Mini-Soccer Game(Lab 5 Project)

Group Members:

Neel Rathod (214914329)

Kashyap Patel (216785339)

Ayush Sharma (217581075)

Oriana Quevedo (216685869)

Course: EECS 3311

Section: Section A&B

TA= Kazi Mridul, Naeiji Alireza

Date: November 03, 2021

Professor: Alvine Boaye Belle, Ph.D.

Part I: Introduction

The purpose of this software project is to successfully create a Java application using different design concepts such as UML that were taught in the class. The application will allow the user to play a simple soccer game. The user will be allowed to control the player in the game, which is striker and there will also be a goalkeeper, which will be automated to have it's own movement. The striker role is to score as many goals as possible in one minute and the goalkeeper will try to catch the ball before it enters the goal net and throw it back to the striker side. There is also a menu bar which has options to resume, pause or to start game.

There are two main goals to complete this software project, the first part of the project is to analyze the requirements of the project and based on that we will identify what OOP principles we can use to implement object-oriented software design. To make the software object oriented, we need to focus on 3 aspects: analyze the requirements of the project, design the project based on those requirements, and finally implement that design in actual software.

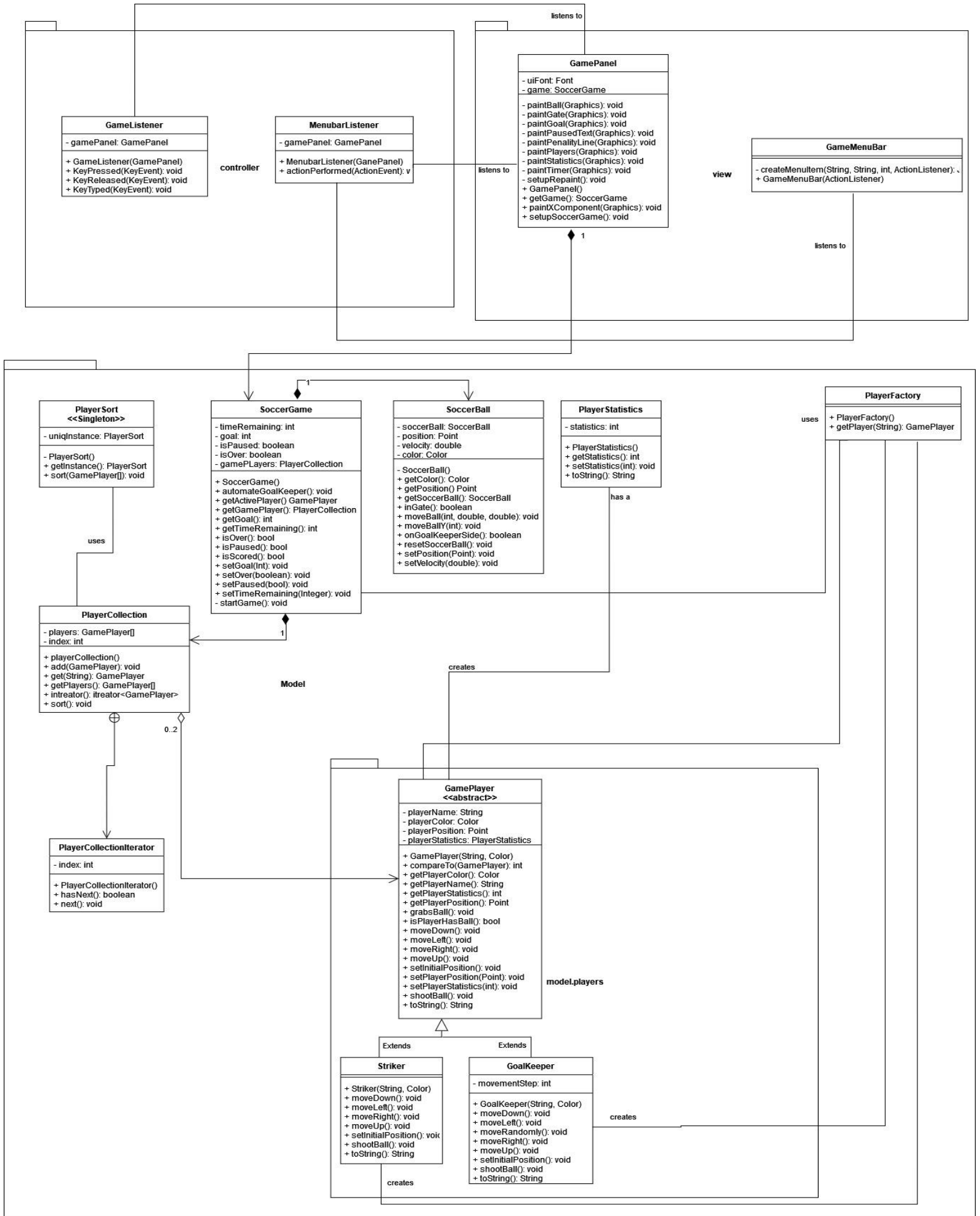
The main challenge with the project is to design the software in the model,view,controller (MVC) design. The separation of the MVC aspects of the code make it hard to design it in this way. However, once the design is finalized the implementation part gets easier since everything is outlined and clear.

