EECS 3311 - Software Project 2

Lab Group 30

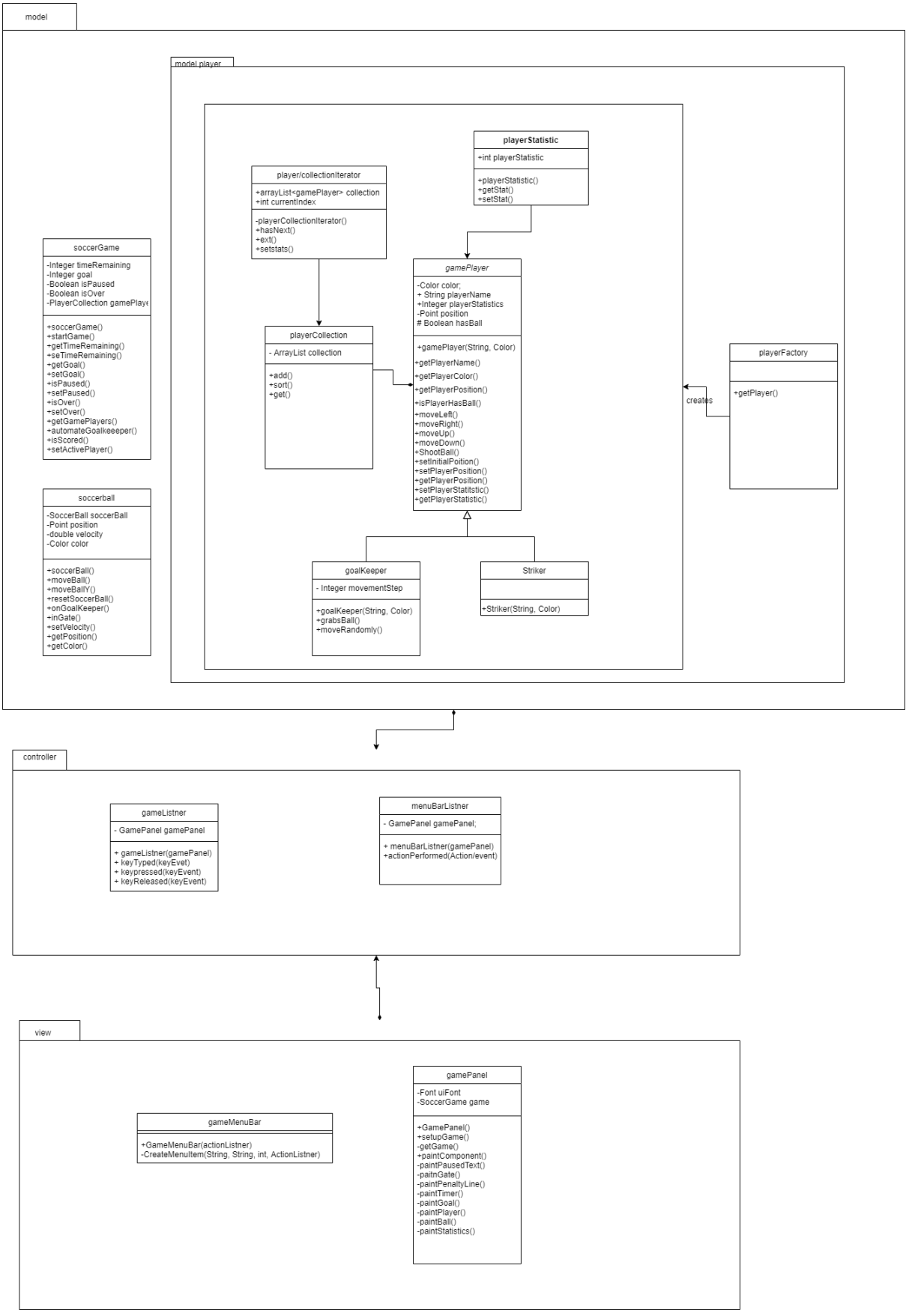
Parsa Alizadeh Tasbiti Haghighi	215246036	alzparsa	Section B
Ali Shariat	216710345	alish42	Section A
Syyed Hussain Fatmi	217180050	hfatmi12	Section A

PART I - Introduction

The software project called MiniSoccerGameProject was a mini soccer game in which a user plays against an automated goalkeeper and tries to score as many goals as possible in the allotted time. An incomplete project was provided and the goal of the project was to implement and complete the model package of the project. The software project involved some challenges, including group work and collaboration, and building test cases. A variety of concepts were used, some of which were inheritance, encapsulation, abstraction and polymorphism. The project also included the factory design pattern. In this report, the design pattern will be shown using a UML class diagram of the project. The design principles used will be explained thoroughly. This report will contain a detailed description of the implementation, and will be accompanied by a short demonstration video of the game.

