EECS 3311 Lab: Software Project 2

Kurt Magsipoc-Wilson 216047953

Abbyrahm Harrymanoharan 215099369

Oreoluwa Ogunlude 216594285

Corneille Bobda 217548595

section A TA: Kazi Mridul

Introduction

The purpose of this project is to make an application for a mini soccer game. The application should have menu options game and control. The game menu is to be used to start a new game and close the game, the control menu is for pausing and resuming the game. The game interface should include two players: a striker and a goalkeeper and a gate/net on a field. The interface also has the time remaining and goals scored displayed. The timer will start from 60 seconds and when it reaches 0 the game will end. While the game is being played the striker is to try to shoot the ball into the net for a goal and the goalkeeper will move randomly left or right in front of the net to prevent a goal. If the goalkeeper catches the ball or the striker misses the goalkeeper returns the ball to the striker's side of the field, both counts as if the goalkeeper caught the ball. If the striker gets the ball in the net the game will be paused, and the position of the striker and the goalkeeper is reset. When the game ends the players do not move and the stats of both players are displayed (balls caught and goals scored). The arrow keys are used to control the striker's movement and the space bar is used to shoot the ball.

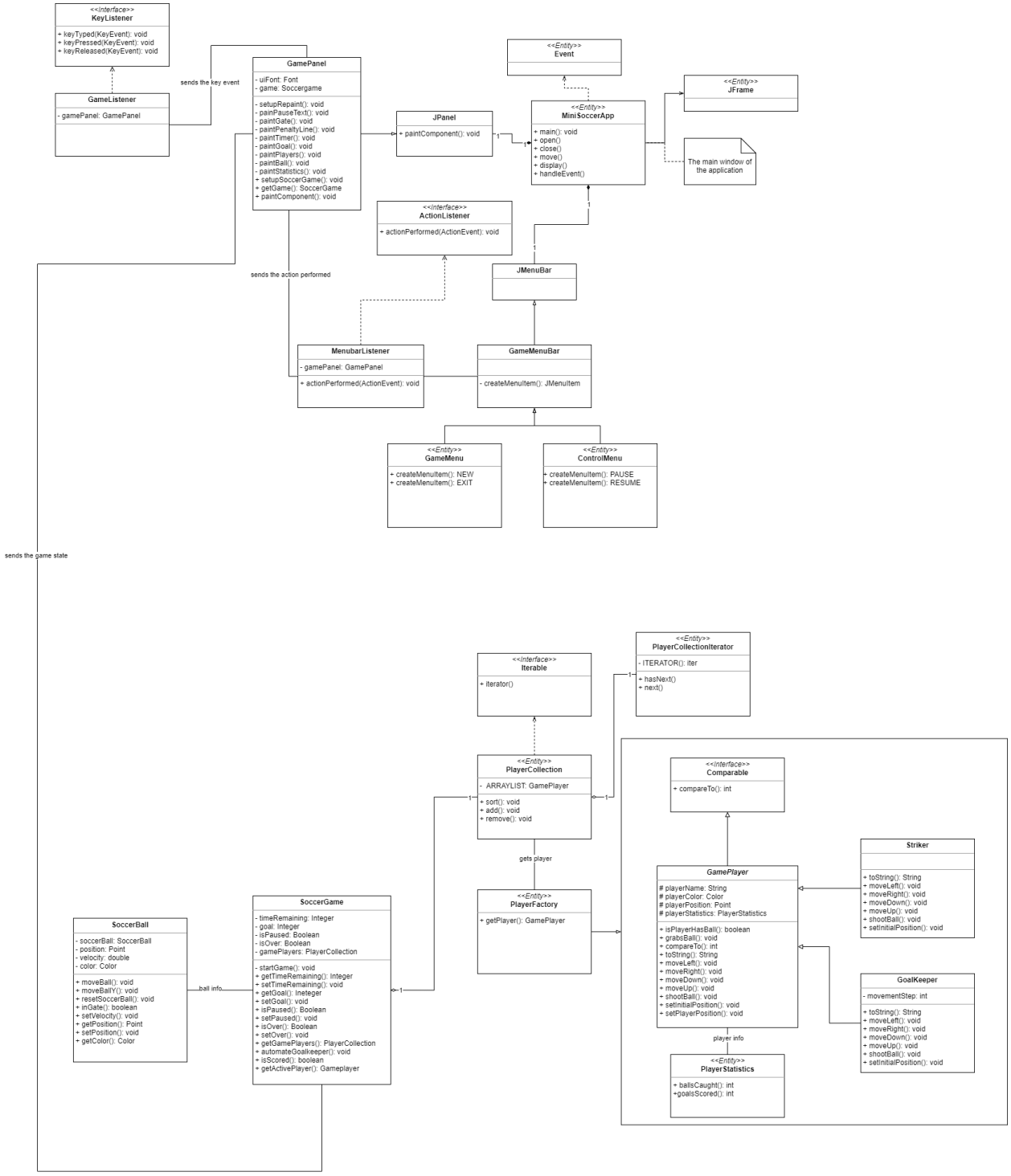
During this project we will face a few challenges with designing the application and implementing that design. One of which is being able to have multiple people develop the UML diagram concurrently.