The analysis of the project is very crucial as it helps us in finding and describing the object concepts that will be required to design the project. It will also help us specify software objects and the way they collaborate to satisfy the requirements.

The second aspect of OOD focuses on design of the project, to create the successful design, we will create UML diagram which will help us identity the main OOP principles (Abstraction, Inheritance, Polymorphism, Encapsulation) that are needed to make this project Object oriented. Usage of Encapsulation can be seen in all the classes by having private attributes and providing explicit getters and setters. Inheritance is shown by the GoalKeeper and Striker class, since these two classes inherit from a single abstract. Abstraction is shown by the presence of the SoccerGame and PlayerFactory class. The SoccerGame class abstracts the management of the various aspects of a game such as the players, the soccer ball, the state of the game etc. We used polymorphism in the structuring of the GamePlayer class.

The report will be structured to first show the requirements of the project and the analysis on what needs to be achieved for completion of the project. The second part of the project will focus on designing two UML class diagrams for the project based on the requirements and showing the relationships between the classes. The third part of the project will focus on implementing the project based on our UML class diagrams that we created in part 2 and highlighting the various software tools that we used to implement this project. The last part of the report will focus on sharing the learning aspects from the software project while providing some recommendations

Part II: Design of the solution



NOTE: To get a better view of the UML go to UML.jpg! MiniSoccerGameProject/UML.jpg.

The UML diagram above shows the various classes which comprise the software project.

We have utilized the MVC architecture for the design of our software.

view: Starting with the view Package, we have 2 classes in it. The *GamePanel* class and the *GameMenuClass.*

controller: Now in the controller Package we have 2 listener classes, namely *GameListener* and *MenuBarListener.*

model: In the model package, we have the *SoccerGame* class. The *SoccerGame* is composed of a *PlayerCollection* and a *SoccerBall*. We also have the *PlayerStatistics*, *PlayerFactory*, *PlayerSort* and the *PlayerClollectionIterator* classes. These classes help SoccerGame one way or another.

model.players: This is another package that holds our player types. It has the abstract *GamePlayer* class which has two concrete classes, namely, *Striker* and *Goalkeeper* classes.

Our UML class diagram uses 3 design patterns in total and they are as follows:

Factory Design Pattern: This pattern is shown by the *PlayerFactory* class which is used to instantiate new instances of *GamePlayer*. The main method for implementing this design pattern is the *getPlayer(String)* method which instantiates a new *GoalKeeper* or *Striker* based on the argument provided to it.

Singleton Design Pattern: This pattern is shown by the *PlayerSort* class, since this class does not need to have multiple instantiations of itself. This is implemented by having a hidden constructor and providing a method of obtaining an instance of *PlayerSort* and managing that instance through the *uniqInstance* attribute.

Iterator Design Pattern: This pattern is shown by the nested *PlayerCollectionIterator* class. This class implements the *Iterator* interface and provides implementation for the *hasNext()* and *next()* methods. Also, the *PlayerCollection* class itself also implements the *Iterable* interface.

The main OO design principles used are **Inheritance, Abstraction and Encapsulation**.

Inheritance: Inheritance is shown by the *GoalKeeper* and *Striker* class, since these two classes inherit from a single abstract *GamePlayer* class.

Abstraction: Abstraction is shown by the presence of the *SoccerGame* and *PlayerFactory* class. The *SoccerGame* class abstracts the management of the various aspects of a game such as the players, the soccer ball, the state of the game etc. Thus we ourselves don't have to manage the other classes, such as *PlayerCollection, SoccerBall*, and can use the interface provided by the *SoccerGame* class in order to manage them. Similarly, the *PlayerFactory* class abstracts the instantiation of various players. This can be seen in the usage of *PlayerFactory* in the *GamePanel,* which uses the *PlayerFactory*'s methods to instantiate player objects.

