EECS 3311 lab 05
Mini Soccer Game App

Name: Ozan Isik
Github username: ozan98
Student Id: 215001746

Name: Nofal Rizwan
Github username: NofalRiz26
Student ID:213827720

Name: Enes Bahadir Is
Github username: eJustIS-1
Student ID: 216164337

Name: Ugur Can Avcu
Github username: ugurcanavcu
Student ID: 215814239

Professor Alvine Boayle Belle

Introduction

The software project is a mini soccer game that consists of a striker, goalkeeper, and ball. The soccer game is written in Java 11 using Java Swing which is a GUI framework for Java. The interface of the game comprises two menus: Game and Control. The Game menu is used to start a new game or exit the game, whereas the Control menu's purpose is to resume or pause the game. On the left corner of the interface, there is the time that starts at 60 seconds at the beginning of the game and counts down until it reaches 0 in which the game ends. The interface displays the number of goals scored by the striker just below the time.

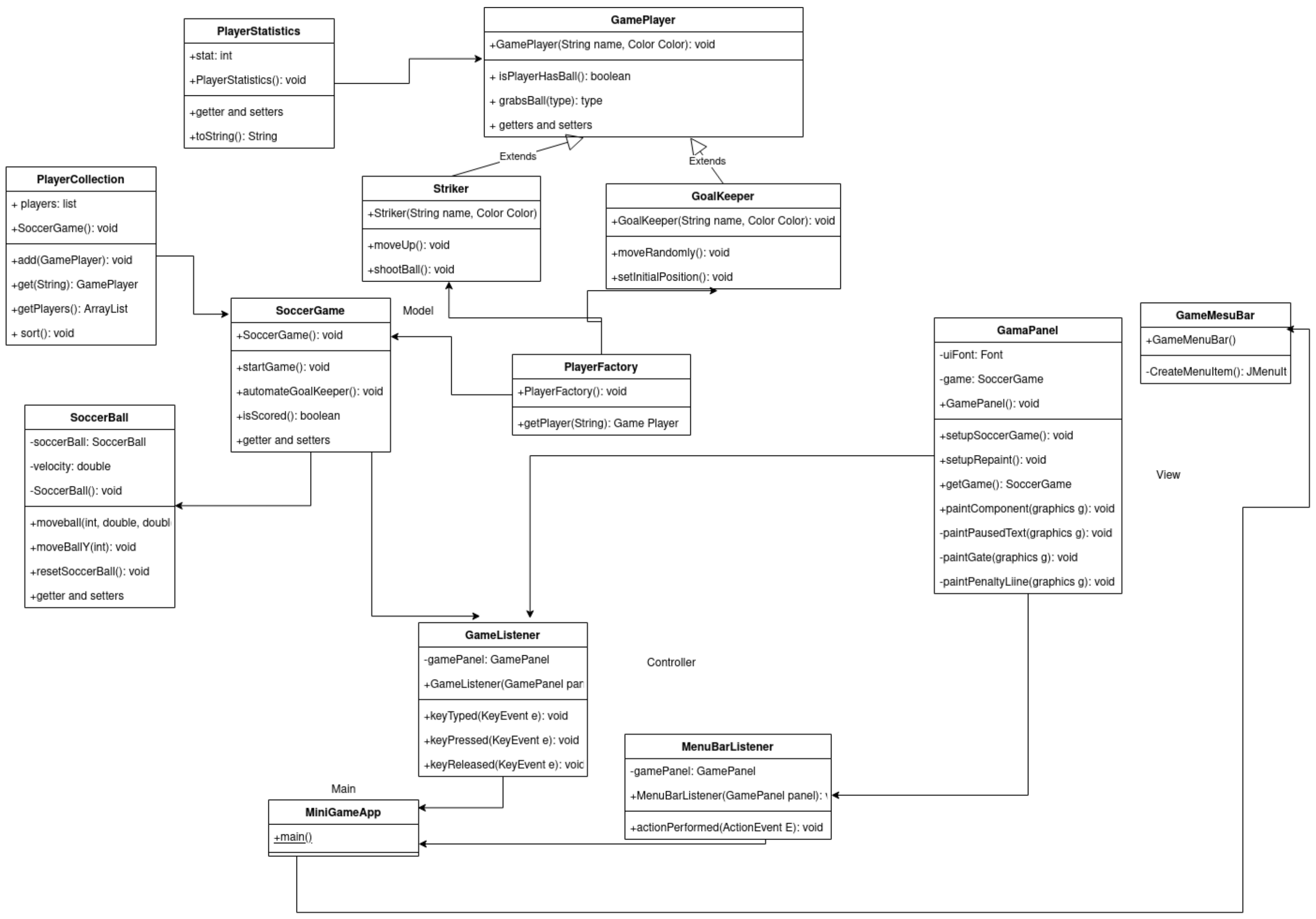
The striker shoots the ball to score a goal during the game, while the goalkeeper tries to catch the ball. Arrow keys are used to move the striker, but the goalkeeper moves randomly left-to-right or right-to-left in front of the Gate which is the area under the post. The space bar is used to make the striker shoot the ball. There is also a penalty line between goalkeeper and striker, and both goalkeeper and striker cannot cross this line. In case the striker shoots the ball in the Gate area, the goal count on the interfaces gets incremented by one, the game will be paused, and the position of the striker, goalkeeper, and ball will be reset. If the ball is caught by the goalkeeper or the ball lands on the side of the penalty line which is closer to the gate, then the goalkeeper kicks the ball back to the striker side of the penalty line and the number of the balls caught by the goalkeeper gets incremented by one. When the game is over, information regarding how many goal striker scored or how many balls the goalkeeper caught will be displayed in the middle of the interface.

