Chun-Kit Chung, Isaiah Linares, Leong Li
217125329, 217287509
EECS 3311, Section B, Second Software Project, TA: Naeiji Alireza

Mini Soccer Game Application Report

Isaiah Linares, Leong Li, Chun-Kit Chung, Zhexu Liao

11/03/2021

Table of Contents

Introduction
Design
Implementation
Conclusion

Introduction

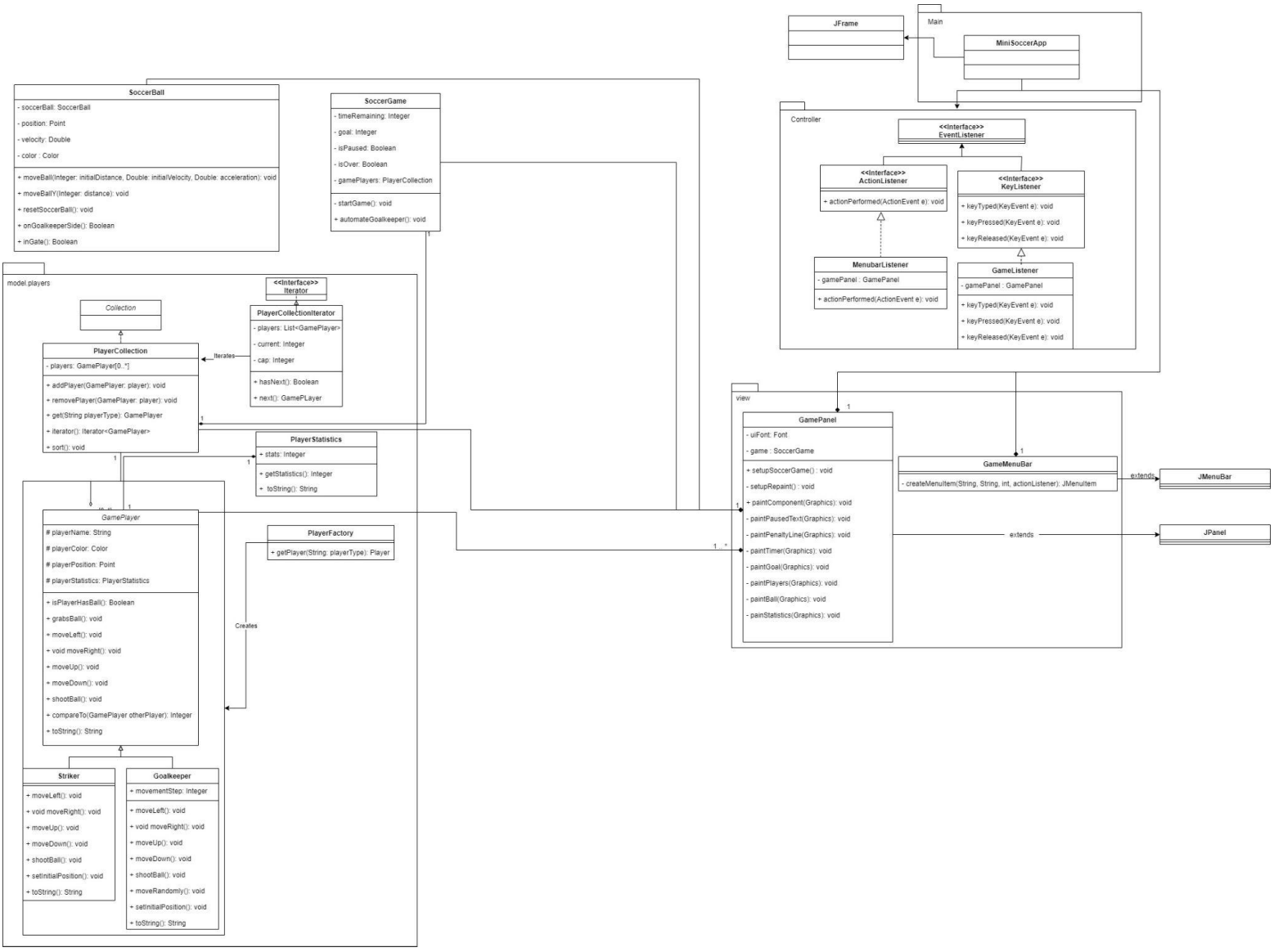
The goal of this software project is creating a Soccer Game Application using object oriented principles and design patterns. This application will display an interface with two menus: Game and Control; allowing for the user to pause or resume the game and to start a new game or exit the game.

