EECS 3311 Section B Lab 5

Mini-Soccer Game

Adrian Koduah - 216793887

Arian Mohamad Hosaini - 214969638

Wenjing Qu - 214568612

Roberto Shino - 213003561

Part I: Introduction

The software project is a group activity focused on designing and implementing a mini-soccer game in Java. Our goals as a group are to: create a design for the mini-soccer game, implement the mini-soccer game, write JUnit tests for the model and create a short video depicting how to run and play the mini-soccer game.

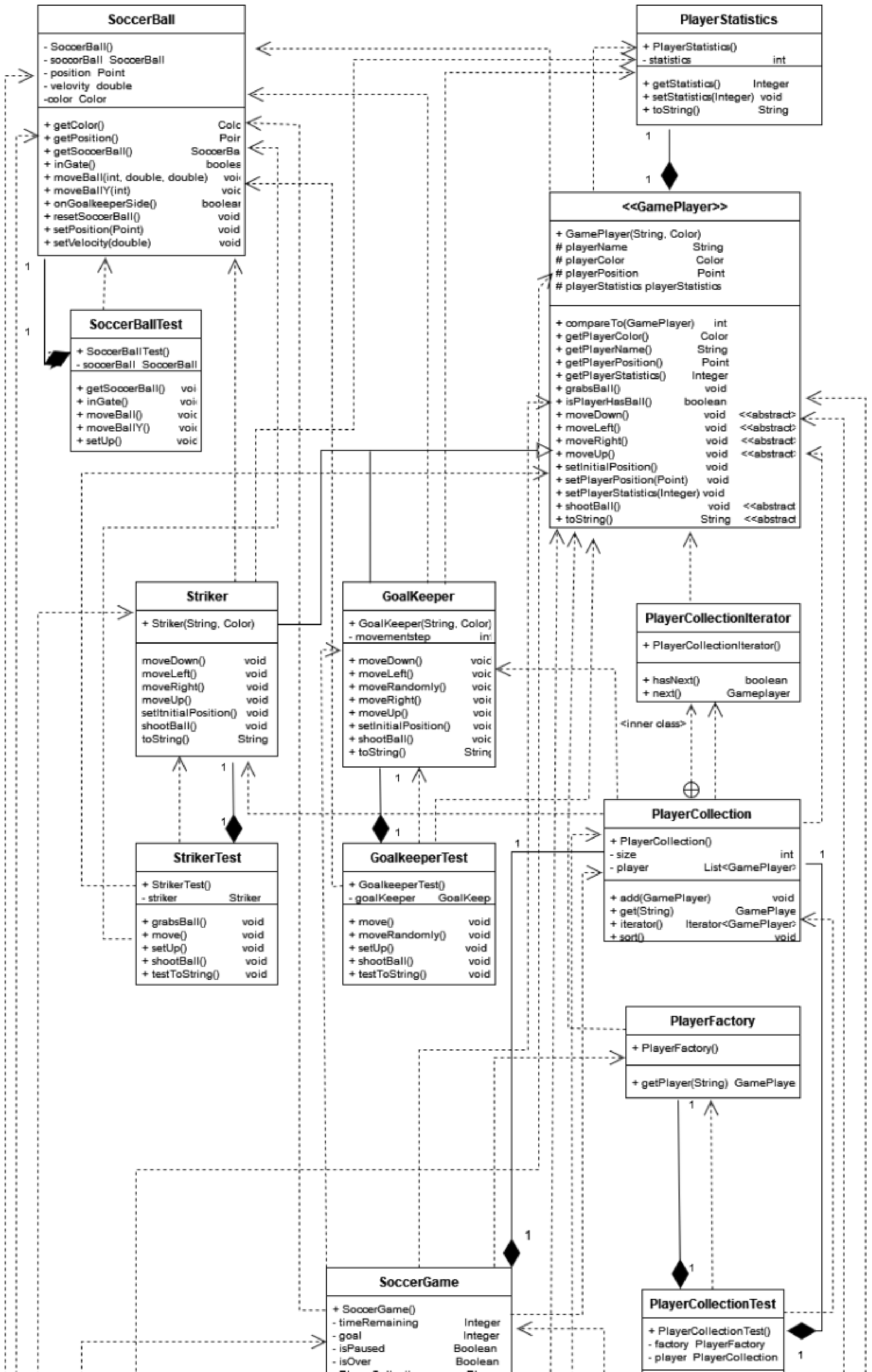
The main challenges associated with the software project would be creating a design which utilizes multiple OOD patterns and principles, implementing the game while ensuring that our code fully executes without exceptions, and writing JUnit tests for our model with 100% coverage.

