**EECS 3311: Software Design F21**

**Software Project 2**

**Names:**

Hassan Hassan - Section A – 215216930

Andrew Ngov – Section B – 217227331

Khalid Gobin – Section B – 216853483

**Part 1: Introduction**

This software project is to develop a mini soccer game that can be played by the User (Striker). The Striker kicks the ball, and the Goalkeeper tries to grab it. If the Goalkeeper succeeds, then the Goalkeeper gets the point; the Striker gets a point otherwise.

The goal behind the development of this project is to figure out:

●      How the Striker, Goalkeeper, goal post, ball and string is drawn using the Graphics class.

●      How the location of Striker, Goalkeeper and ball is managed using x and y coordinates.

●      How the events like various keys pressed to play the game are managed.

●      How to kick the ball to the goal and back to the Striker.

●      How the various string messages regarding the score are sorted.

●      How the movement of the ball is drawn and controlled.

The following challenges are associated with the software project:

●      Figuring out what classes are needed to be defined to achieve the goal.

●      Figuring out how all the classes are integrated to meet the requirements.

●      Determining how the GUI is designed.

●      Determining how to draw various shapes and what should be their sizes and locations.

●      Selecting the best sorting technique for the string messages.

●      Choosing the event and keys to kick the ball.

●      Determining the movements of the ball.

●      How to paint the shapes.

●      How to repaint the shapes using a timer to refresh the game screen.

The following concepts of OOD are used:

●      Encapsulation

All the classes are designed using the concept of encapsulation, they each have their own data sets that is needed to design an object of that particular class. For example, each object of SoccerBall class will have its own set of positions, velocity and colour properties.

●      Abstraction

All the instance variables of the classes are made private, and we have provided setter and getter methods to access or modify the properties of an object indirectly.

●      Inheritance

Inheritance is a tool to reuse a software component as parent/child relation. We have made use of inheritance rigorously. For example, GamePlayer is inherited from the Striker and Goalkeeper classes since a Striker, and a Goalkeeper both are GamePlayers.

●      Polymorphism

                        “One name many forms”

In this project, the moveUp( ), moveDown( ), moveLeft( ), moveRight( ) , setStatistics( ) , getStatistics( ), setInitialPosition( ), isPlayerHasBall( ) , shootBall( ) etc. of GamePlayer class, are made abstract that is meant to be defined by all of its subclasses personally.

So whenever we create an object of any GamePlayer and call the ifPlayerHasBall(), it is automatically linked to the respective class, the object belongs to.

*Design pattern*

The MVC (model-view-controller) is an architectural pattern that allows separating a system into three main subsystems: the Model, the View, and the Controller.

MVC supports a separate implementation of the model from the view.

The model, the controller and the view respectively consists of groupings of elements (e.g., classes).

Each subsystem of the MVC can be further structured using design patterns.

*Singleton pattern*

SoccerBall class follows the singleton pattern since its static final field named soccerBall is an object of the SoccerBall class that is used by all the classes that require a SoccerBall.

*FactoryPattern*

The PlayerFactory class is solely responsible for creating the objects of Striker and Goalkeeper classes.

**Part 2: Design of the solution**

*Basic components of a UML class diagram*

●      Upper section: Contains the name of the class.

●      Middle section: Contains the properties or attributes of the class.

●      Bottom section: Includes class operations (methods).

Access attributes are specified as:

