EECS 3311 Section B 2021 Fall

Project2

Mini Soccer Game

Group 7

Boho Kim 217303033

Saad Shahid 216178477

Yixing Cao 216775140

Yun Lin 214563449

PART-I

1. This software project is all about creating an application that displays an interface with two menus: Game and Control.
    - The Control menu allows pausing or resuming the Game.
    - Game menu allows starting a new game or exiting the game.

    Here are the goals of this software project:

    - The interface should display two game players: a striker and a goalkeeper. The goalkeeper should be in front of the Gate. During the game, the striker should try several times to shoot the ball to score a goal and, each time, the goalkeeper attempts to catch the ball.
    - The goalkeeper should move randomly on the left or on the right of the gate to try to catch the ball. The interface should display the time remaining time (in seconds) in the game, and the number of goals scored by the striker.
    - The remaining time starts at 60 seconds. That time decreases during the game. When it reaches 0 second, then the game should automatically end.
    - The application should be able to pause the game, resume the game, and start a new game. The initial game score is zero.

2. Challenges associated to the software project:
    - Finding relevant/best resources for learning technologies and enhancing skills.
    - Designing the roadmap/design (like UML class diagram) of the complete project.
    - Implementation of the class diagram properly with complete coverage.
    - Testing the correctness and complete working of the project.
    - Meeting the expectations and complete requirements of the project.

3. We will use the below concepts while working in this project:

- **Object/Class:** For representing real world thing (player, ball) in the software project.
- **Inheritance:** Inheritance will embody the generic concept of GamePlayer (GamePlayer is an abstract class) so that all other classes like GoalKeeper and Striker can use it.
- **Encapsulation:** We will bundle methods and attributes related to a particular Player or Ball class in its class so that it could hide the internal representation/state of an object from the outside.
- **Class Diagram:** For describing the structure (attributes and methods of a particular class), and maintaining relationships between classes of the project/system.
- **Design Pattern:** It will help in analysing and designing the system with reusability and transparency. It will also help in clarifying the system architecture for building a better system/project solution.

4. Our reports will be structured in this way:

- Requirements Analysis
- Designing the system
- Implementation
- Testing and Validation.

PART- II

- Singleton Pattern
  There can only be a single ball in a soccer game, therefore it is fitting to have the SoccerBall class be a singleton. It follows the implementation for the pattern: the single instance is a static final object that cannot have another instance created, the constructor for the class is private, and it has the public static method getSoccerBall() that returns the singleton instance.

- Factory Pattern
  The Factory pattern is also implemented in our project. SoccerGame class is dependent on PlayerFactory class to instantiate a GamePlayer object. Within the SoccerGame class, the constructor specifies to the PlayerFactory class which instance of the type of GamePlayer object is needed. It call to instantiate a Striker object and a Goalkeeper object.

- Iterator pattern
  The iterator interface GamePlayer acts as the Iterator and the Aggregate component of the pattern. It is an interface that allows access and traversal of the aggregate. The interface also defines the creation of the Player object. PlayerCollectionIterator class acts as the Concrete-Iterator component, implementing GamePlayer iterator interface. PlayerCollection class acts as the Concrete-Aggregate component of the pattern. It also implements the GamePlayer interface.

- Design Principles
  The OOP principle of inheritance is used within this project. Classes like GoalKeeper and Striker inherit all the attributes and methods from their parent abstract class GamePlayer. Polymorphism is another OOP principle used in the project. GoalKeeper and Striker class inherited the abstract methods for movement from GamePlayer. These classes have their own implementation of the same methods. A Goalkeeper object moves randomly in a gaussian distribution pattern, whereas a Striker object moves five pixels in every direction using the arrow keys. Encapsulation is another OOP principle used throughout. For example, the SoccerBall class restricts direct access to private attributes such as position and velocity. We included getters and setter methods to have access to them. SoccerGame class also has restricted access to its private attributes, but also has public methods to gain access.