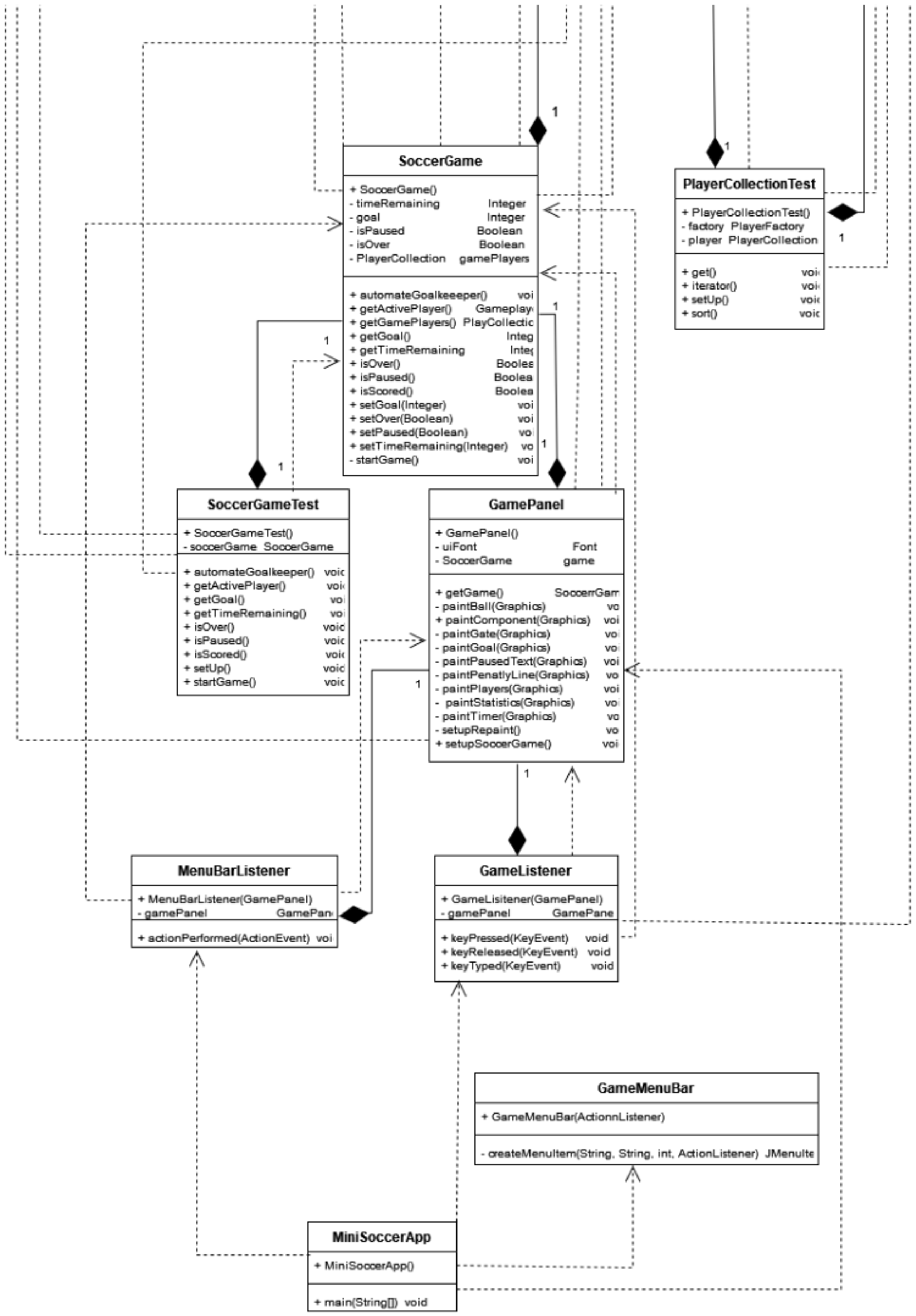
Our mini-soccer game will follow the Model-View-Controller (MVC) design structure. Design patterns and principles which we will use for this project are: factory pattern, singleton pattern, aggregation, inheritance, abstraction and polymorphism.

