for the environment as well as players and the ball. We must also implement classes to take care of location, score, and time. We must make sure that we implement and link all the classes and methods properly, as an issue in one will cause an issue in the whole system. We will be using inheritance in the implementation, as both striker and goal keeper are players in the game. We must also use polymorphism for the movement of the striker and keeper, as well as their positions and actions. The report will be structured as follows: first we will show a UML class diagram of the system and explain how we used design patterns in our diagram. We will then explain the different object-oriented design principles we used within our implementation. Following that we will explain how we implemented the different classes, the unit testing done on the system, and the types of tools and libraries used in development. Finally, this report will end with a conclusion that will give brief idea of the accomplishments and challenges of this project, as well as what we have learned and recommendations.

**Design:**



**Elements:**

* SoccerGame:
  + Contains overall info on the game like time and goals.
  + Instantiates a soccer ball, one ball used throughout game
  + Contains the collection of players for that game
  + Methods give info about the game such as active players, is the game over, is it pause, did the time run out, etc.
* SoccerBall:
  + Made using singleton pattern, constructor private, instantiated by soccergame using get method.
  + Contains color, speed, and position of the soccer ball.
  + The methods give info about the soccer ball, notice a soccer ball is instantiated using the getSoccerBall method which calls the private constructor.
* PlayerFactory:
  + Returns collection of players, part of the factory design method, creates new collections keeping the creation logic hidden from the user.
  + Has getPlayer method to return a specific GamePlayer.
* PlayerCollection:
  + Iterable class that creates a linked list data structure of GamePlayers. Contains methods to add, sort, get, iterate. Notice iterator is a polymorphic function that overrides the built-in iterator method
* GamePlayer:
  + Parent class of striker and goalkeeper, as both are gameplayers.
  + Contains abstract methods for movement(up, down, left, right) as well as abstract methods for shooting the ball, setting initial position, and an override of both compareTo and toString built in methods.
  + Also contains methods to get a players name, color, stats, and position. As well as setters for stats and position.
* Striker:
  + Implements movement for the striker class as well as initial position, shootball, and toString.
* Goalkeeper:
  + Implements movement for the goalkeeper class as well as initial position shootball, and toString.
* Statistics:
  + Contains an int that represents the score of the player.
  + Methods include a setter and getter as well as an override for toString.

**UML design patterns and Object Oriented Design:**

Within this UML class diagram, we have used two design patterns, the factory method and the singleton method. An example of the factory method can be seen in an interaction between the SoccerGame, playerFactory, playerCollection, and GamePlayer classes. Soccergame gets a new player collection from player factory. Playerfactory calls on the iterator class playerCollection to create an Array List of GamePlayers. This way playerFactory can create collections of players. The singleton class would be our soccer ball, with a private constructor and a static soccer ball. The soccer ball is then created by soccergame through the getMethod. In terms of object-oriented design, this project incorporates both polymorphism and inheritance. Both striker and goalkeeper are children of the GamePlayer class. Also notice that Gameplayer abstractly overrides toString and compareTo methods initially and they are further overridden inside both the goalkeeper and striker classes. GamePlayer also initially declares the movement methods as abstract and they are implemented inside of striker and goalkeeper as overridden.

**Implementation:**

We started off this project by initially compiling and running the MVC template that was given. Upon each run, we debugged the errors that came up, creating the classes and methods that were needed just to get it to compilation. Once we had reached compilation, the new task was to make sure that it was running right. Our approach to this problem relied heavily on debugging, and we chose this approach because it made it easier to find the missing elements within the template. Once we had the game running it was an easier task, we just had to make sure that all the functions were doing what they were meant to be doing. In errors where the striker would cross the line or couldn’t shoot, we would setup break points in order to pinpoint the exact instant that the error occurred and then fixed the logic for that part. We use IntelliJ ultimate for this project as it has many built in libraries and tools that make development easier. We managed to connect jacoco to IntelliJ in order to make sure that our codes was error free. We ran junit test on out code, getting a percentage from jacococo to make sure we achieved 100% coverage of the code. Jacoco checks to see where the program could develop an error and whether or not you have covered that error using junit tests. By running it on our code, we were able to make sure that we didn’t have any random bugs that might show up, as everything is accounted for.

**Conclusion:**

This project was a great learning experience. I believe what went well in this project is my ability to believe in myself. Initially I thought I had already joined a group, by the time I realized I hadn’t, there was one group left. After joining the last group available with one other person, it turned out my partner was dropping the course. Due to these unfortunate circumstances I had to finish this project alone. I have learned that sometimes in life you are dealt a bad hand, sometimes that hand means doing the work of four people. Working in a group enables each person to focus on a small amount of work, resulting in a better overall project. Groups also enable diverse thought to come together in one harmony. At the same time, sometimes group members are incompatible, which can result in some unneeded issues.