EECS 3311 - Lab 05
Design Report

Amir Mohamad, Saniz Momin, Akif Prasla, Hasnain Saifee

TA - Alireza Naeiji

Nov 7, 2021

Contents

1 Introduction

The software project is about generating a soccer game where there are two players: a goalkeeper and a striker. You have 60 seconds to score as many goals as possible, after each goal is scored your score increments and the game is paused. You can resume the game by pressing key R or by going to the menu option selecting control which has a sub menu resume and pause. To score a goal the striker can move by pressing up down left right key and to score a goal striker has to press space key. The goalkeeper randomly defends the gate by moving in the left and right direction bounded by the gate's length. After the timer becomes 0 the user gets the statistics of how many goals the striker has scored and how many saves has the goalkeeper made. When the game is over, the user can start playing again by navigating to the menu bar, clicking on the Game menu and clicking to a new game or exit if the user wants to end playing. The main-goal of this software project is to use OOP concepts and efficient software design patterns to write code efficiently and follow the DRY principle which is (Don't repeat yourself). In this software project, to properly organize our soccer game system, the system is divided into a set of layers where each layer focuses on a specific aspect and has an abstraction level. MVC architectural pattern is used to divide the system into three main subsystems where the model comprises data of the system, for example keeping record of total goals and saves of striker and goalkeeper respectively. The view consists of the user interface part where the data in the model subsystem can be viewed. The controller which handles interaction between the modal and the view. The detailed design of the software system can be seen using creational design patterns such as factory design pattern as well as prototype design pattern which focuses on creation of objects. For example players are created via the PlayerFactory class which has getplayer method which takes string as parameter. All that is needed to pass is a string and it will return the appropriate object. Doing a project with partial implementation takes time to understand the logic and the functionality of code written. It took a while until we all understood what is the role of each subsystem, once we got that it was easy to implement and finish the rest. Based on our understanding of this software project we are going to write the report with the basic idea and the designs implemented for the top level and organization of the software project. We will create a UML diagram to make our explanation of the design patterns used more clear. We will then explain the main elements in this system and explain their functionalities. We will conclude our report by giving brief overview of our journey in building this software project.

