Project: 2nd Software Project, Course: 3311

Name: Darshan Patel, ID: 217637638, Section: B, TA: Naeiji Alireza

Name: Jay Patel, ID: 217667841, Section: B, TA: Naeiji Alireza

Name: Rishikumar Patel, ID: 217620097, Section: B, TA: Naeiji Alireza

Name: Sagarkumar Patel, ID: 217637604, Section: B, TA: Naeiji Alireza

Introduction

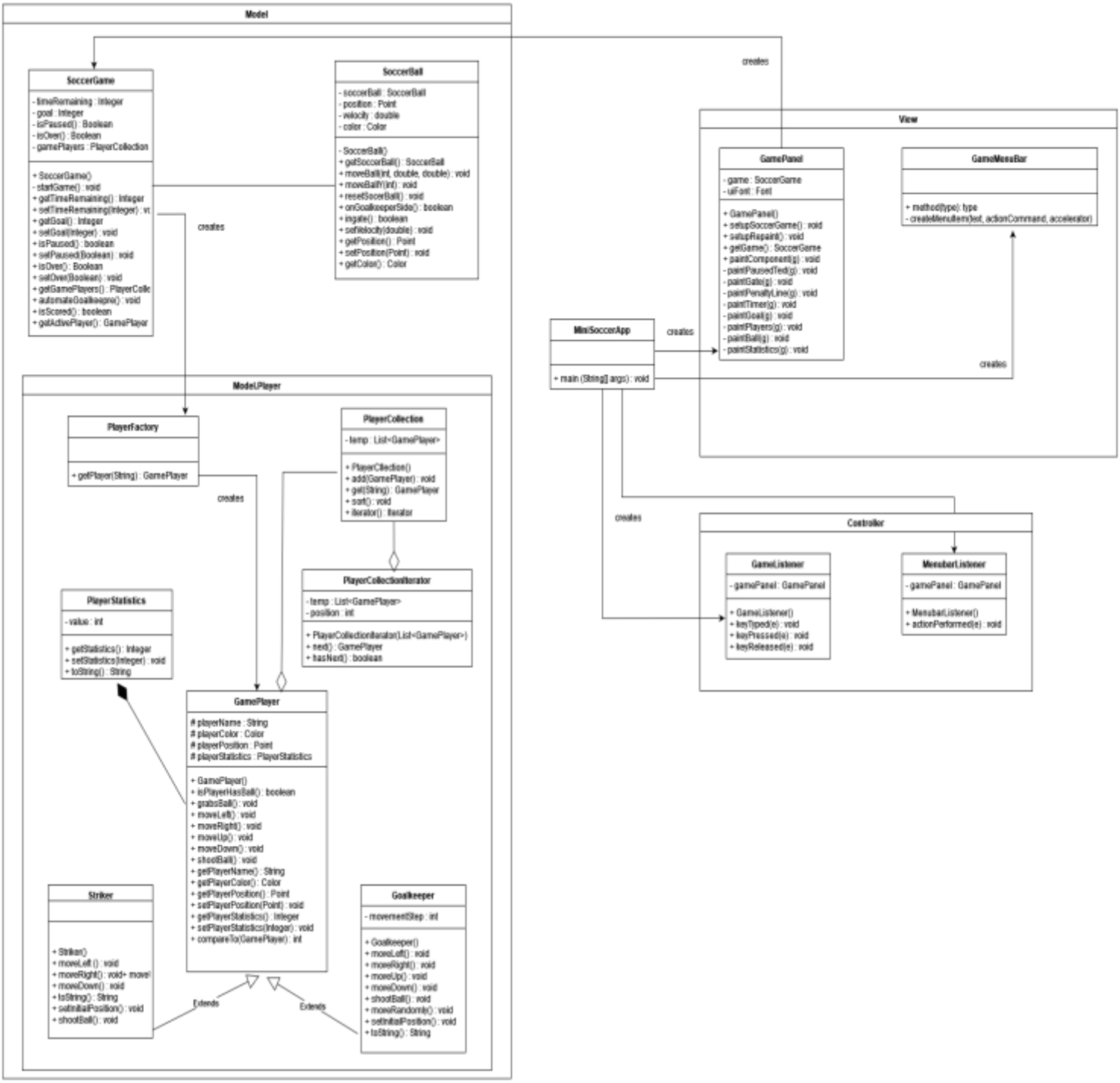
- The Software project is about a GUI(Graphical User Interface) Application regarding implementation of a mini soccer game. The game included a Striker and a Goalkeeper. The Striker can move in all directions and shoots the ball using the spacebar key. The Goalkeeper moves unidirectionally, precisely in a gaussian distribution fashion. The game ends when the timer runs out (60 seconds). It also pauses when Striker manages to score a goal. When the game is finished, it displays the statistics for each player. If the striker fails to score a goal and given that the ball is in the goalkeeper area, the goalkeeper shoots back the ball.