The challenges of this project include designing and implementing: behaviour of goalkeeper and striker, displaying time remaining and enforcement of time rules, score/catch rules, and pausing. There are many restrictions and invariants that apply to the behaviour of the goalkeeper and striker. Some integral restrictions are: the goalkeeper is in front of the Gate, the striker tries several times to score a goal and each time the goalkeeper attempts to save it, and either the goal keeper or the striker can cross the penalty line. We must also display the time and score while enforcing the rules surrounding them. This will be difficult as there are many rules to this game including: the time starts at sixty second and the game must end when it is down to zero, you can always pause/resume/start a game, and the initial score is zero. In order to overcome these challenges we must use contracts, testing, and object oriented design and analysis to develop an implementation that can satisfy out requirements.

To create this application, we will be using Object Oriented Analysis to find distinct object concepts applicable to our application as well as object oriented design to specify software objects needed to satisfy our requirements. We will use encapsulation, abstraction, inheritance,

and polymorphism in our planning and implementation of this game. The design patterns used in the design of this application include Factory design pattern and the Iterator design pattern.

This report is structured into four sections, the introduction section where we explain the goal of the project and how we plan to carry to accomplish this goal, the design section where we show the Object Oriented Analysis and Design as well as all the UML Diagrams and Models used to provide structure for this application allowing efficient implementation , an Implementation section going over how we decided to implement this application, and finally a conclusion where we will reflect on accomplishments and shortcomings of this project.



Design

Factory Pattern

We used the factory pattern to allow the client (in this case the SoccerGame class) to get an instance of a class without revealing the instantiation logic. In this case we used this pattern for our PlayerFactory class which returns an instance of the GamePlayer class to the SoccerGame class. This was useful in this situation as we could return either a Striker or a Goalkeeper and could hide the instantiation logic from the client using polymorphism.

Iterator Pattern

We also used the Iterator pattern, which allows the client to access the elements of an aggregate object sequentially without exposing internal representation. In this case the client was the

Some of the object oriented design principles we used in the class diagram were abstraction, encapsulation, polymorphism and inheritance.

For abstraction, we have created the abstract GamePlayer class. To access the abstract class, we created the Goalkeeper and Striker class to inherit the abstract class. Abstraction is also used when we use interfaces. The interfaces we used in this project were the ActionListener, KeyListener and EventListener interfaces. From there, the GameListener and MenubarListener implemented the KeyListener and ActionListener interfaces respectively. For the GameListener class, the keyPressed method was used to move the player and shoot the ball. In the MenubarListener class, the actionPerformed method was used to start a new game, exit, pause and resume the game.

As for encapsulation, in order to make sure that "sensitive" data is hidden from users, we declared all class variables as private as well as providing public get and set methods to access and update the values of the private variables.

In regards to polymorphism, it was used when creating the Striker and Goalkeeper class. In this case, these are subclasses of the GamePlayer class. They are different forms of a player and that is why the Striker and Goalkeeper classes are extending the GamePlayer class. Polymorphism allows us to use the inherited attributes and methods using inheritance from the GamePlayer class and perform a single action in many different ways. In this case it was the methods for the movement of the player and goalkeeper as well as shooting the ball.

Implementation

MainSoccerApp is the main class that allows launching the application. It creates and composes instances of classes from the controller and view packages. This class initialize the main frame, menu and panels of the application's interface.

GameListener implements the KeyListener interface. This class composes of one private final variable of type GamePanel. This variable is a reference to the GamePanel object passed through the constructor's parameter upon initialization. This class has three methods, which all override the KeyListener's methods. The keyPressed method identifies which key is pressed, and assigns actions to the soccerGame's active player accordingly.

MenuBarListener implements the ActionListener interface. This class composes of one private final instance of the GamePanel class, and has a method which takes an ActionEvent instance and assigns a set of actions depending on its action command. This can include new, exit, pause, and resume, which sets a new game, terminates the current game, pauses and resumes the game respectively. An exception handling was also included to handle runtime exception when the method receives an invalid action command.

GamePanel is a panel class with inherited properties from the JPanel class. Its main functionality involves setting up the panel in the interface with paint methods and setting up a soccer game by creating a new SoccerGame instance.

GameMenuBar instantiates the menu bar of the interface, inheriting properties from the JMenuBar. It constructs a bar at the top of the interface with the help of a createMenuItem method, to display the game menu and the control menu.

SoccerBall class maintains a single instance of the SoccerBall class. It contains methods to configure the position of the ball, the speed and distance, and check which side of the goalkeeper the ball is at. It's attributes include a position of type Point, velocity of type double, and color of type Color.