2 System Design

2.1 Overview

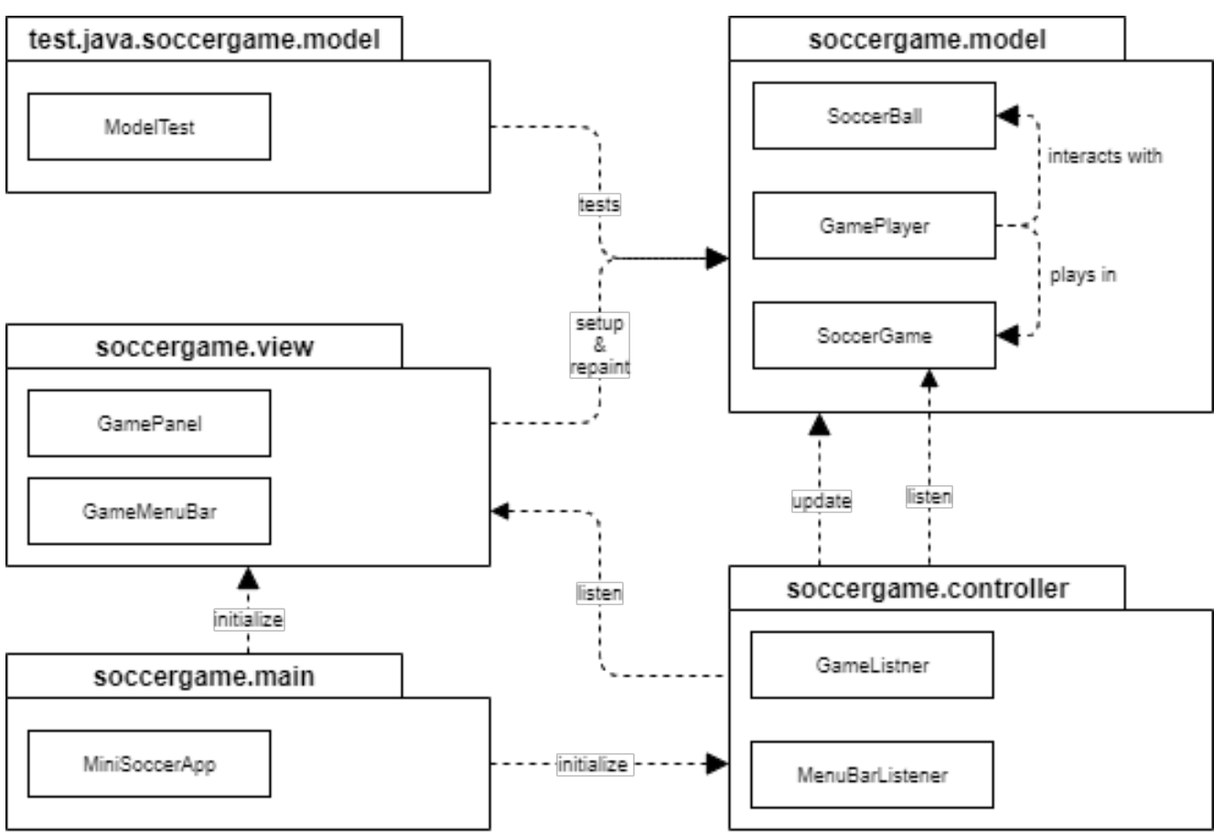

Figure 1: The system design of SOCCERGAME's

3 Front end

3.1 Overview

The front end of this project is implemented using the MVC design arcitecture. We have the components of our front end separated into the `soccergame.view` package and the controllers into the `soccergame.controller` package. This creates a separation of concerns between the classes which makes the development process easier. The main component of the front end is the GamePanel calss. It is where everything is displayed and managed by the controllers so the user can interact with the soccer game. The entry point to our program is the MiniSoccerApp class, which initializes all the components and sets up the GUI frame with the GameMenuBar and the GamePanel.