- The two main challenges we faced during this project are implementing the PlayerCollectionIterator class since none of us have used the Iterator class until now. And even writing the JUnit tests was something new to us.

- For this software project, we will be using the concepts of OOD as described below:
    - **Abstraction** - To hide unnecessary details of a class from another class.
    - **Encapsulation** - Keeping the states and methods of an object inside a class.
    - **Inheritance** - Letting a child class use the methods and variables of the parent class.
    - **Polymorphism** - Passing different parameters to a method to have it perform different things.

- The structure of the report is in accordance with the PDF handed out by the professor in the Mini-Soccer game lab manual.

Design of Solution



Design patterns used:

The iterator design pattern is used to create an instance of the PlayerCollection class. GamePlayer class is a corresponding class since it shares an aggregation relationship with the PlayerCollection class.

The project is designed with the idea of MVC (model view controller) design. The SoccerGame class creates a PlayerFactory instance. Which in turn, creates the players (Striker and Goalkeeper) using factory design pattern.

OO design principles used:

- Abstraction : The GamePlayer class uses the principle of abstraction. It has several abstract methods which are implemented in the Goalkeeper and Striker classes. The same can be achieved through the use of interface too.
- Encapsulation : As I mentioned in the introduction, several methods in several classes have been privatized or set as protected in order to hide unnecessary information. The getters and setters have been used to access the data for such classes. For example, the attributes of the GamePanel class are kept private just like many of the methods.
- Inheritance : The Striker is a GamePlayer. And, the Goalkeeper is a GamePlayer. Therefore, the Striker and the Goalkeeper classes inherit from the GamePlayer class. Which means that the methods and attributes of the GamePlayer (parent class) are used and implemented by Striker and Goalkeeper (child classes).
- Polymorphism : The only instance of polymorphism in this project is when the Striker and Goalkeeper classes implement the abstract methods in the GamePlayer classes according to their needs.

This project heavily relies on factory design pattern and the OO principles as elaborated above.

Implementation of the solution

Since part of the code was already given, we were only supposed to implement 4 classes. Firstly, the PlayerCollection consists of GamePlayer instances. Therefore, they share aggregation relationship. PlayerCollection class consists of add and get methods which are implemented accordingly. It also has an iterator method which calls the PlayerCollectionIterator class. Also, PlayerCollection and PlayerCollectionIterator classes share aggregation relationship. PlayerCollectionIterator class consists of a constructor method, and methods to determine if there is a player in the collection at the position next to the current position, and also return it if needed. The PlayerFactory class uses the factory design pattern to create a Striker or a Goalkeeper, basically a GamePlayer. The PlayerStatistics class has getter and setter methods to update and access a player's statistics. It also implements toString() methods, without which the summary could not be displayed.

The MiniSoccerAppclass (main class), upon running it, creates instances of GamePanel, GameMenuBar, GameListener and MenubarListener classes. The GamePanel class further instantiates SoccerGame class. Moving further, the SoccerGame class creates GamePlayer instances using the PlayerFactory class. Also, SoccerGame shares a relationship with SoccerBall. The players created are added to a collection using the PlayerCollection class. If required, the PlayerCollectionIterator can be used too. The statistics of the players are updated and stored using the PlayerStatistics class.

This is actually the first time we made use of JUnit testing. The tests are not highly effective, but the coverage is more than expected. We did not use Jacoco, instead the in-built coverage feature of Eclipse was used.

We used the following tools/libraries:

- Eclipse (Version: 4.21.0)
- JRE [JavaSE-15]
- JDK 15
- JUnit 5

Conclusion

Implementing PlayerFactory and PlayerStatistics was not troublesome. As I have used javax.swing components in the previous project, it was easier for me to comprehend and implement classes.

When I managed to get the game working, I realized that the summary was not displayed as per the expectation. I traced the code intently, adn figured that the toString() method needed to be implemented. PlayerCollection and PlayerCollectionIterator were a bit challenging.

To use JUnit testing. To figure out the UML for quite many classes. Also, none of us have been avid users of Javadoc, but for a change this time, we used it heavily!

The main advantage of working in a group is to have the workload distributed. And getting different ideas/opinions/comments/views/perspectives/questions from different people. When it comes to drawbacks, my mind goes null.

Top three recommendations:

- To use OOD and OOP principles.
- Learn new things from the internet
- Start like an early bird and regret less later!

Rishi and Darshan mainly focused on implementing the missing classes. Sagar and Jay figured out the UML. Together, this feat was achieved. Overall, working in groups was a good experience.