Encapsulation: Usage of Encapsulation can be seen in all the classes by having private attributes and providing explicit getters and setters. These getters and setters provide a consistent interface for retrieving and changing the attributes of a class. This effectively encapsulates the data within the class.

Polymorphism: We used polymorphism in the structuring of the *GamePlayer* class. Since this is an abstract class it is also the superclass of *Striker* and *GoalKeeper* classes. We can see this when we create an array of Gameplayer type which is populated by stickers and goalkeepers.

Part III: Implementation of the solution

In the **view** package we have:

The *GamePanel* class embodies the canvas on which the game is displayed. It is composed of a *SoccerGame* class and provides various methods of the form *paint*()* in order to paint the various components of the game such as the players, gate, statistics, etc. on the screen. This class also extends the *JPanel* class.

The second class is the *GameMenuClass* which is used to draw the main game menu which allows a player to stop, reset or resume a game. This class extends the *JMenuBar* class.

In **controller** we have:

GameListener and *MenuBarListener.* These classes listen for any events, like key presses, in the *GamePanel* and *GameMenubar* classes, respectively. The *MenubarListener* implements the *ActionListener* class and the *GameListener* class implements the *KeyListener* interface.

For **model** package we have:

The *SoccerGame* class controls the state of the game by its various fields such as *isPaused*, *timeRemainig, etc.* and provides methods in order to change said fields. The *SoccerGame* is composed of a *PlayerCollection* and a *SoccerBall*.

The *SoccerBall* represents a soccer ball in the game. It stores various attributes unique to a soccer ball such as the color and also stores the current state of the soccer ball like its velocity and position. The *SoccerBall* is used by the *SoccerGame* class.

The *PlayerCollection* represents a collection of players. This class also provides an iterator in order to iterate through the players. It is aggregated of at most 2 *GamePlayer* objects.

The *GamePlayer* class is an abstract class which represents a player in the game. This class provides implementation of various methods for other concrete subclasses, such as for moving the player, interacting with the ball, setting player statistics, etc. Each *GamePlayer* has a name and a color associated with it. The two concrete classes which extend the *GamePlayer* class are *Striker* and *GoalKeeper.* The *Striker* class represents a striker in the game, who tries to strike the *SoccerBall* into the gate*,* whereas the *GaolKeeper* class represents a goalkeeper, who tries to prevent the *SoccerBall* from getting into the gate.

The *PlayerFactory* class embodies the Factory design pattern and thus provides an interface for other classes, such as the *SoccerGame* class, to instantiate subclasses of the *GamePlayer* class.

The *PlayerSort* class embodies the Singleton design pattern, since multiple instances of this class are unresourceful. The main method of this class is the *sort* method which takes in a

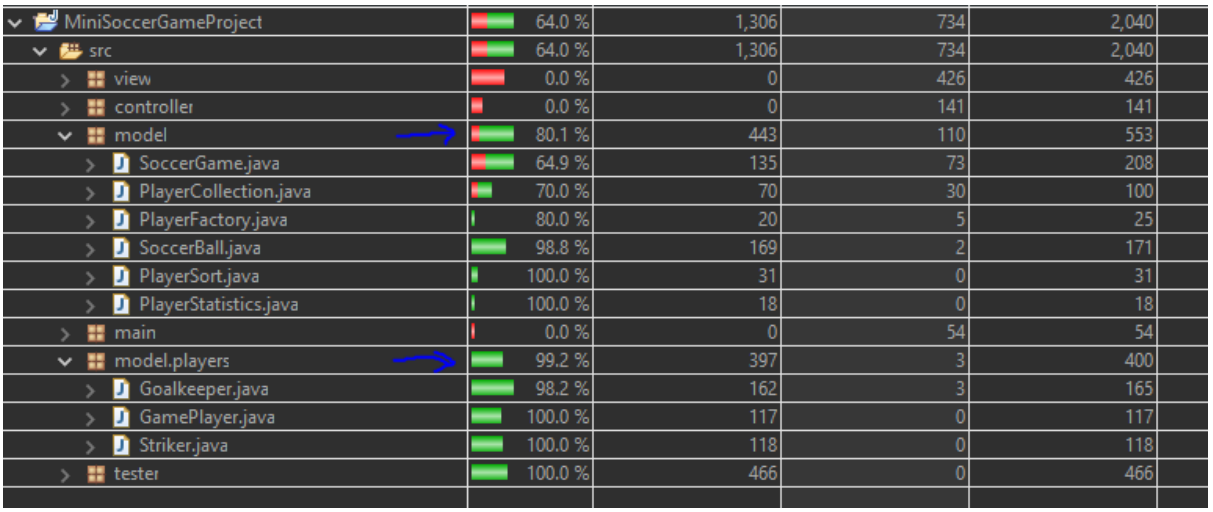
PlayerCollection and sorts the *PlayerCollection* based on which player has the highest statistic in descending order.