The remainder of our report will be broken up into three other parts. In part two we will include the UML class diagram depicting our system. We will also break down how we used OOD patterns and OOD principles in our design in terms of which classes used them as well as how they used them. In part three we will describe how we have implemented and compiled the classes of our system. We will also include the documentation of all the classes and specify which tools and libraries we used during the implementation. In part four we will wrap up by summarizing our experience with the project, and how we feel we did on the project.

Part II: Design of the Solution

(our diagram was too large and had to be fit inside two pages, the pdf format of our diagram can also be viewed from our repository)


SoccerBall
- SoccerBall()
- soccorBall SoccerBall
- position Point
- velovity double
-color Color
+ getColor() Colc
+ getPosition() Poir
+ getSoccerBall() SoccerBa
+ inGate() boolea
+ moveBall(int, double, double) voic
+ moveBallY(int) voic
+ onGoalkeeperSide() boolear
+ resetSoccerBall() void
+ setPosition(Point) void
+ setVelocity(double) void
PlayerStatistics
+ PlayerStatistics()
- statistics int
+ getStatistics() Integer
+ setStatistics(Integer) void
+ toString() String
<<GamePlayer>>
+ GamePlayer(String, Color)
# playerName String
# playerColor Color
# playerPosition Point
# playerStatistics playerStatistics
+ compareTo(GamePlayer) int
+ getPlayerColor() Color
+ getPlayerName() String
+ getPlayerPosition() Point
+ getPlayerStatistics() Integer
+ grabsBall() void
+ isPlayerHasBall() boolean
+ moveDown() void <<abstract>
+ moveLeft() void <<abstract>
+ moveRight() void <<abstract>
+ moveUp() void <<abstract>
+ setInitialPosition() void
+ setPlayerPosition(Point) void
+ setPlayerStatistics(Integer) void
+ shootBall() void <<abstract
+ toString() String <<abstract
SoccerBallTest
+ SoccerBallTest()
- soccerBall SoccerBall
+ getSoccerBall() voi
+ inGate() voi
+ moveBall() voic
+ moveBallY() voic
+ setUp() voic
Striker
+ Striker(String, Color)
moveDown() void
moveLeft() void
moveRight() void
moveUp() void
setItnitialPosition() void
shootBall() void
toString() String
GoalKeeper
+ GoalKeeper(String, Color)
- movementstep int
+ moveDown() voic
+ moveLeft() voic
+ moveRandomly() voic
+ moveRight() voic
+ moveUp() voic
+ setInitialPosition() voic
+ shootBall() voic
+ toString() String
PlayerCollectionIterator
+ PlayerCollectionIterator()
+ hasNext() boolean
+ next() Gameplayer
<inner class>
StrikerTest
+ StrikerTest()
- striker Striker
+ grabsBall() void
+ move() void
+ setUp() void
+ shootBall() void
+ testToString() void
GoalkeeperTest
+ GoalkeeperTest()
- goalKeeper GoalKeep
+ move() void
+ moveRandomly() void
+ setUp() void
+ shootBall() void
+ testToString() void
PlayerCollection
+ PlayerCollection()
- size int
- player List<GamePlayer>
+ add(GamePlayer) void
+ get(String) GamePlaye
+ iterator() Iterator<GamePlayer>
+ sort() void
PlayerFactory
+ PlayerFactory()
+ getPlayer(String) GamePlaye
SoccerGame
+ SoccerGame()
- timeRemaining Integer
- goal Integer
- isPaused Boolean
- isOver Boolean
PlayerCollectionTest
+ PlayerCollectionTest()
- factory PlayerFactory
- player PlayerCollection
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1


