EECS3311 Project 2
David Zakharian – 217117037 Sec A Lab 01,
Orion Nelson – 216305377 Sec A Lab 01,
Dennis Johnson – 210666915 Sec A Lab 01,

EECS 3311 Lab:
Second Software Project – Mini Soccer Game

EECS3311 Project 2
David Zakharian – 217117037 Sec A Lab 01,
Orion Nelson – 216305377 Sec A Lab 01,
Dennis Johnson – 210666915 Sec A Lab 01,

Contents

EECS3311 Project 2
David Zakharian – 217117037 Sec A Lab 01,
Orion Nelson – 216305377 Sec A Lab 01,
Dennis Johnson – 210666915 Sec A Lab 01,

Part 1: Introduction

For this software project, the goal is to develop a functioning mini soccer game with rules specified in Part II: Design. The application itself will have two menus. With the Control menu, the game can be paused and resumed. The Game menu allows creating a new game or exiting the current one. The game will feature a striker and goalkeeper. The former will try and shoot the ball in the net while the latter will try to catch it and throw it back. The two players can't cross the penalty line. Using arrow keys, the sticker can move. Spacebar can be used to shoot. The goalkeeper moves randomly. Countdown till end of game will be displayed with the application in seconds and total goals made during the period. The striker will have 60 seconds to try and shoot as many times into the net. Each time a shot is successful, the goal count will be increased by one.

Challenges associated with this software project include not only coordinating the different classes to interact with each other to produce the final outcome, but also coordinating as a group to develop the project. Although the MVC (Model-View-Control) architecture helps sort classes, there could be issues figuring out how the classes interact. Drawing out and designing out the classes in Draw.IO became useful to figure this out. Unlike solo projects, this one requires continuous communication between the team members to successfully complete it. There could be disagreements about design until we reach consensus.

Some of the Object-Oriented Principles (OOP) used to carry out the software project include *polymorphism* and *inheritance*. The creational design patterns we used were the factory pattern and singleton pattern. We also used a structural design pattern, composite pattern, help us design the menu in the final implementation of the soccer game. For our project:

- The factory pattern: Was used for generating game players (goal keeper and player.)
- The singleton pattern: Was used for creating a soccer ball as only one is needed.
- The composite pattern: Was used for to create the menu bar and its respective components

These will be discussed in further detail in part II of our report

As this project was developed, the report will reflect the stages of development from design to implementation and modification. The final application will be created to meet the goals outlined at the start of this section. This paper will also highlight what went right, wrong, and what could have been done better to approach the task required.