OOD will be used in the creation of the application to help define the different objects involved in making the mini soccer game as well as their various states and behaviors. In the case of the GamePlayer object will have various states and behaviors like player color and set player position. The project will also require the use of the main OOD principles abstraction, encapsulation, polymorphism, and inheritance. Abstraction is used to make the interaction between the different classes smoother and easier. Encapsulation is used to keep everything together and easily accessible. Inheritance is useful when using generalization to create objects in the case of the game inheritance will be useful in the creation of the players. Polymorphism is useful when you use the same method for different objects, for example the movement of the striker and the goalkeeper. The project will require the use of design patterns one of which is the factory design pattern in the creation of the game players. We will also use the singleton design pattern for the soccer ball and soccer game classes.

This report consists of 4 parts namely the introduction, design of the solution, implementation and conclusion. This report will go on to discuss the details of the design of our project. It will explain the design patterns used and the OO design principles in our UML class diagram. We will also discuss our implementation of our design

Design of the solution

UML Class Diagram of the Mini Soccer App



Below are the corresponding classes of the UML class diagram, their descriptions and the relationships between them.

- GamePlayer: any player of the soccer game. Implements the Comparable interface.
- Goalkeeper : it comprises the goal keeper's attributes and operations. Extends the GamePlayer class.
- Striker: it represents the striker's attributes and operations. Extends the GamePlayer class
- PlayerStatistics: it indicates the statistics of a player. Is associated with the GamePlayer class.
- PlayerCollection: an iterable Collection of players that needs to have methods to initialize a collection of players, to add a player and to remove a player from the collection. This class implements the Iterable interface and has a one-to-one aggregation relationship with the SoccerGame class as well as the PlayerCollectionIterator class. It is also associated with the FactoryPatternDemo class.
- PlayerCollectionIterator: iterator that allows iterating over the PlayerCollection. Has a one-to-one aggregation relationship with the PlayerCollection class.
- GamePanel: extends the JPanel class and is associated with the GameListener class. This class has methods to help with the creation of the UI.
- KeyListener: provides important methods that are overridden in the GameListener class and acts as its parent class.
- GameListener: depends on the KeyListener interface and is associated with the GamePanel class. This class allows for the game to react to key entries.
- JPanel: has a one-to-one aggregation relationship with the MiniSoccerApp class and is a parent class to the GamePanel class. This class is used to support the GamePanel class through inheritance.
- ActionListener: acts as a parent to the MenubarListener class. Used to provide support to the MenubarListener class.
- GameMenuBar: extends the JMenuBar class, is a parent to the GameMenu and ControlMenu entities and is associated with the MenubarListener class.
- JMenuBar: acts as a parent class to the GameMenuBar class and has a one-to-one relationship with the MiniSoccerApp class.
- GameMenu: is a child class to the GameMenuBar class
- ControlMenu: is a child class to the GameMenuBar class
- MenubarListener: implements the ActionListener interface and is a child of it. It is also associated with the GameMenuBar class.
- SoccerBall: creates one soccer ball that is reused throughout the whole game. It is associated with the SoccerGame class
- SoccerGame: Has an aggregation with the PlayerCollection class and is associated with the SoccerBall class. Provides methods used to manage the game.

- Comparable: provides the compareTo() method to the GamePlayer class by acting as its parent.
- PlayerFactory: it gets an instance of a gameplayer (goalkeeper or striker) whenever a class needs it. It is associated with the GamePlayer class.

When designing the UML class diagram, the Factory design pattern and the Singleton design pattern are very much present.

