Mini Soccer Game
EECS3311 F2021

Naeiji Alireza (naeiji@yorku.ca)
Gbemisola Akerele (Student# 216167041)
Domenico Calautti (Student# 216 374 530)
Nancy Hao (Student# 217120197)
Misato Shimizu (Student#215310246)

PART Ⅰ (Introduction)

What is the software project about and its goals?

This software project is about implementing a functional soccer game using Java graphical user interface (GUI) using *javax.swing* objects and components, and object-oriented programming (OOP) and object-oriented design (OOD), while using design patterns. The goal for this software project is to allow users to play a small and simple soccer game involving a shooter (named Striker) and a goalie (named Goalkeeper).

What are the challenges associated with the project?

Some challenges associated with the software project are:

- The project contains more files compared to the last lab project, which makes the whole project more complicated.
- As this is the first group project in the course, there needs to be clear plans beforehand among group members to complete the program efficiently.

What are the concepts we will use to carry out the project?

The OOD principles we will use are:

- Abstraction: This is utilized for Goalkeeper and Striker to inherit methods of movements from Gameplayer. Both Goalkeeper and Striker can each have unique movements by this principle.
- Inheritance: Goalkeeper and Striker inherit from GamePlayer so that each class owns its player's name, color, position and statistics.
- Encapsulation: The project mostly utilizes this principle to protect data in each class. For example, SoccerGame contains private variables, timeRemaining, goal, isPaused and isOver which implement getter methods to obtain value and setter methods to set values.

The design patterns we will use are:

- Singleton: SoccerBall shows this design pattern. The class has a private constructor, and static method, *getSoccerBall()*, which returns the single instance of the SoccerBall used throughout the software project. The instance is called by SoccerBall.getSoccerBall() when we instantiate an object, SoccerGame.
- Factory: With Goalkeeper and Striker implementing GamePlayer which is treated as an interface, PlayerFactory creates the player objects when SoccerGame is initialized.

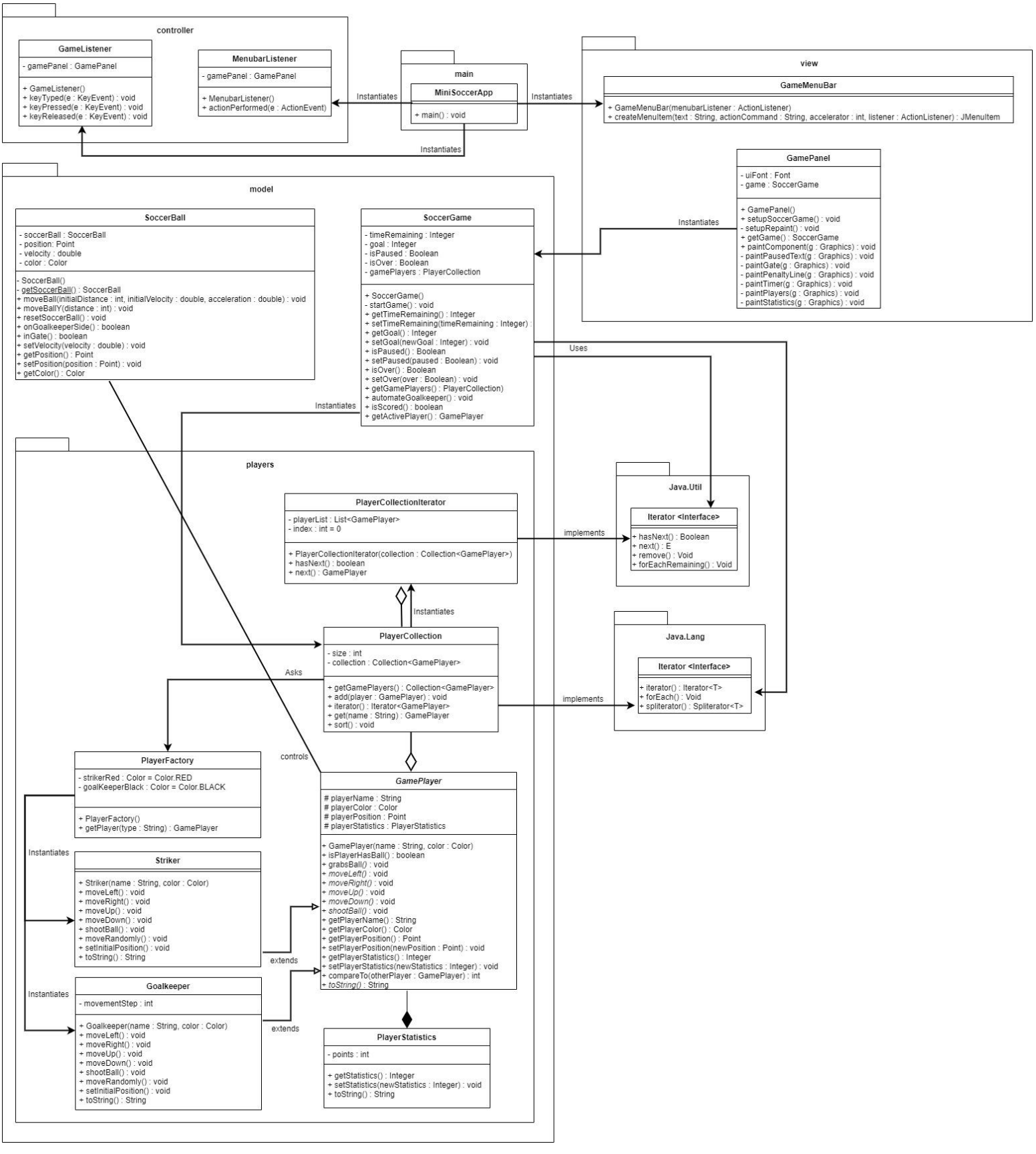
How are we going to structure our report?