SoccerGame
+ SoccerGame()
- timeRemaining Integer
- goal Integer
- isPaused Boolean
- isOver Boolean
- PlayerCollection gamePlayers
+ automateGoalkeeeper() voi
+ getActivePlayer() Gameplay
+ getGamePlayers() PlayCollectic
+ getGoal() Integ
+ getTimeRemaining Integ
+ isOver() Boolea
+ isPaused() Boolea
+ isScored() Boolea
+ setGoal(Integer) voi
+ setOver(Boolean) voi
+ setPaused(Boolean) voi
+ setTimeRemaining(Integer) vo
- startGame() voi
PlayerCollectionTest
+ PlayerCollectionTest()
- factory PlayerFactory
- player PlayerCollection
+ get() voi
+ iterator() voi
+ setUp() voi
+ sort() voi
SoccerGameTest
+ SoccerGameTest()
- soccerGame SoccerGame
+ automateGoalkeeper() voic
+ getActivePlayer() voi
+ getGoal() voi
+ getTimeRemaining() voi
+ isOver() voic
+ isPaused() voic
+ isScored() voic
+ setUp() voic
+ startGame() voic
GamePanel
+ GamePanel()
- uiFont Font
- SoccerGame game
+ getGame() SoccerrGam
- paintBall(Graphics) vo
+ paintComponent(Graphics) voi
- paintGate(Graphics) vo
- paintGoal(Graphics) vo
- paintPausedText(Graphics) voi
- paintPenatlyLine(Graphics) vo
- paintPlayers(Graphics) voi
- paintStatistics(Graphics) voi
- paintTimer(Graphics) vo
- setupRepaint() vo
+ setupSoccerGame() voi
MenuBarListener
+ MenuBarListener(GamePanel)
- gamePanel GamePan
+ actionPerformed(ActionEvent) voi
GameListener
+ GameLisitener(GamePanel)
- gamePanel GamePane
+ keyPressed(KeyEvent) void
+ keyReleased(KeyEvent) void
+ keyTyped(KeyEvent) void
GameMenuBar
+ GameMenuBar(ActionnListener)
- createMenuItem(String, String, int, ActionListener) JMenuIte
MiniSoccerApp
+ MiniSoccerApp()
+ main(String[]) void
1
1
1
1
1
1
1
1
1
1

The most prominent structural design pattern used in this project is the MVC pattern. The model is where the bulk of the implementation lies, and it is made up of many elements of the application which the user cannot see or interact with. The model contains the main functionality of the players and the game itself. The classes in the model include: SoccerBall, SoccerGame, GamePlayer, Goalkeeper, Striker, PlayerFactory, PlayerCollection and PlayerStatistics. The view contains the main elements of the project which the user sees, such as the menu bar and the game window itself. The classes included in the view are GameMenuBar and GamePanel. The controller contains the elements of the project which detect the users inputs and allow the user to play the game and utilize the controls (such as pause/resume). The classes included in the controller are GameListener and MenubarListener.

Another design pattern that is used in our design is the factory pattern. In factory pattern, associated objects are created without exposing the creation logic to the user, yet users are able to use/control said objects through the interface. In our system, all objects which the user can see or interact with through the interface use the factory pattern. These classes include: GameMenuBar, GamePanel, GameListener, MenubarListener, SoccerBall, SoccerGame, Goalkeeper and Striker.

Lastly, the third design pattern which we used is the singleton pattern, and it is another prominent pattern in our system. In the singleton pattern, the instantiation of a class is limited to only one single instance. In our system, all instantiated objects use this pattern, and are instantiated only once each. These class objects include: PlayerFactory, PlayerCollection, Striker, Goalkeeper, SoccerBall, SoccerGame, GameMenuBar, GamePanel, GameListener and MenubarListener.

Our system makes use of several OOD principles, which include inheritance, abstraction, polymorphism and aggregation. Goalkeeper and Striker inherit from GamePlayer, which is an abstract class with abstract methods that include: moveLeft(), moveRight(), moveUp(), moveDown() and shootBall(). Goalkeeper and Striker both make use of polymorphism by overriding said inherited methods. GameMenuBar extends JMenuBar and is another example of inheritance. GamePanel extends JPanel and is also an example of inheritance. GamePanel also makes use of polymorphism by overriding the paintComponent(...) method which it inherits from JPanel. PlayerCollection, PlayerCollectionIterator, GameListener and MenubarListener implement the interfaces Iterable, Iterator, KeyListener and ActionListener. These four classes included in our project are our only instances of interface inheritance.

Part III: Implementation of the Solution

In our model, there is an abstract class called GamePlayer. It has a playerName, playerColor, playerPosition, and playerStatistics, which all have getters and setters. It contains abstract methods for player movement and shooting the ball. It also contains methods for checking if the player has the ball, and holding the ball. GamePlayer has an abstract toString() method, and it also has two child classes, Goalkeeper and Striker. Striker's shootBall() method is implemented to shoot up, while GalKeeper's shootBall() method shoots down. Goalkeeper's toString() method returns the number of blocks the goalkeeper made, while Striker's toString() method returns the number of goals made by the striker. Goalkeeper also contains a moveRandomly() method, which makes the goalkeeper move side to side at the soccer net.