- Factory Pattern: let's say the client wants to create a GamePlayer instance at random, whether a GoalKeeper instance or a Striker instance. Rather than having to modify the code in the GamePlayer class, the client can use the PlayerFactory class to achieve such a feat.

- Singleton Pattern: let's say while playing the game, if the striker scores the position of the striker and the ball must reset. While resetting, the client wants to reuse the same soccer ball rather than a new one. For this reason, the SoccerBall class is a singleton class.

The Object Oriented design principles used in the UML class diagram are the following:

- Inheritance: we observe inheritance in the program through the GamePlayer class which acts as a parent class to both the Goalkeeper and Striker classes. Inheritance is also observed through the GamPanel class which extends the JPanel class and the GameMenuBar class which extends the JMenuBar class.

- Abstraction: we make use of abstraction in the program through the creation of the abstract class GamePlayer. When running the program, the information of any game player instance whether the goalkeeper or the striker is irrelevant to the client. The client also does not need to know how the movement of the different gameplayers are made possible or how their statistics are calculated.

- Encapsulation: we make use of encapsulation within the SoccerBall class by making the position and the soccerball variables private. Getters and setters methods are created to access these variables.

- Polymorphism: polymorphism is alive in the Goalkeeper and Striker classes as the methods moveLeft(), moverRight(), moveUp(), moveDown(), shootBall(), setInitialPosition() and toString() are all overridden and are a different version of the ones in the parent class GamePlayer. In the GamePanel class the paintComponent() method is overridden.

Implementation

The MVC design pattern was used to implement the game. This pattern is broken into three different objects that have specific uses. The model contains the application data, however, does not contain any logic to allow this data to be presented to the user. The view is what allows the user to be presented with the data but does not know how to use the data or what it means. The controller is what connects the model and the view as it listens to inputs that are triggered by the view and executes the appropriate action.

Previous classes were already implemented in the controller and view packages. Here is a breakdown of the implementation process of the missing classes required in the model package and their interaction within the system.

PlayerFactory

This class was implemented using the factory pattern and creates GamePlayer objects based on the given input. It has an association with the GamePlayer class and its subclasses Striker and GoalKeeper. When creating the GamePlayer object, it assigns a name and a random color when the object is created.

PlayerCollection

PlayerCollection is a class that creates an iterable collection of GamePlayers and implements the Iterable interface. This class has one field called gamePlayers that contains the collection of GamePlayers created. Also included in this class are methods that allow to sort, add, remove, and get GamePlayers from the created collection. It also has a method that allows the list of GamePlayers to be iterated over.

PlayerStatistics

This class is used to store and retrieve the statistics of the GamePlayers in the game. It has one field that holds the individual score of the GamePlayer. This class has a constructor to set the score field to 0 when called. There are three methods present: the first two are getter and setter and the last returns the score field as a String.

The tools that were used in this software project are the following:

Git - This tool was used for our source code management.

GitHub - A version control system that uses Git which allows for team members to collaborate together and make changes to the code.

Eclipse - The IDE that was used for development.

Terminal - A command line system that was used to input commands to run the software and utilize Git.

JDK - The Java Development Kit was used to develop the application using its API.

Diagrams.net - The website used in creating the UML Class diagram.

Conclusion

Some things that went well with the software project is that we were able to apply a lot of the skills and knowledge learned during the course to the project. We are able to use our knowledge of design patterns, OOD, and OOD principles to come up with a design suited for the project.

Some issues we had with the project is that we had trouble working on the same parts of the project at the same time. There were some parts of the implementation that we had trouble with as well but we were able to come together and solve them.

Through the software project we have gained insight on design patterns, OOD, and OOD principles in a more practical sense. We gained this insight by having to both build our design with implementation in mind and through the implementation gained a better understanding of the design itself.

Some advantages of completing the lab as a group is that it allowed us to help each other when we got stuck and allowed us to focus on our individual strengths. Some drawback to working in a group was that we had trouble working on a single file together. Another issue is that sometimes we needed information that other members of the group had or were waiting on the completion of certain aspects before we could continue.

To ease the completion of the software project we could:

- Provide simple testing to get started
- Create a schedule with deadlines that take time to complete and test into to account
- Have regular meeting to discuss progress and difficulties and how to move forward

TASK	ASSIGNED	WORKED ON
Design	Corneille, Oreoluwa	Corneille, Oreoluwa
Implementation	Abbyrahm, Kurt	Abbyrahm, Kurt
Report	All	All

All members of the group were active in completing the project and helped out when questions were asked.