Our report consists of 4 major sections, Introduction to introduce the project, Design to show utilized OO design patterns and principles, Implementation to describe the program in detail and Conclusion to sum up the project.

Gbemisola Akerele (Student# 216167041)
Domenico Calautti (Student# 216 374 530)
Nancy Hao (Student# 217120197)
Misato Shimizu (Student#215310246)

Mini Soccer Game
EECS3311 F2021
Naeiji Alireza (naeiji@yorku.ca)

PART Ⅱ (Design)



Mini Soccer Game
EECS3311 F2021
Naeiji Alireza (naeiji@yorku.ca)

Gbemisola Akerele (Student# 216167041)
Domenico Calautti (Student# 216 374 530)
Nancy Hao (Student# 217120197)
Misato Shimizu (Student#215310246)

How have we used design patterns?

Factory pattern containing classes:
SoccerGame: As the client that asks PlayerFactory to create a player.
PlayerFactory: instantiates a striker, or GoalKeeper.
GamePlayer: Abstract player class.
GoalKeeper and Striker: subclasses that extend GamePlayer.

Iterator pattern containing classes:
SoccerGame: As the client.
Iterator: As the iterator interface
PlayerCollectionIterator: As the concrete iterator that implements Iterator.
Iterable: As the interface defining the aggregate collection.
PlayerCollection: As the concrete aggregate that implements Iterable.

How have we used OO design principles in the class diagram?

In the class diagram, abstraction is seen as the fields of classes are private. The visibility of *private* does not allow the fields to be accessed by other classes that are not relevant to it. In GamePlayer, its fields are protected, meaning that the fields can be accessed by any class within the same package. There are setters and getters methods to set and access the value of the private fields.

In the class diagram, encapsulation is used to keep the data (methods, and fields that operate on the data) in one class bundled (private). This is so that there is restriction from accessing any data directly and allows access without revealing any complexity.

Inheritance is used as Striker and Goalkeeper inherit from their parent class of GamePlayer. The child classes can reuse fields and methods from their parent class.

Polymorphism can be seen in the inheritance of the concept of a GamePlayer. Both Striker and Goalkeeper have different customizations of the abstract methods declared inside GamePlayer. Polymorphism uses these methods to perform different operations.

PART Ⅲ (Implementation)

How have we implemented and compiled all the classes of the class diagram in Java?

Gbemisola Akerele (Student# 216167041)
Mini Soccer Game — Domenico Calautti (Student# 216 374 530)
EECS3311 F2021 — Nancy Hao (Student# 217120197)
Naeiji Alireza (naeiji@yorku.ca) — Misato Shimizu (Student#215310246)

First, MiniSoccerApp which is considered as a client instantiates the major setups such as GameListener, MenubarListener, GamePanel and GameMenuBar. As soon as GamePanel begins to structure the panel, SoccerGame will be instantiated with its default constructor. In the constructor, SoccerBall will be set by utilizing the Singleton pattern, PlayerFactory will create players, Goalkeeper and Striker, with GamePlayer as an interface (Factory pattern), and add them to a collection in PlayerCollection by using the features of PlayerCollectionIterator. Moreover, PlayerStatistics holds a number of points, and it can be obtained in each GamePlayer. Most classes such as GamePlayer, SoccerGame and SoccerBall contain their own getter and setter methods to protect their private data.

