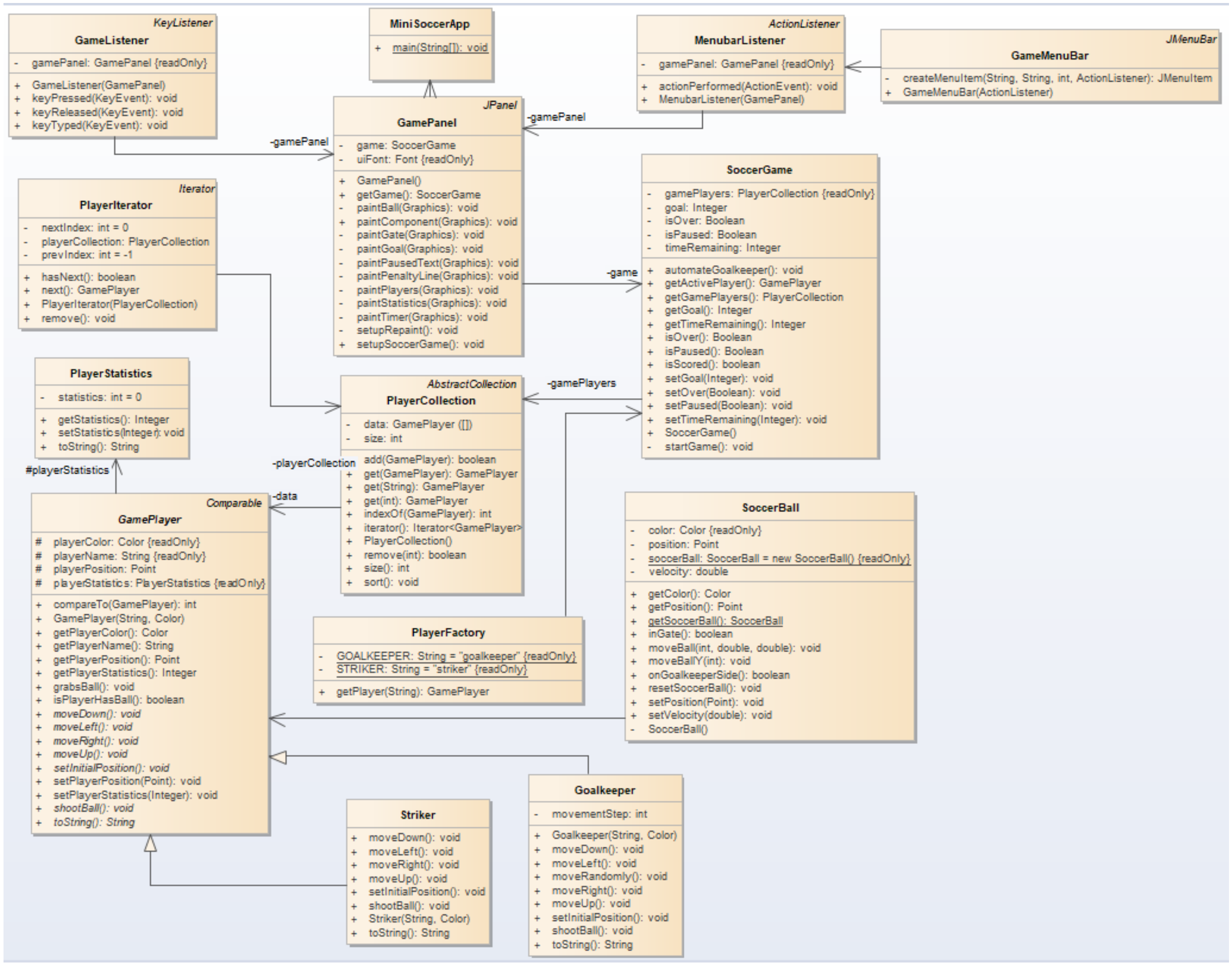
Andrew Hocking, 215752835
Deep Patel, 216519126
Ohm Patel, 215296814
Rahim Sayani, 217005364