The challenge associated with the software project was to analyze the different classes and methods and figure out what depends on what since the project is quite big. A significant amount of time was spent on this part of the project.

In this software project, iterator, factory, and singleton design patterns along with polymorphism, inheritance, abstraction and encapsulation design principles are used.

The report of the project will comprise four parts which are introduction, design, implementation, and conclusion, respectively.

Design



The two design patterns we used for our UML diagram were Factory design and MVC design pattern. For the behavioural aspect of the project we have added an iterator pattern that lets you traverse elements of a collection without exposing its underlying representation. The project follows a MVC model where the Model, View and Controller classes are interlinked with each other which provides more organization to the code and the plan of the project. The SoccerBall and SoccerGame classes comprise the model which dictates the behaviour of how the game is played. In addition to that our players, Striker and Goalkeeper, are created through the Factory design pattern. This implementation is done around the Factory class, PlayerFactory, which also connects to the SoccerGame class. There is a main class called MiniSoccerApp which instantiates objects for the model, controller and view. The controller binds the model and view together and is represented by the GameListener and MenuBarlistener classes. The view provides the user interface for our games and is controlled by the GameMenuBar and GamePanel classes.

In this project, we have utilized some of the Object Oriented Design Principles for our application such as polymorphism, inheritance, abstraction and encapsulation. Firstly, inheritance is used in the GamePlayer class. GoalKeeper and Striker classes inherit some of the methods and attributes from the GamePlayer class in order to reuse these information again. There is no need to rewrite the mutual methods and attributes, so the design uses Inheritance (generalization) design principle for solving this issue. Secondly, Another design principle that is used in the project is abstraction. Abstraction design principle helps the developer to show only essential information to users. This principle is used in our classes to hide important information with some techniques such as making attributes private instead of public. Thirdly, polymorphism is used to override some of the inherited methods from other interfaces. For example, we have used polymorphism in the GameListener class in order to override some of the methods from KeyListener Interface. Lastly, we have also utilized one other object oriented design in our project which is encapsulation. We have used this design principle in order to prevent direct access to our classes which was important for us to hide our important data from users and also the main information cannot be changed or converted to something else, thanks to this design principle.