PART – III

1. Our Github repository with full implementation:
   https://github.com/kim-boho/MiniSoccorGame-3311-Project2-.git

2. With given project, we needed to implement PlayerStatistics, PlayerFactory, PlayerCollection and PlayerCollectionIterator class. Other classes were given.

   - **MainSoccerApp**: This class is to run an application. It only has a main method and this method uses GamePanel and GameMenuBar for an interface, and Listener classes to get input.

   - **GameListener**: It implements KeyListener interface. It is used to obtain input from the keyboard by the user while a soccer game is running.

   - **MenubarListener**: It implements ActionListener interface. It is used to obtain input from the menu bar by the user's mouse while the soccer game is running.

   - **SoccerBall**: This creates a soccer ball for the game. While running game, only one ball exists. So, it has a SoccerBall which is static and final as a field. It controls the position and moving of the ball using methods.

   - **SoccerGame**: It is a class to control the whole game. It is represented on a panel that consists of GamePanel and GameMenuBar, and is manipulated by Listener classes (GameListener, MenubarListener). So, it has only one ball and player collection per game. Players are created using PlayerFactory and are stored in the collection class.

   - **GamePlayer**: This is an abstract class for the player object. It is a parent class of Goalkeeper and Striker. So, the unimplemented method is defined in the child's class. It has methods to move an object, and basic getter and setter. Also, since it implements Comparable, objects can be compared using compareTo method. It is created by PlayerFactory class. Created players are added to players' collection in SoccerGame class. Each player has one statistics class to store their score.

   - **Goalkeeper**: This is a child class of GamePlayer. It extends all properties of the GamePlayer class, and has its own property such as moveRandomly() and shootBall() to function as a goalkeeper in the game.

   - **Striker**: This is a child class of GamePlayer. It extends all properties of the GamePlayer class, and has its own property such as shootBall() to function as a striker in the game.

- **GameMenuBar**: This is for the interface. It organizes graphic elements that are needed for the menu bar in the interface. It is used by the main method in MiniSoccerApp.

- **GamePanel**: This is for the interface. It organizes graphic elements that are needed for a soccer game in the interface. It is used by the main method in MiniSoccerApp.

- **PlayerFactory**: It is used to create GamePlayer objects in an application. Using Factory pattern design, it recognizes the player's type by input (String) and creates a proper type of game player object. SoccorGame class that manages the game uses PlayerFactory to create GamePlayer objects.

- **PlayerStatistics**: It is used to store a game player's score. The score is accessed using getter and setter. GamePlayer objects are sorted based on score.

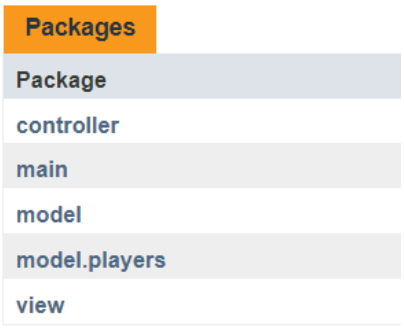
- **PlayerCollection**: It is used to collect GamePlayer objects. Players can be added and obtained using getter and setter. Sort method sorts players based on their score in descending order. Iterator method generates Iterator using PlayerCollectionIterator class. It is used by SoccorGame class to manage GamePlayer objects.

- **PlayerCollectionIterator**: It is for iterator for PlayerCollection. Such as another iterator, it has hasNext() and next() methods. (We did not add remove method because it is not used in this project) It implements the Iterator interface and provides Iterator for PlayerCollection class.

- **ModelTester**: This is a class for the Junit test of model & model.players package.

  (MenubarListener: While running the application, we found a bug related to golakeeper's moving. When we start a new game without pausing, goalkeeper keeps moving randomly. It is supposed to be set as the initial position and stop moving. So, we added a constraint not to start a new game without pausing. It shows messages "You should pause the game first to start a new game." when starting a new game without pausing.)