There is a PlayerStatistics class, and its only purpose is to return an Integer instance that tells the player's stats. If the GamePlayer is a striker, it will be goals, but if the GamePlayer is a goalkeeper, it will be shots blocked. There is also a PlayerCollection that stores players. It has a private size, and allows you to add and get players. It sorts players based on the GamePlayer's compareTo() method, which compares players based on their playerStatistics. There is also a PlayerCollectionIterator class, stored in the same Java file as PlayerCollection. If iterator() is called on PlayerCollection, it will return a new PlayerCollectionIterator instance.

There are two more classes in our model, SoccerGame and SoccerBall. SoccerGame is responsible for running the game. It keeps track of the time, goals scored, game being paused, and ending the game. SoccerBall has a position which has a getter and setter, and a velocity which only has a setter. It has methods for movement, resetting soccer ball position, checking if the ball has made it past the goalkeeper side, and checking if the ball has made it in the net.

In our view, we have only two classes, GameMenuBar and GamePanel. GameMenuBar is the bar that has all the buttons for controlling game flow, such as "New game", "Exit", "Pause", and "Resume". GamePanel is responsible for drawing all the objects on the screen, such as the soccer net, field, players, and ball. It is also responsible for creating the pause text and game over screen.

In the controller, there are only two classes, called GameListener and MenubarListener. The MenubarListener responds to events when the player clicks any of the buttons on the GameMenuBar, and then acts accordingly. The GameListener handles the controls of the striker. The user uses the arrow keys to move and the spacebar to shoot the ball, and the GameListener will respond to those user inputs.

The JUnit test for the SoccerBall test if the player has the ball, and if the ball is inside the soccer net. The SoccerGame JUnit test checks if the goals are being recorded, if the game pauses correctly, if the time is being counted down correctly, and if the players are in their correct places. The JUnit test for Goalkeeper checks if the goalkeeper is moving correctly, if it can shoot, and if it is displaying its statistics correctly. The JUnit test for Striker is similar, but it also checks if the Striker is grabbing the ball correctly. Finally there is the PlayerCollection JUnit tests, which checks if the sort() method is working, and the iterator is correctly going through the elements in the collection.

Eclipse was the IDE which we used for implementing our project. We also relied solely on the JUnit testing framework to achieve testing coverage on our implementation. Diagrams.net was used for creating the UML diagram.

Part IV: Conclusion

We worked quickly to make things testable and had our game running in only three days. Also we discussed issues frequently regarding our project and we were able to solve them.
One thing we figured out at the start was the workflow which is how well we can work together for the accomplishment of the given task. Teamwork and collaboration are always closely associated with one another to produce the same measurable results. We had the opportunity to learn about technical limitations based on the experience when working on the project and got some hints of what motions might be monotonous.

Because of limited time, we were not able to find the best way to work on the project. We only found the fastest way to deal with it due to the fact that we couldn’t even pay attention to the details that add “fun” elements to the game. This could have been easier with a flowchart and well defined pictures. We didn't follow the design. We made a lot of mediocre things instead of going for a few excellent things or a well designed pattern which had adverse effects on the program pipeline. As a result, the final result was quite different from initial expectations.

We’ve learned how to break down complex tasks into parts and steps, we also improved on our planning and time management skills. We have gained skills that are relevant to both group and individual work, including the ability to refine understanding through group discussion and explanations. We‘ve developed new approaches for resolving differences, supporting and how to encourage ourselves while working on tough or complex projects. We also learned to find effective peers to emulate, share diverse perspectives, delegate roles and responsibilities, hold one another accountable and to pool knowledge and skills.

One advantage of completing a lab in a group is the ability to give ideas to each other and collaborate on what kind of design you want to do. Another advantage is when the lab is bigger and more complex, you can divide the work between group members and complete tasks more efficiently. One disadvantage is that sometimes group members struggle with interpreting each others’ code, so communication is key to overcoming this difficulty. Another disadvantage would be group members having different schedules and personal responsibilities, which makes it more difficult to plan and synergize accordingly.

Our top three recommendations to ease completion would be 1) use a version control tool such as GitHub or GitLab, so all group members can access source code easily 2) use an online messaging tool such as Discord or Slack for efficient communication between group members, and 3) always follow up with your group members’ progress.

Group Member	**Tasks**
Wenjing Qu	Implementation, Design, Report
Roberto Shino	Report, Documentation
Arian Mohamad Hosaini	Implementation, Report
Adrian Koduah	Design, Report