Implementation

The design pattern used for the implementation is the MVC design pattern, where functionality of the program is organized into different modules and interactions between modules is done in a single module. The implementation of this lab starts in the MiniSoccerApp class which is the main class of the whole program. The MiniSoccerApp class creates two objects of GameListerner class and MenuBarListener class. These two classes are the controller modules of the MVC pattern that handle requests from the main class and request services from the model module and view module. GameListener class keeps track of the status of the game and is able to move the players based on the status of the game. It does this by making requests to the SoccerGame class which is part of the model module. These requests can be for example checking if the game is paused or if the game has ended. If these requests are false then the controller can move the player. SoccerGame class is part of the model module along with SoccerBall, GamePlayer, GoalKeeper, Striker, PlayerCollection, and PlayerFactory that handle all the logical functions of the program. The SoccerGame class will start the game after being created by the controller GameListerner with the startGame() method. However, before starting the game, SoccerGame will create players from the PlayerFactory class and store them in an arraylist from the PlayerCollection class. In the PlayerFactory class players are created using the Striker and GoalKeeper class that both inherit from the GamePlayer class. GamePlayer class, the parent class of Striker and GoalKeeper, uses the PlayerStatistics class to hold statistical info on the player. The SoccerGame class will also create a soccer ball using the SoccerBall class that handles all the functionality of the soccer. In the model module, SoccerGame holds all the important logical information that the controller can request upon. The other controller class, MenuBarListener, handles the menu bar depending on the status of the game which it receives from the GamePanel class. GamePanel alongside the GameMenuBar class are part of the view module that handles the interface of the program. The implementation that we had to incorporate to the program was to complete the model module. In the model. In the model module we added the classes PlayerCollection, PlayerFactory and PlayerStatistics. The purpose of PlayerCollection is to store a collection of game players. This is done with using an ArrayList storing GamePlayer objects in PlayerCollections. GamePlayer is the parent class of Striker and GoalKeep class. Thus we are storing a collection of Striker and GoalKeeper objects. PlayerCollection provides methods to the SoccerGame class like add() to add players, sort() to sort the players in collection, get() to get a specific player based on name and getPlayers() to get the whole collection of players. SoccerGame class uses these methods accordingly to set up the game for the controller. The purpose of PlayerFactory is to create players. This is done using the Factory design pattern. PlayerFactory creates players by using the parent class GamePlayer and creates Striker and GoalKeeper through the OO principle of dynamic linking. PlayerFactory also provides the method getPlayer() to return a created player based on the requested name. This method is used by SoccerGame to get a created player to be then added to PlayerCollection using the add() method of PlayerCollection. The purpose of PlayerStatistics is to store info on a specific player. In PlayerStatistics we store the statistical value in an Integer object. PlayerStatistic provides methods to GamePlayer like setStatistics() to store a value and getStatistics() to retrieve the statistical value of that player. We thought it was unnecessary to include a PlayerCollectionIterator because we implemented PlayerCollection with an ArrayList. Therefore we can use the ArrayList iterator to iterate a collection of GamePlayer. We agreed that using an ArrayList iterator would be the best way to implement this as it makes it easy and it's more space efficient as we are not creating another class. Throughout the implementation we made sure to follow the MVC pattern. This includes every interaction between modules is done through the controller classes and that model and view do not interact with each other directly. This is also evident in the UML diagram.

The tools used to implement this program were using VS Code with JDK 11.0.11. All the necessary information regarding the classes and methods are documented in the code.

Conclusion

After figuring out what depends on what, implementation part of the project got easier. Implementation part went considerably well. As it is mentioned earlier in the report, figuring out the relationship between classes and how the missing model classes should behave was the most time-consuming part of the project. In addition to that, we spent a considerable amount of time to decide on the design pattern that we need to use which should be suitable and efficient. This project was extremely helpful and effective since it forced us to analyze and understand(design) the whole project before we start implementing it.

The main advantage of the completing the lab in a group was to have different ideas on the design and implementation part of the project which eventually leads us to come up with a better product at the end of the process. The challenging part of completing the lab in a group was finding a time frame in which each group member was available.

We believe that understanding/analyzing and designing the project before implementing is essential. It helps you to find best design pattern that you can use on the project and makes the implementation part easier. Finding most convenient time for each group member is crucial as well since it eventually affects the productivity and creativity of the group members. The last recommendation we can make is that we would not rush to try finishing the project, on the contrary it is always better to do in depth analysis before starting the project.

Even though everyone was involved in the project, each group member had an assigned part of the project that needs to be done. Ugur worked on the report, specifically introduction and conclusion parts, whereas Ozan especially worked on implementation and testing part of the project. Enes and Nofal were assigned to get UML diagram done. Therefore, each group member was collaborative.