Part I: Introduction

This software project consists of implementing the model component of a Mini Soccer Game. The goal of this project is to apply learned Object-Oriented Design principles and design patterns to implement missing model classes in the Mini Soccer Game. The challenges associated with this software project are determining the functionality and relationships of the unimplemented classes, as well as implementing these classes. The OOD principles used throughout the software project are Encapsulation, Polymorphism, and Inheritance. Encapsulation is the privatization of class variables to prevent access and modification from outside of their respective classes. Polymorphism occurs when methods in a child class override the functionality of methods in the parent class. Polymorphism only occurs in an inheritance relationship. Inheritance is a relationship between a parent class and a child class, whereby the child class inherits and extends the functionality of the parent class. The design patterns implemented in the software project are the Factory pattern and Singleton pattern. The Factory pattern creates objects without exposing the creation logic to clients. The Singleton pattern ensures that only one instance of a class is created by privatizing the class constructor and maintaining reference to the single instance within the class. This report will be structured into four parts; an introduction, design of the solution, implementation of the design, and conclusion.

Part II: Design of the Solution

UML Class Diagram


KeyListener
GameListener
- gamePanel: GamePanel {readOnly}
+ GameListener(GamePanel)
+ keyPressed(KeyEvent): void
+ keyReleased(KeyEvent): void
+ keyTyped(KeyEvent): void
MiniSoccerApp
+ main(String[]): void
ActionListener
MenubarListener
- gamePanel: GamePanel {readOnly}
+ actionPerformed(ActionEvent): void
+ MenubarListener(GamePanel)
JMenuBar
GameMenuBar
- createMenuItem(String, String, int, ActionListener): JMenuItem
+ GameMenuBar(ActionListener)
-gamePanel
JPanel
GamePanel
- game: SoccerGame
- uiFont: Font {readOnly}
+ GamePanel()
+ getGame(): SoccerGame
- paintBall(Graphics): void
+ paintComponent(Graphics): void
- paintGate(Graphics): void
- paintGoal(Graphics): void
- paintPausedText(Graphics): void
- paintPenaltyLine(Graphics): void
- paintPlayers(Graphics): void
- paintStatistics(Graphics): void
- paintTimer(Graphics): void
- setupRepaint(): void
+ setupSoccerGame(): void
-gamePanel
Iterator
PlayerIterator
- nextIndex: int = 0
- playerCollection: PlayerCollection
- prevIndex: int = -1
+ hasNext(): boolean
+ next(): GamePlayer
+ PlayerIterator(PlayerCollection)
+ remove(): void
SoccerGame
- gamePlayers: PlayerCollection {readOnly}
- goal: Integer
- isOver: Boolean
- isPaused: Boolean
- timeRemaining: Integer
+ automateGoalkeeper(): void
+ getActivePlayer(): GamePlayer
+ getGamePlayers(): PlayerCollection
+ getGoal(): Integer
+ getTimeRemaining(): Integer
+ isOver(): Boolean
+ isPaused(): Boolean
+ isScored(): boolean
+ setGoal(Integer): void
+ setOver(Boolean): void
+ setPaused(Boolean): void
+ setTimeRemaining(Integer): void
+ SoccerGame()
- startGame(): void
-game
PlayerStatistics
- statistics: int = 0
+ getStatistics(): Integer
+ setStatistics(Integer): void
+ toString(): String
AbstractCollection
PlayerCollection
- data: GamePlayer ([])
- size: int
add(GamePlayer): boolean
+ get(GamePlayer): GamePlayer
+ get(String): GamePlayer
+ get(int): GamePlayer
+ indexOf(GamePlayer): int
+ iterator(): Iterator<GamePlayer>
+ PlayerCollection()
+ remove(int): boolean
+ size(): int
+ sort(): void
-gamePlayers
-playerCollection
#playerStatistics
-data
Comparable
GamePlayer
# playerColor: Color {readOnly}
# playerName: String {readOnly}
# playerPosition: Point
# playerStatistics: PlayerStatistics {readOnly}
+ compareTo(GamePlayer): int
+ GamePlayer(String, Color)
+ getPlayerColor(): Color
+ getPlayerName(): String
+ getPlayerPosition(): Point
+ getPlayerStatistics(): Integer
+ grabsBall(): void
+ isPlayerHasBall(): boolean
+ moveDown(): void
+ moveLeft(): void
+ moveRight(): void
+ moveUp(): void
+ setInitialPosition(): void
+ setPlayerPosition(Point): void
+ setPlayerStatistics(Integer): void
+ shootBall(): void
+ toString(): String
SoccerBall
- color: Color {readOnly}
- position: Point
- soccerBall: SoccerBall = new SoccerBall() {readOnly}
- velocity: double
+ getColor(): Color
+ getPosition(): Point
+ getSoccerBall(): SoccerBall
+ inGate(): boolean
+ moveBall(int, double, double): void
+ moveBallY(int): void
+ onGoalkeeperSide(): boolean
+ resetSoccerBall(): void
+ setPosition(Point): void
+ setVelocity(double): void
- SoccerBall()
PlayerFactory
- GOALKEEPER: String = "goalkeeper" {readOnly}
- STRIKER: String = "striker" {readOnly}
+ getPlayer(String): GamePlayer
Goalkeeper
- movementStep: int
+ Goalkeeper(String, Color)
+ moveDown(): void
+ moveLeft(): void
+ moveRandomly(): void
+ moveRight(): void
+ moveUp(): void
+ setInitialPosition(): void
+ shootBall(): void
+ toString(): String
Striker
+ moveDown(): void
+ moveLeft(): void
+ moveRight(): void
+ moveUp(): void
+ setInitialPosition(): void
+ shootBall(): void
+ Striker(String, Color)
+ toString(): String