EECS 3311 - Software Project 2
Lab Group 30

PART II - Design



EECS 3311 - Software Project 2
Lab Group 30

In this project, two design patterns were used. The Factory pattern and the Iterator pattern.

Factory Pattern:

- Includes the PlayerFactory, Goalkeeper, Striker and GamePlayer classes.
- The PlayerFactory is the main component of this pattern. This class allows the creation of different GamePlayers based on a given name.
- Using the getPlayer() method in the PlayerFactory, an appropriate GamePlayer is returned. If the name supplied corresponds to a Striker, the factory creates a new player that is a Striker and returns it. Likewise for Goalkeeper.

Iterator Pattern:

- Includes the PlayerCollection, PlayerCollectionIterator, and GamePlayer classes.
- The PlayerCollection is a collection of GamePlayers. We must have the ability to iterate through this collection. The PlayerCollectionIterator allows us to iterate through the collection easily.
- Using iterator() in the PlayerCollection class, we get a PlayerCollectionIterator for the corresponding collection.
- Once we have a PlayerCollectionIterator, we can use hasNext(), next() and remove() on the collection.

Throughout the project, many Object Oriented Design Principles have been used. Principles such as Inheritance, Abstraction, Encapsulation and Polymorphism.

Inheritance:

- Inheritance is involved in the relationship between Striker and GamePlayer. A Striker is a type of GamePlayer. Striker inherits all attributes and abilities of GamePlayer, as it is a child of GamePlayer. Strikers have the same properties like name, color, position ect. Striker has its own unique behaviours, as in the way a ball is shot or movement. The related methods are overridden.
- Similarly, inheritance is involved in the relationship between Goalkeeper and GamePlayer. A Goalkeeper is a type of GamePlayer. Goalkeeper inherits all attributes and abilities of GamePlayer, as it is a child of GamePlayer. Goalkeeper has its own unique behaviours, as in the way a ball is shot, movement and the ability to grab the ball. The related methods are overridden or added.

Polymorphism:

- Polymorphism describes the relation between a Striker and a GamePlayer. The Striker is a GamePlayer but has its own way to move and shoot a ball. We use the same method, but it is overridden in the child class to give it a unique implementation without causing confusion.
- Polymorphism also describes the relation between a Goalkeeper and a GamePlayer. The Goalkeeper is a GamePlayer but has its own way to move and shoot a ball. We use the same method, but it is overridden in the child class to give it a unique implementation without causing confusion.

Encapsulation:

- Encapsulation is used throughout the project. Attributes are kept private and are accessed through getters. This can be seen with GamePlayers (Goalkeepers and Strikers). It can also be seen in the collection and its iterator.

Abstraction:

- Abstraction can be seen in the way the internal processes of the game are hidden from the user. All the user knows is that the keeper moves randomly, the player is controlled with arrow keys, and the ball can be shot. The user does not know how the keeper is automated, nor do they know how the arrow keys change the position of the striker. They also do not know how the ball is animated and moved. Similarly with the game controls in the top bar, the user does not see the inner mechanisms. The user only has access to the visual interface.

Part III - Implementation

This project was built from the incomplete base code provided. The model package needed to be completed in order to run the game. Below, each class and its implementation will be explained. A javadoc containing this information was generated and is available (in the github repository) in addition to the following explanations.

- **GamePlayer**: This class represents any player in the soccer game, be it a goalkeeper or a striker. This class holds all common attributes and methods. Common attributes for any player include a name, color, position and statistics. Each player can also be in possession of the ball. These attributes were added to the class. Appropriate getters and setters were also added. All players can move in the left, right, up and down directions, so these methods needed to be implemented as well. In other classes, collections of players need to be sorted and so a compareTo() function also needed to be implemented. Players were compared based on name.
- **Goalkeeper**: This class represents a player that is a goalkeeper, meaning it must extend GamePlayer and inherit its abilities. Goalkeeper movement is automated and thus the movement is different and all movement methods must be overridden and some additional methods must be added - i.e. moveRandomly().
  A goalkeeper also has the ability to handle the ball with their hands, so a method grabsBall() must be implemented. This method allows the goalkeeper to grab the ball if it is within range or if it goes out for a goal kick.
  A goalkeeper kicks the ball in a different manner, so the shootBall() method is overridden.
  Goalkeepers are different and thus the string representation must be different. The toString() method is overridden to provide a string representation of the saves made by the keeper.
- **Striker**: This class represents a player that is a striker, meaning it must extend GamePlayer and inherit its abilities. Striker movement is controlled by the user and thus the movement is different and all movement methods must be overridden. These methods move the position of the striker by 5 in any given direction. If the player is in possession of the ball, the ball's position is moved with the player.
  A striker kicks the ball in a different manner, so the shootBall() method is overridden, to kick the ball in the direction of the goal.
  Strikers are different and thus the string representation must be different. The toString() method is overridden to provide a string representation of the goals scored by the striker.
- **PlayerStatistics**: Each player has some statistics. Goalkeepers have a number of saves made throughout the game and strikers have a number of goals scored. The

