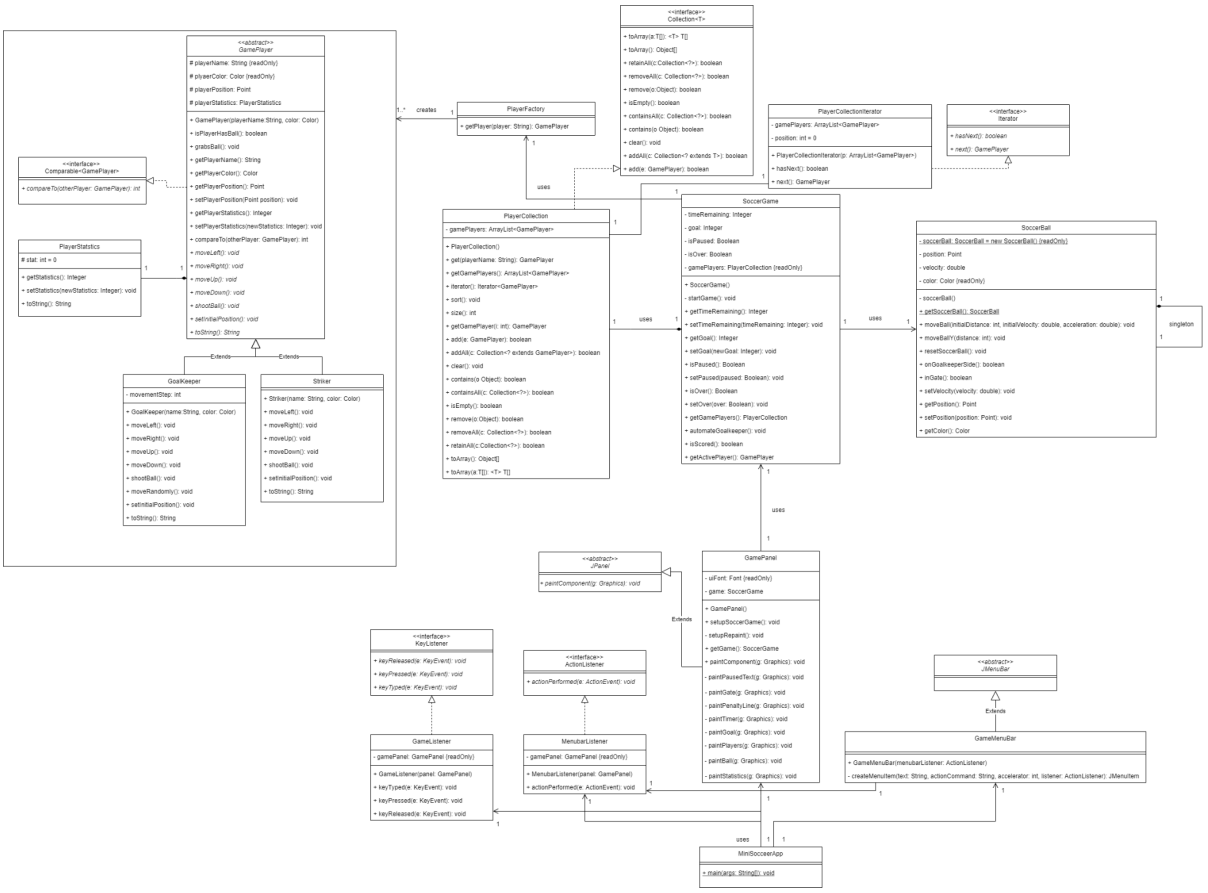
Mini Soccer Game Project (Second Software Project/Lab 5) EECS3311
Usama Taqikhan, 215414758, Section A TAs: Kazi Mridul (kmridul@yorku.ca)
Muhammad Abbasi, 215614555, Section B Alireza Naeiji (naeiji@yorku.ca)
Muhammad Zaighum, 214947022, Section A Due: November 9, 2021, 11:59PM EST
Dongwon Lee, 215805260, Section B

Introduction

For this software project, we were assigned the task of making a mini-soccer game. The game consists of a striker that strikes the ball towards the gate and a goalkeeper that tries to stop the ball from going in the gate. The striker has 60 seconds to score as many goals as possible and the goalkeeper tries to make as many saves as needed. The striker is controlled by the user and the goalkeeper moves in front of the gate(net) randomly. The number of saves the goalkeeper makes and the goals the striker scores are recorded and displayed when the timer ends. There is a menu allowing the user to pause or resume the game, start a new game or exit the game. The biggest challenge we faced when implementing our design for this project was implementing the iterator for player collection as we weren't sure what it was used for and had trouble understanding exactly what was the purpose of it for this project. We also had difficulty understanding what was required for the iterator and hence had trouble implementing it. Two of the patterns that will be used in this project are factory pattern and singleton pattern. Both these patterns are considered to be creational design patterns. They are focused on the instantiation of objects and are essential in allowing the decoupling of a system. Singleton also has the ability to be used within the implementations of patterns such as the abstract factory, builder and prototype. The report will be structured based on the requirements given. The written aspect will include the introduction which will be followed by the design section. This part of the report will include the UML class diagram of our system and also an explanation of how the design patterns are used. Furthermore, we will analyze the code for this project and describe/explain its implementation, documentation, JUnit testing and the tools used. The written report will end with a conclusion that will talk

about various things such as the positives/negatives of this project. To go along with the written report there will also be a video demonstration of how to launch and run this application and the entire code which will be submitted through GitHub.