Part II: Design

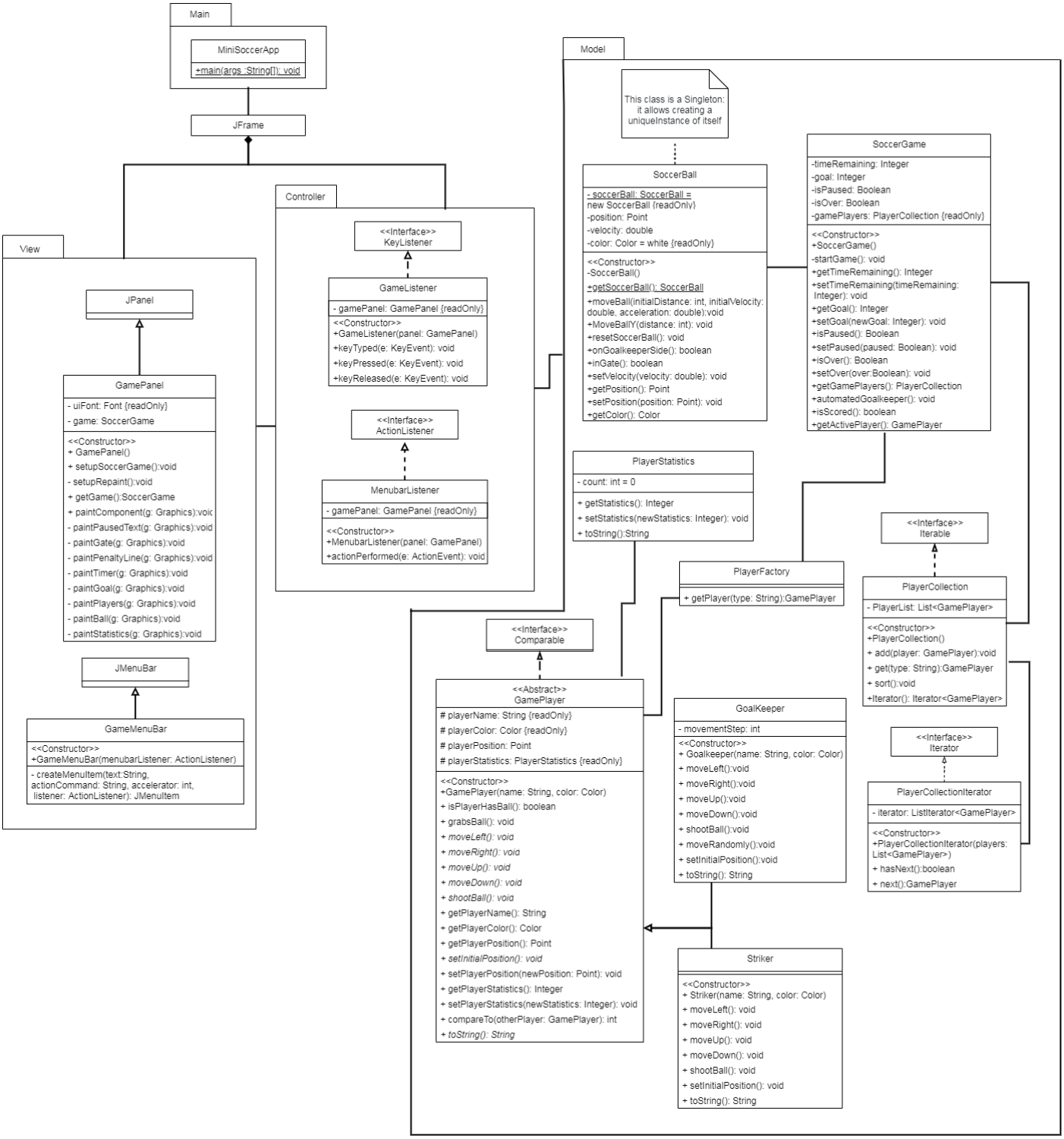

Figure 1: UML Diagram of Mini-Soccer Game showing the MVC components (drawn in Draw.io)

EECS3311 Project 2
David Zakharian – 217117037 Sec A Lab 01,
Orion Nelson – 216305377 Sec A Lab 01,
Dennis Johnson – 210666915 Sec A Lab 01,

As mentioned in Part I, there are at least three design patterns shown in the Unified Model Language (UML.)

Singleton Pattern:

The *SoccerBall* class is an example of the singleton pattern since it was used to represent the soccer ball. At any given point, only one soccer ball is used and it was *statically* implemented. This pattern allows creating only one object without instantiation of the class object. The benefit of this was it allowed to ensure only one ball was present and it also allowed us to utilize it in our factory pattern. See Figure 2 (shown below) to see how it was implemented in our code:

```
10 public class SoccerBall {
11
12     /**
13      * Initialize a soccer ball as per pattern.
14      */
15     private static final SoccerBall soccerBall = new SoccerBall();
```

Figure 2: Example of Singleton Pattern

Factory Pattern:

The factory pattern was used to create game players for the *SoccerGame*. It requests a concrete type of abstract class *Gameplayer*'s children such as *Striker* and *GoalKeeper*. Thus, different players were created without exposing creation logic. If needed, the Factory pattern could also help our team develop other types of players such as referees or fans. See Figure 3 (show below) to see how it was implemented in our code:

```
10 public class PlayerFactory {
11
12     /**
13      * Factory method to create an instance of player requested.
14      * @param type is the type of player desired.
15      * @return GamePlayer requested if available. Otherwise null.
16      */
17     public GamePlayer getPlayer(String type) {
18         if(Objects.equals(type, "striker")) {
19             return new Striker("Striker", Color.BLUE);
20         }
21         else if(Objects.equals(type, "goalkeeper")) {
22             return new Goalkeeper("Goalkeeper", Color.YELLOW);
23         }
24         return null;
25     }
```

Figure 3: Example of Factory Pattern

Composite Pattern:

