**Mini Soccer Game Software Project**

Asanka Gunatilaka (Section B) 217499450

Faith obadun

Bansari Panchal

Janki Jadeja

T.A: Alireza Naeiji

**Part I - Introduction**

The objective of this lab is to create a mini soccer game through which a player can control the striker and score goals against the goalkeeper. The goalkeeper automatically guards the ‘gate’ by randomly moving left to right. The player wins if more goals were scored than balls were caught by the goalkeeper, by the time the timer runs out. The goal of the project is to design a UML diagram with the aid of design patterns and design principles. Then, implement the code that abides by that design. By doing so we are to get a deeper understanding of the design patterns, principles. Also, be familiar with working in a group environment, learn OO analysis and reflect on our work.

Most challenging part of the project was identifying the correct design patterns that are suitable for the project. It was also difficult to ensure the implementation matches the initial UML diagram. Furthermore, there was some difficulty in where to start the project, but the OOD workflow helped in laying some groundwork.

The design principles used are, polymorphism, inheritance, abstraction, and encapsulation. These principle gives generalized good practices and concrete advice on how to implement our code. Polymorphism is the ability of an object to take many forms. Using polymorphism, we can generate different versions of a particular object that has common features. A benefit of this is code reusability. Inheritance provides code reusability as well. Through inheritance, child objects can inherit common properties and behaviors. Encapsulation protects object internal data from the client by hiding them.

The first design patterns used is the Factory pattern. The factory pattern allows the creation of classes at run time based on the sub classes the client require. The second design pattern used is the Singleton pattern. The Singleton pattern allows only one single instance of an object and provides a global point of access to it.

The report will address each of the questions, from each section, that are required to be addressed, in the form of paragraphs. We will have an introduction section, a design, an implementation, and a conclusion section.

**Part II - Design of the solution**

Diagram, schematic

Description automatically generated

*Please find document named Diagram.drawio for the UML diagram.*

We have used the factory method design pattern. The corresponding classes for this are the PlayerFacory class, the GamePlayer class, the Goalkeeper class, the Striker class. Because we want to create two different possible players at run time, we have created the factory class to generate new sub class objects of the GamePlayer class based on the string (striker or goalkeeper) sent as an argument. This way the user does not need to know details about the sub classes of the GamePlayer class and it encapsulates the object creation.

Second design pattern used is the Singleton pattern. By using the Singleton design pattern, we are generating one instance of an object, in our case a Soccer ball (from the SoccerBall class) that lets other object that uses it have global level access. The Singleton class (SoccarBall) has a private constructor and a public accessor to allow one single instantiation. The SoccerGame class instantiates this and uses this instance in our project. Both Goalkeepr and Striker classes uses the same instance and do the manipulations it needs to do.

The design principles used are, polymorphism, inheritance, and encapsulation. Inheritance is used with classes GamePlayer, Striker and Goalkeeper as they all share some common properties. Using inheritance, striker and goalkeeper classes can inherit some common properties via the GamePlayer class. Polymorphism is also being used in the classes GamePlayer, Striker and Goalkeeper as the GamePlayer can be morphed into Striker and Goalkeeper. We have used encapsulation to restrict access to internal data and only allowing interaction through constructors and public methods. It also uses abstraction through GamePlayer class. It has fields that are shared between Striker and goalkeeper classes. Some methods are defined while some are just declared so its subclasses can define them.

**Part III - Implementation of the solution**

*Please see Github repository to find the implementation of the MiniSoccerGame application.*

In order to implement this project, we first came up with use cases to get an idea of how the application would be used and who the actors might be. This also provided us with a starting point, so we are aware what the application is required to do. Then we came up with a domain model where we were able to note down the relationships the objects and how they behave with each other. This laid the groundwork for the UML diagram that later followed. We then assigned responsibilities to the objects and designed interaction diagrams by using sequence diagrams. We drew sequence diagrams for each use case. This step drew a picture of how the interaction of our objects would work. Then came the UML diagram with complete relationships, fields and methods. In order to create the UML diagram, we used the previous work we had done with use cases, domain model and sequence diagrams. On top of this we used design patterns and design principles. Once we came up with a comprehensive UML diagram we moved on to the implementation. The implementation then was a matter of defining the methods in code and attempting to abide by the design patterns and design principles set out in the UML diagram. Once the implementation was done, we moved on to unit testing where we used Jococo to verify that we covered most of the code where we wanted to test in order to be 100% sure that the implementation is producing the results, we want it to.

Following is a documentation of all the classes and their description.

* Player statistics

Player statistics keeps track of the stats of the particular player. It initializes and maintains a private variable named ‘Goal’ with the number of goals a player has scored. To encapsulate the ‘goals’ field we have added a getStatistics() method to return the number of goals scored by a player and a setStatistics(Integer n) method to update the number of goals. We are also overidding the toString method so the number of goals can be displayed.