Design:



UML class diagram file (UML.drawio) and image (UML.drawio.png) are uploaded on the Github repository. Elements used in the UML are the following: The GamePlayer class is in an inheritance relationship with Striker and Goalkeeper. GamePlayer and PlayerStatistics are in a composition relationship so that GamePlayer has its own PlayerStatistics. PlayerFactory and GamePlayer are in association relationship so that the GamePlayer object can be created by PlayerFactory, which is a Factory class using the Factory pattern. PlayerCollection and PlayerCollectionIterator are in association relationship and the latter provides an iterator to the former. SoccerGame uses PlayerFactory to create GamePlayer objects and store them in the PlayerCollection object, that are in a composition relationship. It also owns a SoccerBall

object, which is an aggregation relationship. GamePanel and SoccerGame are in association relationship. MenubarListener and GameMenuBar are in association relationship. MiniSoccerApp is the client class and has an association relationship with GamePanel, GameMenuBar, GameListener, and MenubarListener.

Two design patterns are used in the project. Factory pattern and Singleton pattern. Firstly, the Factory design pattern. Two game players, the Striker and the Goalkeeper, are used in the program. So a Factory design pattern was used to create those two game players. PlayerFactory is the factory class and it has the method getPlayer() that creates a GamePlayer object. Then the SoccerGame uses the PlayerFactory class to create the two game players needed. Secondly, the Singleton pattern. This program is a soccer game. So, there should be only one soccer ball. In order to maintain only one soccer ball in the game, the SoccerBall class was made into a singleton class and the soccerBall field was made into a static final variable and the getSoccerBall() method was made into a static method. In this way, the SoccerBall class became Singleton and only one SoccerBall instance was maintained.

Four OO design principles are used in the project. Abstraction, polymorphism, encapsulation, and inheritance. Firstly, the abstraction. Striker and Goalkeeper share some common fields and methods, so they are abstracted and an abstract class GamePlayer is created to put those common features. Striker and Goalkeeper inherits the GamePlayer class and overrides or adds additional unique fields and methods. By this way, abstraction was used. Secondly, the encapsulation. Most of the classes have their fields as private fields and the only way to access those fields is by using getter and setter methods. By this way, encapsulation was used. Thirdly, the inheritance. Inheritance is used to avoid multiple use of duplicate fields and methods and this is used in the project. Striker and Goalkeeper inherits the common features from the GamePlayer class. Also, GamePanel inherits from JPanel and GameMenuBar inherits from JMenuBar. By this way, inheritance was used. Fourthly, polymorphism. Striker

and Goalkeeper inherits several movement methods from GamePlayer and implement it according to their own unique movement. And the GamePanel class, which extends the JPanel class, overrides a JPanel method, paintComponent method in order to paint proper elements of the soccer game. By this way, polymorphism was used.

Implementation:

The project was implemented and compiled with 5 different packages: main, view, controller, model, and model.players. The main package has only one class, MiniSoccerApp, which is the client and demo class that shows the whole GUI of the soccer game by using other GUI component classes and connects the listeners so that the user can interact with the soccer game GUI. The view package has those GUI component classes, GamePanel and GameMenuBar. GamePanel extends JPanel and paints the GUI of the main elements of a soccer game such as the striker, goalkeeper, gate, timer, and so on. GameMenuBar extends JMenuBar and paints the GUI of the top menu bar elements such as ‘New Game’, ‘Exit’, ‘Pause’, and ‘Resume’. The controller package has listener classes, GameListener and MenubarListener, that allows the user to interact with the program. GameListener extends KeyListener and defines methods that make the game player model of the soccer game move according to the key pressed by the user. It is connected to the GamePanel. MenubarListener extends ActionListener and defines methods that the option is the top menu bar controls the game status when clicked. It is connected to the GameMenuBar. The model package has the SoccerBall and SoccerGame classes. The SoccerBall class defines the soccer ball in the game and has fields and methods for its movement, position check, and so on. It is a Singleton class that allows only one SoccerBall object. The SoccerGame class defines the soccer game instance and has fields and methods for starting, counting time, and checking the game status. The model.player package has the main element object classes related to the game player,