The composite pattern was used to create the Menu bar and it's sub-elements: "*New Game"*, "*Exit"* in "*Game"* and "*Pause"*, "*Resume"*, in "*Control"*. As shown in Figure 4 (shown below), the code allows the client/user to manipulate the program and meta-aspects of the game. The leaves in this pattern are "New Game", "Exit", "Pause", "Resume" while the Composite class here is the Menu Bar.

```
19     public GameMenuBar(ActionListener menubarListener) {
20         super();
21         JMenu gameMenu = new JMenu("Game");
22         gameMenu.add(createMenuItem("New game", "NEW", KeyEvent.VK_N, menubarListener));
23         gameMenu.addSeparator();
24         gameMenu.add(createMenuItem("Exit", "EXIT", KeyEvent.VK_Q, menubarListener));
25         super.add(gameMenu);
26         JMenu controlMenu = new JMenu("Control");
27         controlMenu.add(createMenuItem("Pause", "PAUSE", KeyEvent.VK_P, menubarListener));
28         controlMenu.add(createMenuItem("Resume", "RESUME", KeyEvent.VK_R, menubarListener));
29         super.add(controlMenu);
30     }
```

```
40     private JMenuItem createMenuItem(String text, String actionCommand, int accelerator
41         JMenuItem menuItem = new JMenuItem(text);
42         menuItem.setActionCommand(actionCommand);
43         menuItem.addActionListener(listener);
44         menuItem.setAccelerator(KeyStroke.getKeyStroke(accelerator, 0));
45         return menuItem;
46     }
```

Figure 4: Example of Composite Pattern (Composite and generic leaf)

Implementation of Patterns:

To use all these patterns, OOP such as inheritance, polymorphism, encapsulation and abstraction were used. For example, *GoalKeeper* and *Striker* inherit methods and properties from the abstract class *GamePlayer* while adding their own properties and methods. Polymorphism is used when the class, *PlayerCollection,* can insert *GoalKeeper* or *Striker* when it accepts a GamePlayer class with method add. Abstraction is used when the game requests a player using the factory method as there is no exposed creation logic. The class, *SoccerGame,* doesn't know how players are created, but it knows if a certain type of player is requested from PlayerFactory class, a matching object will be created. Lastly, encapsulation is used with the class, *SoccerGame,* as fields were made private and only methods can be used to access and modify.

We included additional classes in the model as mentioned in the requirements:

- GamePlayer class: Abstract class which allows to define new players
- Goal Keeper class: An implementation of Game Player class and inherits GamePlayer properties and methods. This class contains the goal keeper's attributes and operations
- Striker class: An implementation of the Game Player class and inherits GamePlayer properties and methods. This class contains the striker's attributes and operations.
- PlayerCollection class: An iterable collection of players which kept track of all players and had the ability to add/remove players from the collection if required.
- PlayerFactory class: Creates player types for the soccer game. Can also used to create players dynamically
- PlayerCollectionIterator: Allows manipulation of players stored in the form a list
- PlayerStatistics: A class that keeps track of player statistics such as balls caught or scored.

Part III: Implementation

Tools:

The project was implemented using a combination of tools:

- GitHub – Software Version Control (SVN)
- Diagrams.net – Unified Modelling Language (UML)
- Maven 3.8.1– Project Management Tool
- Junit 5.7.0– Unit Testing Framework
- Jacoco 0.8.7 – Java Code Coverage
- Eclipse 20210910-1417 – Java Integrated Development Environment (IDE)
- Discord 103871 – Communication
-

Process:

Once the code was uploaded in GitHub, Eclipse was configured to use GitHub as a Software Version Control (SVN) to keep track of code/issues.

All the additional classes requested (and described in Part 2) were implemented using the tools mentioned above. We implemented the classes from main onwards and gradually added code to controller, model and view classes progressively. Once the skeleton structure was working, we added the PlayerStatistics, PlayerFactory, PlayerCollectionIterator and PlayerCollection (in that order.) The initial builds allowed us to identify points of failure and fix them accordingly. For example:

- SoccerGame also required:
  - getPlayer() to be implemented for PlayerFactory (returns player type)
  - getString() of PlayerCollection (returns player based on string input)
  - add(obj player) of PlayerCollection (add players of player type)
- GamePlayer also required:
  - getStatistics (get current score)
  - setStatistics (set current score)
- GamePanel:
  - A sort in Player collection (to ensure the winning player was listed first)

Once the code was completed, we used Junit to test and make sure the classes we implemented work as expected. We added the test package (shown below) so we had a total of 5 classes (model, view, controller, main and test):

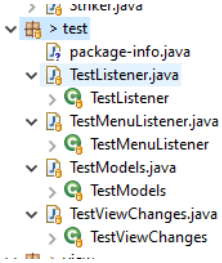

Figure 5: Test package using JUnit5 to test various components of the project

We then used Jacoco to measure code coverage. We achieved a 77.3% coverage and started working on the JavaDocs and report.