Regarding all the classes included in the class diagram

Main
- MiniSoccerApp: This is a client class which triggers the whole program.

Controller
- GameListener: This class implements KeyListener and cares for all the user actions in the game such as pressed keys.
- MenubarListener: This class implements ActionListener and controls behaviors for the menu bar on the top left of the game panel.

Model
- SoccerBall: This class utilizes the Singleton design pattern by holding a method to obtain an object of the class. This also controls the position and speed of the ball, and sets the color to white.
- SoccerGame: This class initializes a new soccer game and controls the whole game by getting and setting the remaining time and setting each game as continuing or finished. Depending on the remaining time and the status of pausing and scoring, the class specifies whether the keeper should keep moving or increment the goal count and pause.

Model.players
- GamePlayer: This class deals with game players in the game. The class includes global variables, playerName, playerColor, playerPosition and playerStatistics. Each variable has its own getter and setter methods, and we can manage the player's status such as whether the player is holding a ball.
- Goalkeeper: This class is inherited from GamePlayer for a goalkeeper's behaviors, and it manages the random movement of the keeper. A global variable, movementStep, is significant for where the keeper moves towards.
- PlayerCollection: This class holds a collection of players in the game, and we can add and get players in it.
- PlayerCollectionIterator: This class controls PlayerCollection by confirming the existence of the next player in a collection and obtaining players after the confirmation.

- PlayerFactory: This class produces players, Striker or Goalkeeper. It also randomly picks a color for each player from an existing list of colors.
- PlayerStatistics: This class deals with the amount of points for each player. A global variable, points, is private and has its own getter and setter methods.
- Striker: This class is inherited from GamePlayer for a striker's behaviors.

View

- GameMenuBar: This class deals with the interface of the menu bar on the top left of the game panel. The class inherits the JMenuBar class for some features as well.
- GamePanel: This class deals with the interface for the whole game in general such as paused text, gate, player line, timer text, goal count text, players, ball and player statistics. The class also inherits the features of JPanel.

Regarding JUnit tests

All the JUnit tests have been added for the model package with more than 80% of its coverage. The JUnit files contain tests for each class constructor and methods.

Regarding the tools/libraries we have used during the implementation

Eclipse 4.18.0 with JDK 15.0.1 is used for this project. Diagrams.net is used to illustrate the UML class diagram. Google Doc is used for the members to collaborate on the report. Github is used to host our software project.

A short video showing how to launch the application and run it

This video is included in the folder in GitHub.

PART Ⅳ **(Conclusion)**

What went well in the project?

Although this is a first group project in this course, we were able to discuss well to complete each task and make the program work without any issues. Utilizing a communication tool such as Discord and sharing this report and a class diagram on Google Docs and draw.io helped us to work on the project smoothly.

What went wrong in the project?

At first, we decided to use Replit to share code. However, not all members were familiar with the software, and we had to transfer all the added code to GitHub later.

Mini Soccer Game
EECS3311 F2021
Naeiji Alireza (naeiji@yorku.ca)

Gbemisola Akerele (Student# 216167041)
Domenico Calautti (Student# 216 374 530)
Nancy Hao (Student# 217120197)
Misato Shimizu (Student#215310246)

What have we learned from the project?

We have learned how we could share our code through GitHub by reading some articles and watching videos on how to effectively use GitHub for group projects.

What are the advantages and drawbacks of completing the lab in a group?

Having more members enables us to complete the program faster as we are able to combine everyone's knowledge. Moreover, examining other members' coding style allows us to learn something new, which is a significant factor for future coding. On the other hand, one of the drawbacks is that miscommunication among members might cause overwriting of other members' completed code.

What are our top three recommendations to ease the completion of the project?

The first recommendation is to communicate between members to maximize the amount of knowledge we can utilize. The second one is to divide tasks for each member so that the whole process of completing the project goes smoothly. The last one is to share useful resources among members such as online articles explaining how to start a group project on GitHub.

Indication of different tasks assigned to members

While each member worked on debugging code and modifying the report, the major tasks for each member are:

Gbemisola: Completed JUnit Test (SoccerBallTest, StrikerTest, GoalkeeperTest, PlayerStatisticsTest, PlayerFactoryTest and PlayerCollectionTest)
Domenico: Set up the GitHub environment, completed PlayerStatistics and worked on PlayerCollection and PlayerCollectionIterator
Nancy: Completed UML class diagram and worked on PlayerFactory, PlayerCollection and PlayerCollectionIterator
Misato: Created a video, modified a part of PlayerFactory and completed JUnit (SoccerGameTest, PlayerCollectionIteratorTest and GamePlayerTest)