3.2 Class Diagram

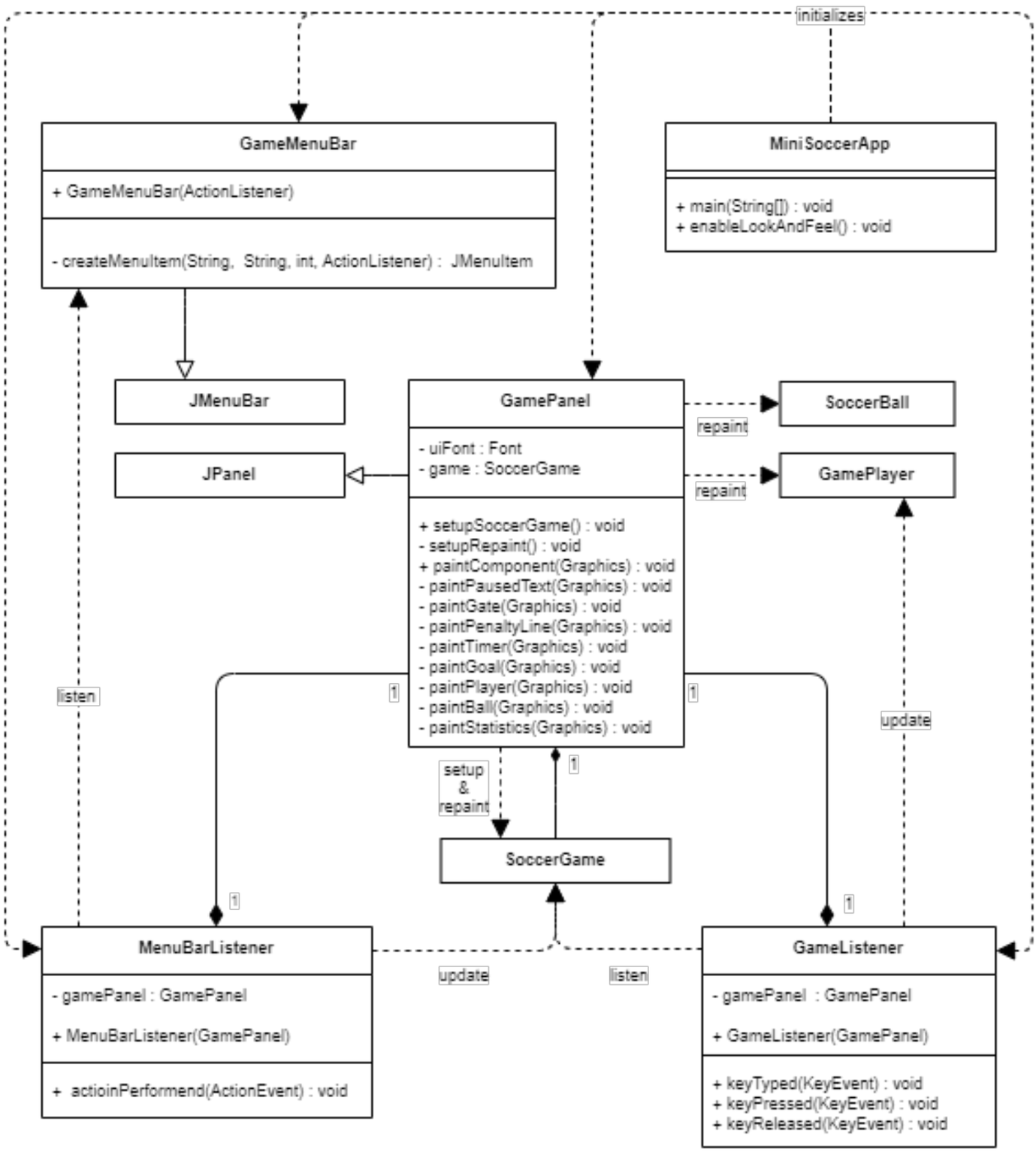

Figure 2: The class diagram of `SOCCERGAME`'s front end model

4 Classes

4.1 MiniSoccerApp

The main runner class which runs the whole project. It is the entry point to the whole project and as such, it initializes most of the other classes in preperation to run as expected. This is where the GUI is also constructed and displayed from.

4.2 GameMenuBar

This class represents the top portion of the view which has various controls such as resume, pause, new game, end game, and quit. This makes it easy for the user to control the application. To provide even more options for the user each of the menu bar optios have their respective keyboard shortcuts. The menu bar is where the state and flow of the game is managed by the user.

4.3 GamePanel

This class initiates the soccer game and gives user a visual representation of the game inside the GamePanel. All the visual components such as soccer ball, game players, soccer gate, etc. are managed and repainted by this class. This class also is composed of a SoccerGame class which helps it maintain the state of the game on the GamePanel. The repainting tasks of the dynamic elements such as the goalkeeper, the striker and the soccer ball are updated per a timer schedule. With these components this class is the main way to setup and repaint our soccergame whenever something changes.

4.4 GameListener

This is a control class which helps to listen for keyboard actions from the user. For this proect, this listener listens for the inputs of they keys left arrow, right arrow, up arrow, down arrow, and the space bar. This listener is used to update the model in response to the users key presses. To be more specific, the Striker is the part of the model this listener updates. So whenever the player position updates after a keypress, the timer task in the GamePanel will accordingly repaint the correct positions when it is time to run. In this way we can simulate the player moving accross the soccer feild in response to the users input. To update the active players this class is composed of a GamePanel with the instance of the soccer game. From the GamePanel we can get the instance of the game and get the active player and update their position accordingly.

4.5 MenuBarListener

This is a control class which connects model and view. This class gives the functionality of resume and paused buttons so when paused button is pressed the whole game is paused along with the timer and when resume button is pressed game resumes. there is also a new Game button which loads and starts a new game and also the ability to quit the application. This class listens to our GameMenuBar for these options and accordingly updates the model accordingly. For example, when a new game is selected, it repaints the GamePanel and model effectively.

5 Back end

5.1 Overview

In this software project we are using combination of factory design pattern and prototype design pattern. we have used combination of both in such a way that we are fulfilling the goal of factory design by hiding the creation of object. prototype design pattern is used here keeping performance in mind i.e. other players such as GoalKeeper and Striker doesnt have code repitition since they extend an abstract GamePlayer class which contains necessary common methods thus reducing code repitition and keeping up the DRY rule which is (Dont repeat yourslef).