Elements of the UML Class Diagram

MiniSoccerApp:	The main class. It constructs the game panel and starts the application.
GamePanel:	A view of MVC. It constructs the game window.
GameMenuBar:	A view of MVC. It constructs the menu bar.
SoccerGame:	A model of MVC. Its role is to count time, and goals.
SoccerBall:	A model of MVC. It follows the Singleton pattern.
GamePlayer:	A model of MVC. It is an abstract class representing the players.
Goalkeeper, Striker:	Models of MVC. These are both subclasses of GamePlayer.
PlayerStatistics:	A model of MVC. It is a data class for storing goals/saves by players.

PlayerFactory:	A model of MVC. It creates GamePlayer instances using the Factory pattern.
PlayerCollection:	A model of MVC. It is a collection class that holds GamePlayer objects.
PlayerIterator:	A model of MVC. An Iterator subclass for the PlayerCollection class.
GameListener:	A controller of MVC. It processes the gameplay keyboard events.
MenubarListener:	A controller of MVC. It processes the menu items' click and keyboard events.

Design Patterns Used

Factory Pattern

The Factory pattern is used in the class PlayerFactory to create and return instances of the classes Striker and Goalkeeper. The method getPlayer() in the PlayerFactory takes a String input, determines whether a Striker or Goalkeeper is being requested, then creates and returns a new instance of the object. The method converts the String input to lowercase using Java's toLowerCase() function. Any whitespaces in the beginning and end of the String are removed using Java's trim() function. The String input is compared to the String "striker" and "goalkeeper" using a switch statement. If the String input matches "striker", a new Striker object is created and returned. If the String input matches neither "striker" or "goalkeeper", an IllegalArgumentException is thrown and the message *Invalid player type* is displayed.

Singleton Pattern

The Singleton pattern is used in the class SoccerBall. The SoccerBall class has a private constructor and maintains a reference to a single SoccerBall object. The SoccerBall class is instantiated by calling the method getSoccerBall() which returns the single instance of the class. This design pattern ensures that only one SoccerBall instance is created and used throughout the application.

OOD Principles Used

Encapsulation

Encapsulation is implemented in the classes PlayerFactory, PlayerStatistics, PlayerCollection, and PlayerIterator. The PlayerFactory has private class variables *striker*, and *goalkeeper*. The PlayerCollection has private class variables *data*, and *size*. The PlayerStatistics has a private class variable *statistics*. The PlayerIterator has private class variables *nextIndex*, *prevIndex*, and *playerCollection*. These variables can only be accessed and modified within their respective classes. Classes that access and modify these variables must use getter and setter methods instead of directly accessing the variables.