+    public access

-       private access

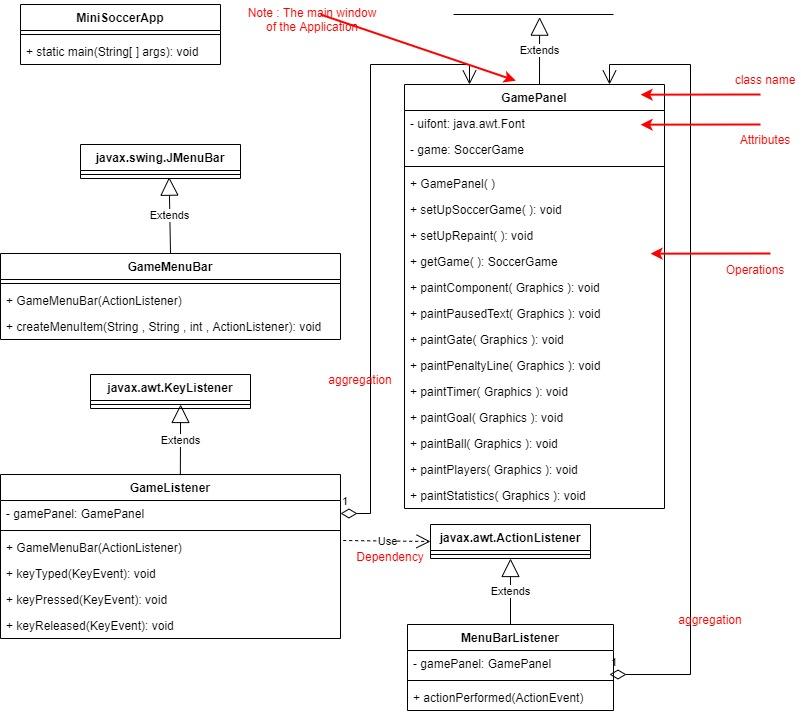
#    Protected access

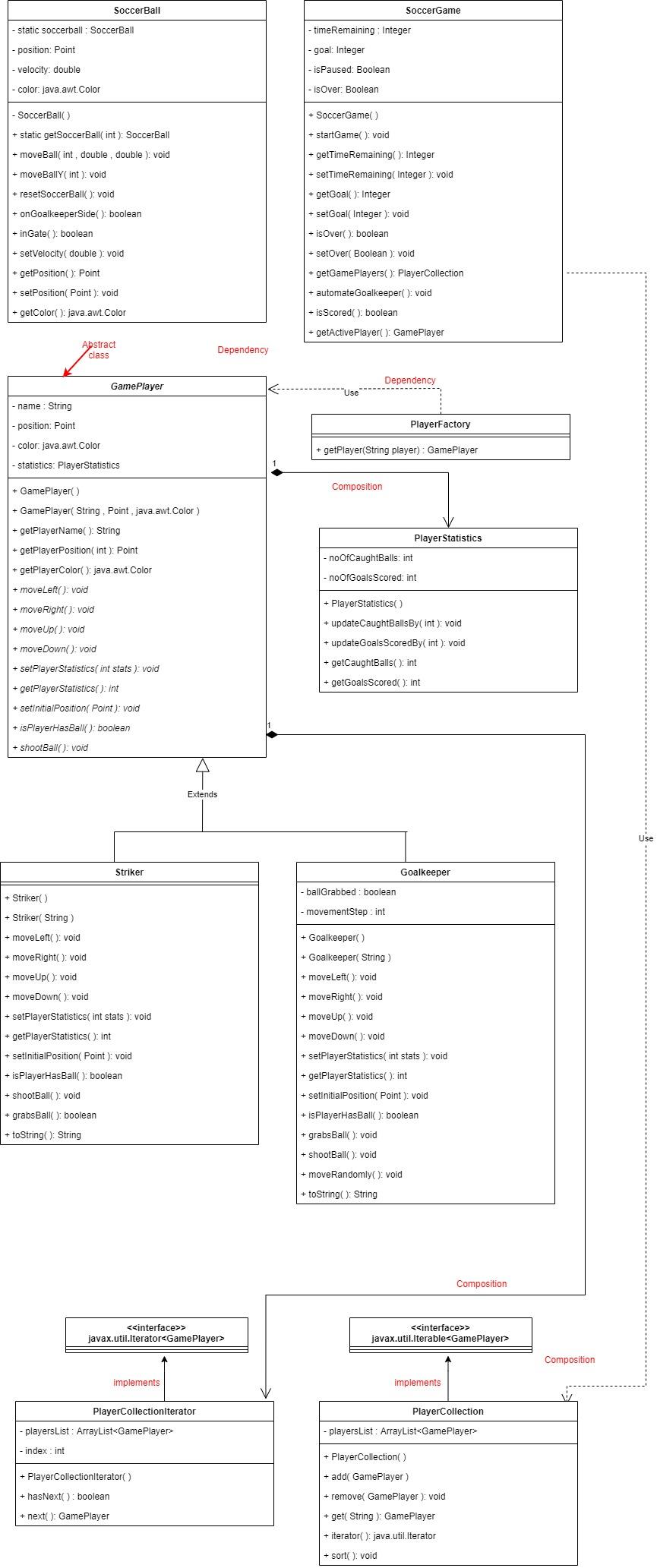
*Basic elements of a UML class diagram*

Class: a set of objects having identical responsibilities. A class embodies a concept which encapsulates state(attributes) and behaviour(operations).

Interface: a set of operations specifying the responsibility of a class.

UML class diagram





Design Patterns used in the project:

1. Singleton Pattern

SoccerBall class uses the singleton pattern so that only one instance of the SoccerBall can be created that can be used by any class in the project.

1. Inheritance Pattern:

Inheritance is a tool to reuse a software component as parent / child relation. We have made use of inheritance. The class GamePlayer is inherited to the Striker and Goalkeeper classes since a Striker and a Goalkeeper, both are GamePlayers.

*Object Oriented Design Principles used*

GamePlayer class

It is an abstract class that encapsulates the name, position, color and statistics of the player in the game. It has several abstract and nonabstract methods in it like getPlayerName() , getPlayerColor() , getPlayerosition() are the defined method whereas moveUp(), moveDown(), moveLeft() moveRight() , setPlayerStatistics() , getPlayerStatistics() , isPlayerHasBall() , shootBall() are abstract methods whose proper implementation is done by the respective sub classes.

Striker class

This class is subclass of GamePlayer class, so the class has rights to access all the properties of GamePlayer class. It has defined moveUp(), moveDown(), moveLeft() moveRight() , setPlayerStatistics() , getPlayerStatistics() , isPlayerHasBall() and shootBall() of GamePlayer class in it accordingly and grabsBall() additionally that checks and determines how the striker grabs the ball if thrown by the goal keeper.