3. Javadoc
   In our repository, Javadoc is available in 'MiniSoccorGame-3311-Project2-/doc/index.html'.
   You can see specific class explanation in index.html.



4. ModelTester class in Model package

   1) **playertest():** Test for PlayerCollection, PlayerFactory, PlayerStatistics PlayerCollectionIterator. Since, the creation of players using the PlayerFactory, the method test for different parameter string passing to getPlayer method. And then, added to player collection by playercollection.add method. Test for the return Boolean for add method. And the combine with playercollection.get method, while add method return true, the get method will return added player. In addition test for setPlayerStatistics and getPlayerStatistics() with playercollection.sort, here the test method first set players' Statistics in ascending order, then sort, after that get Statistics by calling using iterator of playercollection.
   2) **moveTest()**: This is a test for moving of objects. Simply, it tests players moving to up, down, left and right, respectively. Both of striker and goalkeeper are tested. For the goalkeeper, its random moving is also tested. Store the initial x coordinate of goalkeeper and compare it to new x coordinate after random moving.
   3) **soccerTest()**: This is a test for SoccorGame class. First, check whether object is created well. After creating object, check remained time. It should not be larger than 60. It tests basic getter and setter. Using getter, it tests whether players who are striker and goalkeeper were created in the class. It also tests whether the game ball is working well. Set the ball goalkeeper's position. Then, goalkeeper should catch it and get one score for blocking. So, after it, check the score.

5. Eclipse IDE, JRE JavaSE-15, Junit 5, JaCoCo, draw.io, Github.

6. Short video to show how to launch and run application:
   https://drive.google.com/file/d/1haxvAC8VYmFzLyKXrKtC_VnTY36CgkNQ/view?usp=sharing

PART – IV

1. Below are the things went well in this software project.
   - UML class diagram
   - Implementation of player classes like GamePlayer, Striker, and GoalKeeper.
   - Setting timer for the soccer game using Timer class in Java.
   - Testing and Validation using Jacoco for reaching out maximum possible coverage.
   - Designing and Implementation of the complete software project using Abstract Factory Design Pattern.

2. Actually, for the final delivery of project work, everything worked well. But, while implementing the project, we faced little difficulties in implementing Factory Design Pattern for the classes.
   Also, understanding the given project code by the professor, consumed good amount of time because understanding the relationship between the classes/files is always important before starting any project.

3. We learnt about below things while implementing this project:

- Got good confidence in implementing classes and objects.
- How to design the software project based on the given requirements.
- How to draw UML class diagram for structuring the classes and maintaining the relationships between classes.
- How to implement Factory design patterns for the different classes.
- How to perform JUnit Testing on the various modules of the project.
- How to use Jacoco for improving and reaching out maximum possible code coverage.

4. Advantages of completing the lab in group:

   - Work Pressure reduces because we divided the entire project on various modules so that each and every collaborator can work on different/separate modules.

   - Working in group helped in easing the implementation of the project because we can take help from other members if we stuck on a particular thing for so long.

   Drawbacks of completing the lab in group:

   - Time Zone/Availability is different for each and every member of the group. Different members in the group have different schedules in their daily life. So, communication was little bit time consuming.

- Dividing the modules between group members took some amount of time because each and every member have different expertise/specialization (prefer to work on the particular module because of technologies).

5. Below are my three recommendations to ease the completion of the software project:

- Understanding design pattern for designing UML class diagram so that designing of project could be easy and the relationships between classes could be identified easily.

- Understanding JMenu and JPanel in depth is very important.

- Understanding classes and objects in depth so that Factory Design Pattern could be implemented easily on the different types of objects.

6. Tasks completed by each group member.

Boho Kim	code implementation, Junit tests, report part3
Saad Shahid	UML class diagram design & drawing, report part2
Yixing Cao	report structure, short video, report part1 & 4
Yun Lin	code implementation, Junit tests, report part3