Polymorphism

Polymorphism is used frequently throughout the application, whenever the abstract class GamePlayer is referenced. Striker and Goalkeeper are both subclasses of GamePlayer, and are usually used with a GamePlayer reference, such as in the PlayerCollection class and in the PlayerFactory class. The overridden methods in the Striker and Goalkeeper classes are run in place of the abstract methods in GamePlayer when the object calls these methods.

The PlayerCollection overrides the size(), iterator(), and add() methods of the parent class AbstractCollection<T>. The methods override their implementation in the parent class with the *@Override* annotation. This form of method overriding results in run-time Polymorphism. The size() method returns a private class variable size representing the number of players in the game. The add() method checks if the GamePlayer array is full and creates a new array with double the capacity accordingly. The method adds the GamePlayer object parameter to the array. The method iterator() returns an instance of the PlayerIterator class and instantiates it with the PlayerCollection instance (itself).

Abstraction

Abstraction is used in a few places in this application. In PlayerCollection, the *data* array and *size* variable are completely private, so there is no way to directly access either of them outside of the PlayerCollection class itself; they can only be modified indirectly through other public methods like *add* and *remove*. Another example is in Goalkeeper, where the *movementStep* variable is private with no way to interact with it outside of the class. None of these variables have getter or setter methods, they can only be modified from within their respective classes.

Inheritance

Inheritance is evident in the relationship between the PlayerCollection subclass and AbstractCollection<T> superclass. The PlayerCollection subclass extends the functionality, and inherits the methods of the AbstractCollection<T> superclass. PlayerCollection extends the functionality by implementing methods get(GamePlayer), get(String), get(Integer), and sort(). The PlayerCollection inherits and overrides the methods size(), iterator(), and add().

Inheritance is also evident in the relationship between the Striker, Goalkeeper, and GamePlayer classes. The Striker and Goalkeeper subclasses inherit methods and attributes of the GamePlayer superclass. The Striker and Goalkeeper extend the functionality of the GamePlayer superclass by maintaining distinct methods and attributes, as well as overriding superclass methods such as moveLeft(), moveRight(), moveUp(), moveDown().

Part III: Implementation of the Solution

The tools used to complete the coding portion of this software project were Eclipse version 2019-12 (4.14.0) running on macOS 11.2 as well as the *EclEmma Java Code Coverage 3.1.3* JaCoCo plug-in for Eclipse.

Video demo of how to launch and run the application:
https://photos.app.goo.gl/32kv9tAmkncp5H1E8

Descriptions of Newly-Implemented Classes

Below are expanded descriptions of all the newly-implemented classes. The Javadoc for this application can be found here:
https://github.com/AndrewHocking/EECS-3311-Lab-5/tree/main/doc

PlayerFactory

PlayerFactory is a factory method that is used to create and return instances of the Striker and Goalkeeper class. The purpose of this class is to hide the creation logic from clients. The class maintains two private, unmodifiable Strings that are used to determine which object to instantiate. This class contains one method getPlayer() which returns an instance of the Striker or Goalkeeper class depending on the String parameter, or throws an exception if no match is found. A detailed description of this class can be found in the Factory Pattern subsection.

PlayerCollection

PlayerCollection maintains a private array of GamePlayer objects and a private integer representing the size of the array. The class has a constructor that initializes the array with a default length two, and the integer value with a default value zero. The class contains three getter methods for the GamePlayer objects stored in the array. These getter methods can search and return a GamePlayer object based on the playerName, instance of the GamePlayer object itself, or the index at which the object is stored. The class contains a sort() method which utilizes Java's Arrays.sort() method. The class overrides methods size(), iterator(), and add() within the AbstractCollection<T> class. The class contains an indexOf() method which returns the index of the GamePlayer object parameter, or -1 if not found. The class also contains a remove() method which takes an integer parameter and deletes the object at that index from the array. This method returns true if an object was removed and false otherwise.

PlayerStatistics

PlayerStatistics maintains a private integer variable representing the number of goals scored or goals caught depending on the object that instantiates the class. The class contains a pair of getter and setter methods to access and modify the private integer variable. The class also contains a toString() method used to convert the integer variable to a String so that it can be printed on the GUI.