* PlayerFactory

PlayerFactory class generates either a Striker object or a goalkeeper object based on the request from the client. This is done by only one method named getPlayer(String).

* PlayerCollection

Player Collection class stores a collection of players using an ArrayList named playerList. Again, the ArrayList is hidden to the client. A constructor initializes the arraylist and two public methods for add and get, gives the client access to what they need. This class is made iterable through a custom iterator class so it can iterate through the arraylist. There is also a sort method that sorts the the players based on the stats.

* PlayerCollectionIterator

PlayerCollectionIterator builds a custom iterator so the PlayerCollection class can use it to be iterable. PlayerCollectionIterator is a composition of PlayerCollection. It creates a strong relationship by not using aliases. This class implements the iterator class so the required methods of hasNext(), next() are defined.

* MiniSoccerApp

The Main Soccer Game App class acts as the client that uses the classes in the MVC packages.

* SoccerBall

The soccer ball class manages the movement of the soccer ball during the game. The soccer ball is reset when the game begins.

The global variables are position and velocity.

Methods: moveBall(), resetSoccerBall(), onGoalKeeperSide(), inGate()

* SoccerGame

The Soccer game class creates the game, the collection of players. It sets up the timer and monitors it. It also monitors the pausing, resuming, goals being scored.

The global variables are timeRemaining,goal, isPaused, isOver

Methods: startGame(), getTimeRemaining(), getGoal(),isPaused(), isOver()

* GamePlayer

The game player class is an abstract class that acts as the parent class for Striker and goalkeeper classes. It shares common properties between it’s sub classes. It contracuts the base player and has defined methods that both sub classes share. There are also declared methods that it’s subclasses define.

The global variables are playerName, playerColor, playerStatistics

Methods: isPlayerHasBall(), grabsBall(), getPlayerStatistics()

* Goalkeeper

The goalkeeper class manages the movement operations such as move left to right, shooting the ball and move randomly

The global variable is movementStep

Methods: moveLeft(), moveRight(), shootBall(), moveRandomly()

* Striker

The striker class manages the movement operations such as move left to right, of the striker

The global variable is movementStep

Methods: moveLeft(), moveRight(), shootBall(), moveRandomly()

* Game Menu Bar

The game menu bar class creates a UI for the menu options and creates key event listeners so the commands can be connected through to the classes that takes the particular action.

* Game Panel
* he game panel class displays the graphical user interface of the game app showcasing player, timer, ball, penalty line and gate
* GameListener

The game Listener class captures the movement of player by using keystrokes such as move left,right,up and down

* Menu Bar Listener

The menu bar listener class manages the status of game and display message when game is paused or game finished

Using Jococo we have reached just over 80% of coverage for the model package. Please see Junit cases. JUnit cases provides us an automated mechanism to test the implementation of our code without having to manually do it. Jococo provides us with the ability to check the lines of code covered by the JUnit cases so there is a visual representation of what our testing might be missing. Achieving 100% coverage would mean that you can be fairly confident that you have tested your entire implementation. The tools and libraries used areEclipse, draw.io, JUnit and Github.Please see video file named ‘video’ to see a short visual demonstration of the application.

**PART IV - Conclusion**

Due to the strategies, we’ve learned in this class we were able to methodically approach the project by first creating use cases, then coming up with a domain model, a few sequence diagrams and then a UML diagram. With the UML diagram we were able to lay the groundwork for the structure of the project and their dependencies. Also, using design patterns we were able to get an idea of how the classes should be built and structured. We believe time management was the only thing that went wrong with the project. Although the project was progressing well throughout the days, near the deadline the consolidation of the work took more time than we realized. We have learned that perhaps in future projects that we break down tasks, allocate time slots for them and strictly adhere to them so there is better time management. We have also gotten a better understanding of how we can use design patterns for our benefit and how design principles make our implementation better. We’ve received a better understanding of the overall software OO analysis and design. The advantages of completing the lab in a group is that you get to divide the tasks between different group members which gives us enough time to thoroughly focus on the part that is assigned to you. The disadvantages are that we had a hard time figuring out how to break the tasks into sections that can be assigned to members. Also, since we are working online, we found it was somewhat difficult to communicate about the project. There is no recommendation we could provide. The skeleton code provided with this project helped a lot.

Below table indicates the different tasks that were assigned to each team member of the team.

|  |  |  |
| --- | --- | --- |
| **Name** | **Task assigned and completed** | **Collaborative?** |
| Asanka Gunatilaka | UML/Implementation/Report | Yes |
| Janki Jadeja | UML/Implementation/Report | Yes |
| Faith obadun | UML/Implementation/Report | Yes |
| Bansari Panchal | UML/Implementation/Report | Yes |