that are: GamePlayer, Goalkeeper, Striker, PlayerStatistics, PlayerCollection, PlayerCollectionIterator, and PlayerFactory. GamePlayer is the parent class of Goalkeeper and Striker that abstracts the common features of the two. Goalkeeper is the concrete class that inherits GamePlayer and defines the unique attribute and behaviours of the goalkeeper. Striker is the concrete class that inherits GamePlayer and defines the unique behaviours of the striker. PlayerStatistics defines the score of the Goalkeeper and Striker. PlayerCollection is the class that implements Collection to allow handling multiple GamePlayer objects. The objects can be added, removed, sorted, and so on in the collection class. PlayerCollectionIterator is the class that implements Iterator and it provides an iterator for PlayerCollection. PlayerFactory is a factory pattern that allows creating a GamePlayer object. Javadoc of the source code is uploaded on the Github lab5/MiniSoccerGameProject/doc. JUnit tests were also uploaded to Github. We put the test class inside the model package and called it modeltest.java. The classes we were testing were specifically in the model package as that was told to us to test during a lecture. We started by testing various aspects of the Striker class, which in turn also helped test aspects of the GamePlayer class as the Striker class is a child of the GamePlayer class. The aspects that were tested for Striker are the movement methods; testing cases where movement is possible and impossible. For example the moveLeft() method can only work if a Striker's current X coordinate minus five is greater than zero. If the Striker was at point (4, 250), moveLeft() would not run and the Striker would not be moved left. We tested cases where the moveLeft() method would run and would not run. Similarly, this testing process was applied to the other movement methods in both Striker and Goalkeeper. We also tested the getter methods such as getPlayerName() or getPlayerStatistics(). For PlayerCollection aspects such as adding objects, removing objects, getting specific objects were tested. Iterating through the collection was also tested. For PlayerFactory, we tested the creation of Striker and Goalkeeper and also tested whether trying

to create an object not specified by the factory would result in error or be handled accordingly. Our coverage for the model.players package ultimately was able to reach 100% while the coverage for the model package, which has the SoccerGame and SoccerBall classes, was 93.7% so overall, taking the coverage of these two packages and getting the average, we were able to achieve a code coverage of 96.85%.

Tools used in this project are Eclipse 2020-06 (4.16.0) version and JDK version 1.8. We also used JUnit Version 5 in order to make the JUnit tests for the model classes.

The video of this project application is uploaded on Github and was recorded using Zoom screen recording.

Conclusion:

The things that went well in this project was the collaboration and the type of project that was given. Having good and hard working group members was essential in getting this project complete on time and efficiently. Furthermore, having a project related to a sport that most people either play or watch was a great motivation to our group as it was a topic of interest and that made it fun.

Not many things went wrong in this project, but to pinpoint one thing, it would have to be issues with the iterator. This part didn't go so well because no one in the group really understood the reason for its need and how it could benefit us in completing this project. Although it was a challenge, it helped us think outside the box which is not the worst thing.

The things we have learned from this software project are how we can use the design principles and patterns that we have learned throughout the course to implement this project, but also in other courses and in future jobs where projects similar to this will be made. It is one of the more realistic projects many of us have done in University and with the teachings of the design related stuff it turned out to be of great benefit.

There are many advantages of completing this project in groups. Firstly, it divides up the workload, something that is crucial in upper level courses. Furthermore it gets the best out of everyone since we all took care of parts we were best at. Lastly and most importantly it is very beneficial and efficient when there are doubts on how to do something for one of the group members, since there are three other people who have their own ideas and explanations of things that can help learn the concepts much quicker and easier.

Although for our group there were not many drawbacks, since we were able to get the project done on time and to the best of our capabilities. One thing that may have been a drawback was having to do the entirety of this assignment online and not meeting up. It slightly slowed down some aspects of the project that would have been much quicker had this project been done in school together. It also deprived some of our members to get their best ideas on display because they are far greater in explaining things in real life as compared to online.

Three things that would ease the completion of this software would be firstly, as mentioned above, have some sort of meetup in person as it is an asset to some group members ability to be at their best. Furthermore, a little more detail on the iterator for player collection would have been a benefit to our group as it is something we struggled on. Lastly, some more clarity on some of the project instructions might have made it slightly easier to complete. Specifically the requirements of "Part 3: Implementation of the Solution", was a little difficult to grasp at first when this report was being made.

Roles:

Usama Taqikhan	-Coding the Project -Testing the Project -Video explaining the launch of Project - Typing up the report, Editing the report
Muhammad Abbasi	-Typing up the report, Editing the report, organizing the report. - Testing the project
Muhammad Zaighum	-Typing up the report, Editing the report, organizing the report.
Dongwon Lee	-UML class diagram -Creating Javadoc -Typing up the report, Editing the report