The *PlayerCollectionIterator* is a nested class of *PlayerCollection*. This class embodies the Iterator design pattern and provides an iterator for the *PlayerCollection* class. This class also implements the Iterator interface and thus provides implementation for methods such as *hasNext(), next()*.

The last class *PlayerStatistics* represents the game statistics of a *GamePlayer,* that is either the number of goals prevented by the *GaolKeeper* or the number of goals scored by the *Striker.* Each *GamePlayer* has its own *PlayerStatistics,* which is used by *GamePanel* class when displaying or manipulating the game statistics.

The **Javadoc** can be found under the MiniSoccerGameProject/src/index.html. The index.html file will give you the landing page of the Javadoc.

We have written **JUnit** tests to get over 80% coverage in the **model** and **model.players** packages. The JUnit tests can be found in the **tester** package, with the file ***MiniGameTester.java***.



MiniSoccerGameProject	64.0 %	1,306	734	2,040
src	64.0 %	1,306	734	2,040
view	0.0 %	0	426	426
controller	0.0 %	0	141	141
model	80.1 %	443	110	553
SoccerGame.java	64.9 %	135	73	208
PlayerCollection.java	70.0 %	70	30	100
PlayerFactory.java	80.0 %	20	5	25
SoccerBall.java	98.8 %	169	2	171
PlayerSort.java	100.0 %	31	0	31
PlayerStatistics.java	100.0 %	18	0	18
main	0.0 %	0	54	54
model.players	99.2 %	397	3	400
Goalkeeper.java	98.2 %	162	3	165
GamePlayer.java	100.0 %	117	0	117
Striker.java	100.0 %	118	0	118
tester	100.0 %	466	0	466

As it can be seen above, the Jacoco coverage is above 80%.

The **tools** that we used to create this project are: latest version of draw.io to create the UML class diagram, last version of Eclipse IDE, JDK & SDK version 17, git version control & Github.

The short informational video is labeled ***shortInformationalVideo.mp4,*** It gives all the basic information to run the application.

Part IV: Conclusion

The thing that went well while creating this software project was our combined prior experience of coding in Java. We have had exposure to the java language for quite a while, so that helped us how to tackle the project and how to implement the logic behind the project based on the requirements.

The things that went wrong while creating the software project were, creating the software design as the MVC model in the UML class diagram. As we had never gotten any kind of exposure to MVC UML designing before, so this was a learning curve for all of us. We tried a few things, failed; then tried again. The second challenge was working in groups, we had to use version control to the best of our abilities, we had to get used to **GitHub actions** as constantly merging our code manually got tedious. We learned how to work in teams well during this project.

The things that we have learned after completion of the project are creating different design patterns, as we use multiple design patterns for this project. We learned how to work in teams better and about version control.

Three things that we shall recommend to ease the completion of this project will be:

- Always create the project design beforehand and clearly state all the requirements rather than implementing the code first.
- Get a better grasp of UML design concepts and which to particularly use based on the project design and requirements.
- Learning more about GitHub Actions and version control better to have a smooth teamwork.

The following table denotes the different tasks performed by the group members.

Ayush Sharma	**Neel Rathod**	**Kashyap Patel**	**Oriana Quevedo**
Implementing the solution by writing several classes for the project.	Managing the project, by maintaining a line of contact between all the team members.	Implementation of the tester package and JUnit class.	Helping to finish the implementation of the project
Designing the first draft of the UML	Implementing Javadocs for all the classes.	Recording of the video and creating a jar and exe file.	Adding variables & methods to the UML diagram
Delegation of tasks and follow ups.	Finalizing the design of the UML	Polishing the code for final build and finalizing the report.	Making sure the final code is running properly.
Contribution to the report writing.	Contribution to the report writing.	Contribution to the report writing.	Contribution to the report writing.

All in all every member has put in great efforts towards the completion of this project and we are proud and content with the outcome.