PlayerIterator

The PlayerIterator implements the Iterator<T> interface. The class contains a constructor, and methods hasNext(), next(), and remove(). This class maintains a private PlayerCollection object, and two private integers representing the next index and previous index of the current position. The hasNext() method returns a boolean value true if the next index is empty and false if the next index is out of bounds. The next() method returns the GamePlayer object at the next index, or throws an exception accordingly. The remove() method removes the object at the current index (between the previous index and next index), or throws an exception accordingly.

Code Coverage

The JaCoCo code coverage in this project is 94.7%. We created seven JUnit tests, which cover the scope of the application's features – both the ones that are visible to the end user and the ones that are not. The 5.3% of the code that is not covered is all error-handling code, with the exception of the driver method, MiniSoccerApp.main. This method is not tested because we use a different, nearly-identical method to launch the app for the tests, in order to maintain a reference to the GamePanel instance in use, which allows us to test the underlying game.

The JaCoCo report can be found here:
https://github.com/AndrewHocking/EECS-3311-Lab-5/blob/main/JaCoCo_Report/

Below are the descriptions of the test cases that have been implemented:

scoreGoalTest

This test ensures that when a goal is scored, the point is counted, the game is paused, and the players and the ball revert to their original positions.

failGoalsTest

This test ensures that when a goal is not scored, either by missing the net or kicking the ball into the goalkeeper, no point is counted, the game is not paused, and nothing is reset.

pauseResumeTest

This test ensures the pause and resume functions are working properly regardless of the current game state (already paused, already resumed, game is over, game is in normal state).

newGameTest

This test moves the players and the ball, tallies up some goals and some saves, then starts a new game, to check that everything is properly reset (players and ball return to starting position, goals and saves revert to zero, timer resets to 60) when a new game is started.

movePlayersTest

This test attempts to move each player in each direction, and makes sure the player moves accordingly.

collectionAddRemoveTest

This test creates new players using PlayerFactory, stores them in the game’s existing PlayerCollection, and removes several players from the collection. It verifies the collection as it does this.

collectionRetrievalTest

This test verifies the get functions from PlayerCollection, by attempting both valid and invalid retrievals and ensuring the proper errors and null values appear.

Part IV: Conclusion

Elements of this software project that went well were the implementation of the PlayerStatistics, PlayerCollection, and PlayerFactory classes, as well as creating the UML class diagram. The functions of these classes were simplistic, thus they were easily implemented. However, the PlayerIterator class was challenging since it required implementing the Iterator<T> interface. Another challenge of this software project was achieving a code coverage of 80% or higher. This required an understanding of the model, view, and controller classes, as well as the creation of JUnit tests to prove these classes were functioning correctly. From this software project we have learned that we should understand the classes and interfaces we are implementing beforehand, and follow the Object Oriented Design and Analysis steps. These steps include planning ahead and creating UML class diagrams before code is written so that all group members have a fuller understanding of the application and its components. An advantage to completing the lab in a group is that larger and more ambitious programming projects can be pursued as a result of the delegation of work across several group members. Working in groups also provides effective training on how to delegate tasks and meet deadlines when working on software projects. Students are exposed to a collaborative work environment which is more realistic in the workplace, as opposed to an isolated one person operation. The drawbacks of working in groups is that delegation of work through online communication channels is difficult, and delays are frequent between responses from group members. As a result of this online asynchronous work environment, students are not held to a schedule which makes communication challenging. Our top three recommendations to ease the completion of this software project is to first establish communication and second delegate work with group members as soon as possible. The scope of this software project is larger than the first, and group members should be allotted a fair amount of time to complete their part and be given time in case any personal matters arise. The third recommendation is for every group member to contribute to or understand the coding component of the assignment. This portion is crucial to completing the report, video presentation, and UML class diagram of this software project.

Andrew Hocking	● Report: Code Coverage ● Report: Polymorphism & Abstraction ● Java implementation of Newly-Implemented Classes ● Javadoc creation ● Creation and implementation of JUnit tests
Deep Patel	● Video explaining how to launch and run the application
Ohm Patel	● UML class diagram ● Report: Elements of the UML class diagram
Rahim Sayani	● Report: Introduction ● Report: OOD patterns used ● Report: OOD principles used ● Report: Description of Newly-Implemented Classes ● Report: Conclusion