SoccerGame client class initializes the soccer game, setting the initial values for time remaining, goal, state and calls the PlayerFactory to create a striker and a goal keeper and accumulates it as gamePlayers. The class starts up the soccer game, initializing the timer and runs the game, checking its state (paused or not) with every second of the timer. It also contains mutator methods for timer, number of goals, state of the game, and game players. In addition, it automates the goalkeeper, either by catching and throwing the ball or by moving randomly.

PlayerCollection class implements the Iterable interface and iterates through instances of GamePlayer. This class creates and maintains a list of players, and utilizes a PlayerCollectionIterator instance to iterate through the list of players. It also contains mutator methods for the list of players, a method to sort the list and an iterator method which overrides the Iterable interface's abstract method and returns an instance of PlayerCollectionIterator.

PlayerCollectionIterator class implements the Iterator interface and iterates through instances of GamePlayer. It assists PlayerColection class in iterating through list of game players. It contains methods get the next GamePlayer in the list.

PlayerStatistics class maintains a player's statistics as its attribute, and accessed or changed via mutator methods. It also includes a toString() method to override Java's toString() method.

PlayerFactory is a factory class called by the client SoccerGame class, to instantiate game players, either an instance of a goalkeeper or an instance of a striker.

GamePlayer abstract class implements the Comparable interface, to compare instances of game player. This abstract class is an abstraction of a player, declaring common attributes such as name, color, position and statistics. Its abstract methods include directional (moving a player) and toString. Other methods include mutator methods for its attributes, a compare method to override the method in Comparable interface, and two methods for player interactions with the ball (grabbing and checking if player has ball).

Striker class inherits properties from the GamePlayer abstract class. It represents a striker, with methods on a striker's initial positions, movement (up, down, left, right) utilizing GamePlayer's set player position method, and finally an overridden toString method to display statistics.

Goalkeeper class inherits properties of the GamePlayer abstract class. Unlike the Striker class, it maintains an movement step attribute, which is the random value the GoalKeeper moves each second, either left or right. The other methods include setting initial position, an overridden toString, and overridden abstract methods of the GamePlayer abstract class (directional and shoot ball).

Tool used in during the implementation include Eclipse IDE. The java library used for unit testing is JUnit.

Conclusion

One aspect of the project that went well during the software project was making the PlayerStatistics and PlayerCollection class. It was very simple because we had already planned out the object concepts in the UML diagram before we started to implement the actual code. Another thing that went well was the creation of the UML diagram itself. Being able to work in a

group meant that it would be a lot easier to complete tasks as every aspect of the project that needed to be done in a timely manner would be delegated to a group member.

One aspect of the project that went wrong was the overall implementation of the PlayerCollectionIterator class. One difficult aspect was using polymorphism to make an iterator that iterates a PlayerCollection. This was difficult because the PlayerCollection is essentially a collection holding different types of objects all under the parent class GamePlayer. I was eventually able to overcome this by implementing the Iterator to work for any collection of GamePlayers.

Some things we have learned while completing this software project were how to practice object-oriented concepts. For example, the system should be divided into several categories, who inherited from whom, etc. And also practice the use of multiple design patterns, such as MVC pattern, iterator pattern, factory pattern, singleton pattern, etc.. We also learned how to write a simple java program and basic debugging and running methods. There are also some skills about java GUI programming that were learned throughout the process.

Some advantages of completing the lab in a group is that we can discuss/share our different ideas to complete certain tasks. This allowed us to quickly solve problems easily in contrast to working alone. On the other hand, some drawbacks of working together in a group is that some of the group members might not be able to meet on a specific day to work on the project or the communication was cut between group members. This leads to the problem that the other group members would have to rely on the others to make sure they finish their work on time. This would create a lot of stress on the other group members.

Our first recommendation to ease the completion of the software project would be to schedule specific days and times for the group members to meet and work on it together. Another recommendation would be to plan accordingly to make sure certain aspects of the software project are completed on time. This ensures that the group will not fall behind on work. My third recommendation would be frequent check-ins with each group member. This is also in combination with the previous recommendation. This guarantees that there are no issues with the actual work itself and if there are issues, they can be brought up in the check-ins so that the group can try and solve the issue.

Tasks	Portion of Work	Collaborative
Goalkeeper Striker GamePlayer PlayerCollection PlayerCollectionIterator	Chun-Kit Chung Leong Li	Yes

Test Cases		
SoccerGame SoccerBall PlayerFactory PlayerStatistics Test Cases	Zhexu Liao Isaiah Linares	Yes
Writing report	Chun-Kit Chung Leong Li Zhexu Liao Isaiah Linares	Yes