5.2 Class Diagram

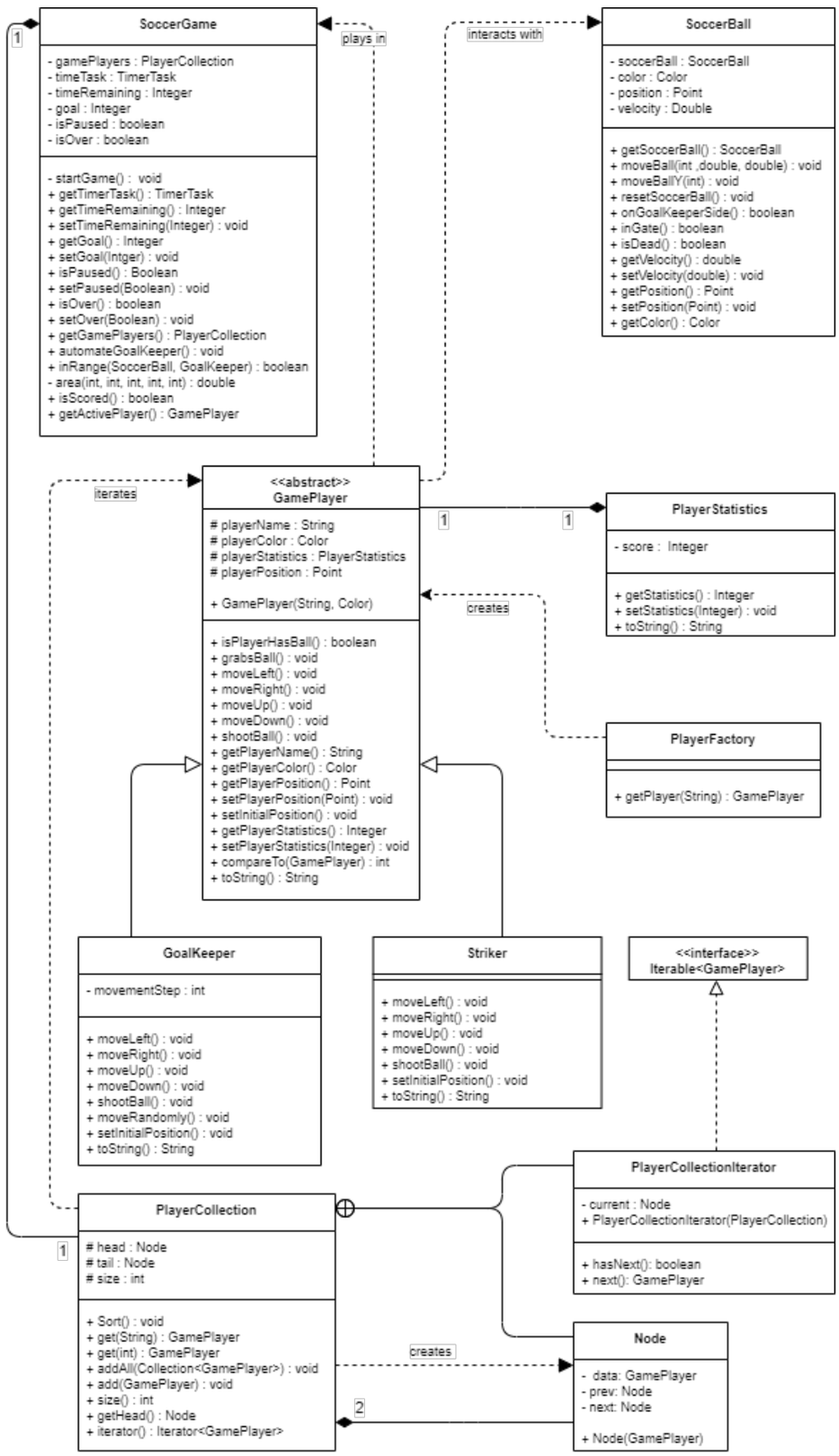

Figure 3: The class diagram of `SOCCERGAME`'s player model (example placeholder image)

6 Classes

6.1 GamePlayer

This is one of the important class required in this software project it is an abstract class and contains essential methods such as moveLeft, moveRight, moveUp, moveDown, shootBall. without the GamePlayer class there is no existence of two players the goalKeeper and striker without which the soccerApp has no meaning.

6.2 GoalKeeper

This is the class which fulfills the role of goalKeeper. it keeps record of number of saves through playerStatistics class. it has functionality of moving randomly left-right guarding the goalpost.

6.3 Striker

This is the class which fulfills the role of a striker. the purpose of moving near to the goalpost and scoring goals is what this class does with the help of methods inherited from its parent GamePlayer such as movingUP down left right and shooting the ball.