Goalkeeper class

This class is subclass of GamePlayer class, so the class has rights to access all the properties of GamePlayer class. It has defined moveUp(), moveDown(), moveLeft() moveRight() , setPlayerStatistics() , getPlayerStatistics() , isPlayerHasBall() and shootBall() of GamePlayer class in it accordingly and grabsBall() additionally that defines how the goalkeeper grabs the ball if kicked by the striker.

**Part 3: Implementation of the solution**

*GamePlayer class*

 This is an abstract class that encapsulates the name, position , color and statistics of the player in the game. It has several abstract and non abstract methods in it like getPlayerName(), getPlayerColor(), getPlayerosition() are the defined method whereas moveUp(), moveDown(), moveLeft(),  moveRight(), setPlayerStatistics(), getPlayerStatistics() , isPlayerHasBall() , shootBall() are abstract methods whose proper implementation is done by the respective subclasses.

*Striker class*

This class is a subclass of the GamePlayer class, so the class has rights to access all the properties of the GamePlayer class. It has defined moveUp(), moveDown(), moveLeft() moveRight() , setPlayerStatistics() , getPlayerStatistics() , isPlayerHasBall() and shootBall() of GamePlayer class in it accordingly and grabsBall() additionally that checks and determines how the striker grabs the ball if thrown by the goalkeeper.

*Goalkeeper class*

This class is a subclass of the GamePlayer class, so the class has rights to access all the properties of the GamePlayer class. It has defined moveUp(), moveDown(), moveLeft() moveRight() , setPlayerStatistics() , getPlayerStatistics() , isPlayerHasBall() and shootBall() of GamePlayer class in it accordingly and grabsBall() additionally that defines how the goalkeeper grabs the ball if kicked by the striker.

*PlayerStatistics class*

This class has encapsulated noOfCaughtBalls and noOfgoalsScored that keep track of balls caught by the goalkeeper and goals scored by the striker, respectively. In addition, it has methods like updateCaughtBallsBy() that updates the balls caught by the goalkeeper, updateGoalsScoredBy() that updates the goals scored by the striker and some getter methods of the properties.

*PlayerFactory class*

This class provides a method named getPlayer() that takes a string and returns the Striker or Goalkeeper object accordingly.

*PlayerCollection class*

It implements the Iterable interface. This class has encapsulated a list of GamePlayer class objects that denotes a list of all the players and provides methods as add() that adds the given player to the list, remove() that removes the given player from the list, get() that returns the player with the given name from the list  sort() that sorts the players depending upon their statistics and an iterator() that returns the PlayerCollectionIterator class object to iterate the players in the list.

*PlayerCollectionIterator class*

It implements the Iterator interface. This class has a list of GamePlayer class objects that denotes the list of all the players from the PlayerCollection class. In addition, it implements the abstract methods of the Iterator interface like hasNext() that returns true if any player is remaining in the list while accessing and the next() that returns the next available player in the list.

Junit testing

Junit tests are written to test the code. Jacoco is implemented to test the code coverage during unit testing.

All the classes are compiled and executed on Eclipse IDE.

Eclipse IDE for Java Developers

Version: 2020-12 (4.18.0)

Build id: 20201210-1552

OS: macOS 11.1

Java version: 12.0.2

**Part 4: Conclusion**

The inheritance and polymorphism principles of OOD helped here a lot. The classes were created using the principle of inheritance so that Striker and Goalkeeper classes inherited the GamePlayer . GamePlayer class provided some abstract methods like moveLeft() , moveRight(), moveUp(), moveDown(), isPlayerHAsBall(), shootBall(), etc., that are defined by all of those subclasses according to that particular need and while creating an object of that particular GamePlayer. The respective method is chosen automatically due to the polymorphism.

Initially we did not put the grabsBall() in the Striker class so it was unable to grab the ball while the ball was within the 55 points radius of the striker . we found it wrong at the later stage. Then, we put the grabsBall() method that grabs the ball while it is in the 55 points radius of the striker and returns true if the ball was grabbed by the striker, and false otherwise.

We learned how to use the inheritance and polymorphism in a real project and how to develop a gui, manage the key events, manage the x-y coordinates of the screen and positions of various drawings, drawing various shapes, manage timed activities and sort various GamePlayers.

Teamwork has a lot of advantages like the expert of a field is assigned the task of his expertise, so that the group software project could be done more efficiently. A big problem is broken down into several small problems using the principles of OOP and then each team member is assigned a piece of the problem that they can do quite efficiently.

Abstraction, inheritance and polymorphism are the top recommendations to solve a bigger assignment in a much more efficient way.

There were 4 members in my group and each member did the task assigned to them successfully. A member was assigned to create the PlayerStatistics class that maintains the statistics of the goals and balls captured by the player. The GamePlayer class was designed by another member who put some common definition methods of Striker and Goalkeeper in the class as defined versions and some common behaviour methods as abstract to be defined by the respective subclass particularly. A member was assigned the job to design the Goalkeeper class that extends the GamePlayer class, and PlayerCollection class. Finally, the last member was assigned the job to design the Striker that also extends the GamePlayer class, and PlayerCollectionIterator class.