PlayerStatistics class is made to house each player's match statistics. PlayerStatistics is zero on creation as no saves or goals have occurred yet. A getter and setter is available to retrieve and change the statistics.

- **PlayerFactory**: This is a factory class that creates players given a name. Depending on the name given, the factory will return an appropriate GamePlayer of type Striker or Goalkeeper.
- **PlayerCollection**: The PlayerCollection is an iterable Collection of GamePlayers. This class holds a number of players that can be added and removed.
- **PlayerCollectionIterator**: This class is an implementation of Iterator. It provides the ability to iterate through the PlayerCollection, using hasNext(), next() and remove().

- Write JUnit tests for the model: use Jacoco to reach a 100% coverage. Discuss that in your report: 5

This project was completed mainly using Eclipse 2019-12 (4.14.0) and jdk version 13.0.1. Tests were made with JUnit. Some parts of this project required java libraries. Classes imported include the following:

- java.awt.Color
- java.awt.Point
- java.util.ArrayList
- static org.junit.jupiter.api.Assertions.*
- org.junit.jupiter.api.Test

Implementation of testing part of the project using JUnit is in the package testing of the project. The main challenge of this part was mostly covering all the test cases relative to the model.

- **hasBallTest**: 5 testcase to check all the 4 false case where and y are smaller and larger than the x and y required for striker to carry the ball and a true case to check the current case
- **moveDownTest**: 2 case for moveDown method to check false and true cases for method moveDown()
- **moveUpTest**: 2 case for moveUp method to check false and true cases for method moveUp()
- **moveRightTest**: 2 case for moveRight method to check false and true cases for method moveRight()
- **moveLeftTest**: 2 case for moveLeft method to check false and true cases for method moveLeft()

A video demonstrating the project is included in the github repository.

EECS 3311 - Software Project 2
Lab Group 30

PART IV - Conclusion

The implementation of the classes went well once the requirements and given classes were studied. Proper planning and understanding of the necessary components makes it easier to add the missing components.

Only minor problems were encountered during the project, all of which were fixed. The PlayerCollection and PlayerCollectionIterator classes were not implemented properly at first. The Iterator design pattern was overlooked, resulting in an implementation that needed to be redone. Once the pattern was followed the classes were properly constructed. THe second problem encountered was that the ball was moved incorrectly. When the striker is in possession of the ball, the ball must move with the striker. In the implementation, the ball was being moved in the wrong section, making the ball lag far behind the player. This was fixed once the code was moved to the correct location.

It is easy to learn by doing. By implementing different design patterns, such as the factory pattern and the iterator pattern, it provided a deeper understanding of these concepts and how they can be utilised. Complex tasks can be broken down by combined effort.

There are both advantages and disadvantages of working in a group. Some of the advantages of working in a group include a lighter workload as the work is divided among members. There are also the disadvantages. Working in a group requires collaboration, which can be quite difficult when members of the group have different schedules that do not necessarily align. Being online also has drawbacks, as there are no in person meetings making communication a challenge.

There are some ways to make this project less challenging. It is very beneficial to have a detailed breakdown of what is required. For example listing all the classes and their attributes/capabilities makes it much easier to implement the classes. Another way to better achieve the goals for the project is to prioritize the tasks. By prioritizing the tasks, the most important steps are completed first and it is easier and less stressful to get the rest of the project done. Lastly, to have better experience completing any project updates done to the projects should be communicated so everyone can see the improvements of themselves and other group members at any time.

As we worked in a team different tasks were assigned to each member. Below is a record of the activities of each member.

Member	**Task Completed**	**Collaborative**
Parsa Alizadeh Tasbiti Haghighi	Participated in project planning Created UML diagram Oversaw creation of tests Contributed to lab report	Yes
Ali Shariat	Participated in project planning	Yes
Syyed Hussain Fatmi	Participated in project planning Took charge of code implementation Managed and contributed to lab report Handled assignment submission	Yes
George Saweres		No