6.4 PlayerStatistics

This class contains the statistics such as score. It helps keep track of a players score using its score feild. Thus it has the necessary methods to manipulate the score.

6.5 PlayerCollection

This class is a custom data structure used to store a list of GamePlayer objects. It has functionalities such as sorting a list of players based on score, retreival of players, insertion of one or more players, and the abbility to use a custome iterator to iterate the list of players. This class uses inner classes Node, and PlayerCollectionIterator to acheive these functionalities.

6.6 PlayerCollectionIterator

This class is a custom iterator class used to help iterate our list of players in our PlayerCollection datastructure. It simply traverses from the head Node of the PlayerCollection and returns the next GamePlayer if it exists.

6.7 PlayerFactory

This class is a factory class that implements the factory design pattern and helps to create players by the player name. The only functionality for now is the abbility to create a GoalKeeper and a Striker with a predefined color.

6.8 SoccerGame

This class contains all the important aspects as it has the object notations of the other classes like PlayerCollection and it then calls the PlayerFactory class to further initiate the GamePlayer(goalkeeper and striker).

6.9 SoccerBall

This class represents the ball of the game and deals with the positioning of the ball when in play. It is a static Singleton class to make sure that we only have one instance of the ball at all times.

6.10 OOP Principles

Abstraction: In this software project, some unnecessary details are hidden from other classes. We will give you examples where abstraction is used. Take a look at the PlayerFactory class which is used to generate players such as goalkeeper and striker. The way object is created is completely hidden, what is done is a string is passed such as "goalkeeper" or "striker" and object is returned.

Encapsulation: In this software project, you will see the concept of encapsulation used where some logical implementation required for some classes to give required functionality is set to private so that other classes cannot access it. Examples such as GamePanel class has private paint methods such as paintPausedText(), paintGate(), paintPentaltyLine() etc as they are only required for the proper implementattion of the GamePanel class.

Inheritance: The concept of Inheritance is one of the most required principle used in this software project for better performance reasons. The GamePlayer class which is the parent class of goalKeeper and striker, doing this has enabled us to reduce code repititoin significantly. No child class even needed their own fields all they have is methods implemented which they inherited from GamePlayer class.

Polymorphism: In this software project the concept of polymorphism is also used. You will see in the PlayerFactory class that players are instantiated with the syntax GamePlayer goalKeeper = PlayerFactory.getPlayer("goalkeeper") which upon decoding means Goalkeeper goalkeeper = new GoalKeeper(); Polymorphism means "many forms" therefore though the object created is of the GamePlayer class but the methods takes form according to the new SpecificPlayer() where (SpecificPlayer- goalkeeper, striker) and whenever any method is invoked such as striker.shootBall()it uses the implementation of that specificPlayer class though the instant type is GamePlayer this happens because of inheritance where it is considering the latest form of the method. Therefore if it is GamePlayer goalkeeper = new GoalKeeper() and if goakKeeper.shootBall() is invoked this will call the latest implementation of shootBall() which is implemented in the GoalKeeper class.

6.11 Design Patterns

Singleton: The singleton design pattern is used for the SoccerBall class to maintain one instance of the gall throughout a soccer game instance. This assures us that we will always be manipulating one ball whenever the game players interact with the SoccerBall class. Along side the GamePlayer classes, the SoccerGame class also uses the SoccerBall class to manage some of the game functionalities such as `automateGoalKeeper()`.

Iterator: The iterator pattern is used in our PlayerCollection datastructure. More specifically, we use a custom inner class iterator called PlayerCollectionIterator which helps iterate over the list of players in our PlayerCollection. Because PlayerCollection implements Iterable, we have the ability to use java features such as enhanced for loops. This pattern is used wherever theses enhanced for loops are used or where the iterator is explicitly used.

Factory: The factory design pattern is used in the PlayerFactory class. This class helps us to create the GoalKeeper and Striker GamePlayer classes via their player name. This factory class is used by the SoccerGame class to initialize the goakKeeper and striker into a soccergame.