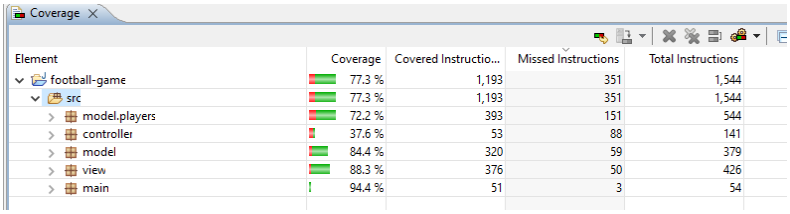

Figure 6: Jacoco manual test output

The entire process (compile, build, test and generate docs) was automated using Maven. By using option, "Maven Install",

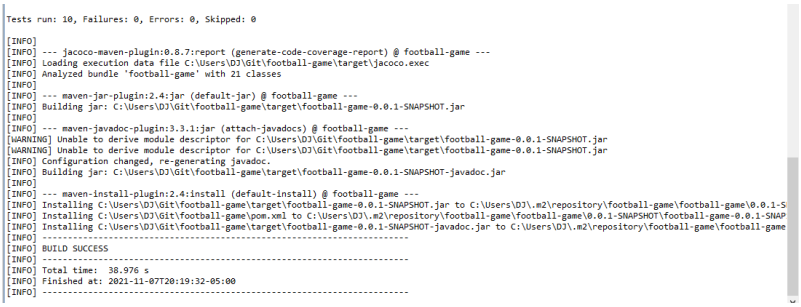

Figure 7: Output of using Maven Install using Maven & pom.xml

The Java Docs were re-generated using Maven and the build output can be found here in this folder:

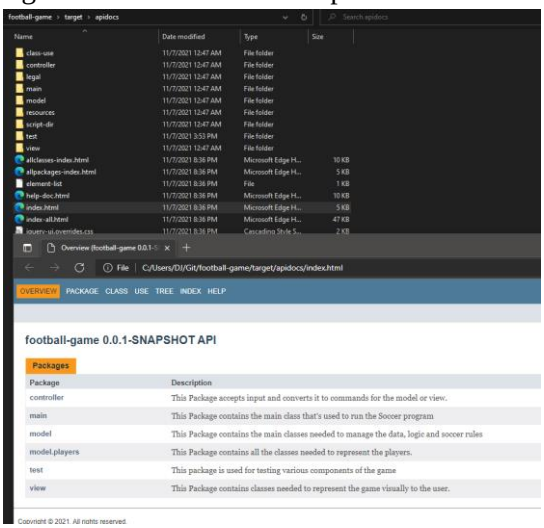

Figure 8: JavaDocs describing the packages and their contents

Finally, we were able to create an application that does the following:

- Create an application that displays an interface with two menus: Game and Control
    - Game Menu allows starting a new game or exiting the game
    - Control allows pausing and resume game
- Create an interface that displays two game players: a strike and a goal keeper.
- The interface always displays the time remaining (60 seconds onwards and ends when the timer reaches 0.
- The goal keeper is always in front of the gate while the player can try to score while the goal keeper score. The goal keeper moves randomly to left or right trying to catch the ball
- The arrow keys allow movement of the striker while the goal keeper is moving randomly and the striker/goal keeper cannot cross the penalty line.
- End the game with the remaining time is 0
- Pause the game and ensure that neither the game player or ball can move
- When the game is finished, the statistics of each game player is sorted and displayed while disabling pause and resume until a new game is started.
- The video can be found here: https://youtu.be/2aL0P4bcZAM

Part IV: Conclusion

What went well in the software project?

Overall, the project was collaborative in the sense each group member worked on what they were strongest in and contributed where possible.

What went wrong in the software project?

We were unable to achieve 100% code coverage due time constraints even though we implemented most of the testing.

What have you learned from the software project?

Designing, implementing and testing a software project from start to finish is a complicated issue with multiple people and multiple requirements.

What are the advantages and drawbacks of completing the lab in group?

Some of the advantages of working in a group is that allows the strongest people take charge of what's easiest for them but it also discourages the weaker members from fully understanding what's required or how to implement it.

What are your top three recommendations to ease the completion of the software project?

Separation of code implementation, code testing and documentation by different team members alone with separate deadlines for each instead of a combined deadline for everything would help improve work flow and give each team member a chance to contribute more.

Task Output

	Design / (due)	Implementation / (due)	Testing / (due)	Documentation / (due)
David Z	x	x		
Orion N			x	
Dennis J				x
Rongfie Bian	N/A	N/A	N/A	N/A

Deadlines

	Design / (due)	Implementation / (due)	Testing / (due)	Documentation / (due)
David Z	Nov 3, 21	Nov 3, 21		
Orion N			Nov 7, 21	
Dennis J				Nov 7, 21
Rongfie Bian	N/A	N/A	N/A	N/A

We had one member in our team who initially joined our Discord group but he did not contribute afterwards and we had no way of contacting him as we had no contact information for him.