7 Implementation

7.1 Class Implementation

This projects after reading all the requirements and design steps was initializing a java gradle project using `gradle init` task, then customizig it to fit our project. After this, we started implementing the required classes in IntelliJ or Eclipse. Since our team used git we were able to work on different development environment. The IDEs in combination with gradle helped us compile and run our project. After the inital setup was completed, we iteratively started working on the classes while testing them. The key members in testing our project so we improved our coverage were Saniz, Akif and Hasnain. Using gradle helped to manage `jacoco` and `junit` dependencies so we can use the TDD (Test driven development). We compiled our java classes with java 16 with the help of our IDEs and the `gradle run` command. In this project, gradle is used it to manage the look and feel dependencies, create a `jacoco` coverage report, and provide an easy way to run the project. Git was used along with Github so our team could collaborate effectively (this came in handy since we were all in different time zones).

7.2 Player Collection

The player collection is a custom collection build using an interanal node and iterator class. The PlayerCollection class implements iterable so we can iterate over an internal doubly linked list of GamePlayer objects. This is where the iterator design pattern is used.

```
1 // custom Node of doubly linked list
2 class Node
3       GamePlayer data;
4       Node next;
5       Node prev;
6
7       Node(GamePlayer data)
8           this.data = data
```

7.3 Sorting

Within the player collectoin we use a simple bubble sorting algorithm to sort the linked list. This helps to sort the players by their statistics.

7.3.1 Pseudocode Code

```
1  // bubble sort
2    fun sort()
3       Node current, index
4       GamePlayer temp
5       if (head != null)
6           for (current = head; current.next != null; current = current.next)
7               for (index = current.next; index != null; index = index.next)
8                   if (current.data.compareTo(index.data) > 0)
9                       temp = current.data
10                      current.data = index.data
11                      index.data = temp
```

7.4 JavaDoc

Generating the JavaDoc is easily possible through the `gradle javadoc` task. This will generate the corresponding `index.html` for the projects documentation in the `build/docs` directory.

7.5 Jacoco

We use gradle to manage the Jacoco library dependency. We only test the model class as specified in the requirements. The tests covered each case and we managed to get 100% coverage.

7.6 Execution

Using github actions, we made sure that our build was dependent on our unit tests. On GitHub we have incorporated tags in the README with the build state and the coverage state. As of currently, we have 100% coverage with a passing build.

8 Tools

8.1 Source Code

IDE: IntelliJ IDEA 2021.2.1 and Eclipse, to implement Java classes
Documentation: vscode, LaTeX, for documentation (dessign report, etc.)

8.2 Build

Build: Gradle 7.2 (to manage dependencies and build)
Java Version: openjdk version "16.0.1" 2021-04-20
Version Control: Git with Github
Unit Testing: junit-jupiter-engine:5.8.1
Coverage: JaCoCo library, Codcov
Github Actions: used to neatly assert validity of tests before pull requests and pushes to main

9 Conclusion

This software project helped us to work as a group and also get an insight of how to collaborate while working on the same project. We learned how to work in team and coordinate with each other. This project also helps us understand the design principles and use them in an adequate way according to the project need. One bad thing was that we tried to work on the whole project at once and finish it up quickly. This turned out to be a bad idea. We stepped back a little and went through the code bit by bit and understood how the code is connected and what's missing. We also traced the code on a paper which helped us a lot. The things that I learned through the journey of creating this software project is initial planning, if it's a project which requires lot of demand to be fulfilled. I used divide and rule principle where I breakdown the demands of this project into subpart and tried to find what should be the role of that subpart in the project how many classes will be required and is there a way to implement code reusability by using OOP Principles. We also learned how to work thoroughly with github and collaborate when working on the same project. We learned design principles which were taught to us and how to implement them in our project. Another thing we learned is how to manage time efficiently as we were all in different time zones. The advantages of working as a group is that we get to learn concepts from various perspectives. This opens the way to go forward and choose the most efficient path to reach the final output. Disadvantages can be difficult to find a common time for collaborating, too many ideas to deal with, etc. Few recommendations for easy completion of project for upcoming students are that they should ask for help understanding the flow of code in the project, attend lab session and ask for help from TA, make sure that they understand the basic concepts of JAVA and also the software concepts taught in the class.

9.1 Roles

Setup: Amir
Setup the project on github to make sure the team could collaborate together.

Testing: Amir, Saniz
Created unit tests within the `ModelTest` class. These tests covered all the branch casses and thus achieved 100% coverage while testing the functionality of each components tested.

Uml: Akif, Hasnanin
Created the uml.

Report: Saniz, Akif, Hasnain
Created the report.