Bilkent University

Cs 319
Object-Oriented Software Engineering Project
Design Report (Final)

Section 2 / Group 2K

Ege Darçın

Alptuğ Albayrak

Özgün Özkan

Furkan Erdem

Instructor: Bora Güngören

TABLE OF CONTENTS

1) Introduction

1.1) Purpose of the System

Head Soccer is a 2-D arcade game. By designing heads of character's much bigger compared to their size, providing power ups and optimizing character's controls, main purpose is to provide users with a game in which they can have an entertaining experience. In addition, in order to provide user with more variety, thanks to random mode, different ball types, different conditions and goal sizes can be set each round. Head soccer aims to offer the players an entertaining game with variety and good design.

1.2) Design Goals

1.2.1) Well-Defined Interface

While designing the user interface, our main purpose is to make the game as desirable and enjoyable as we can. User interface plays a key role in creating a positive first impression on players as much as it directly influences how desirable the game is. Game will be consisted of two backgrounds which are normal football field and a green pitch with tribunes. In addition, players are going to be provided with a clear description of the location and the intended purpose of the buttons which aims to provide users with conveniences while playing the game.

1.2.2) Ease of Use and Learning

Easiness of usage and learning the game rules in an easier way, provides player with getting use to the game in a short time which directly effects player's entertainment while playing the game. In order to accomplish that, help section will be provided to the players which clearly enlightens players related with the purpose and the gameplay of the game.

1.2.3) Performance

Since the performance of the game, directly effects the quality, there are some certain features that our program should support such as fast loading. By coding the program wisely and creating a program with a high run time performance, we will make the game smooth which will also increase the entertainment of the game.

1.3) Maintenance

1.3.1) Portability

In a software design, portability is an essential issue which helps the program to be reached by a large number of users. In order to accomplish that, our program is going to be implemented on Java which is supported by lots of OS, devices and versions. As a result, our program will be able to play in most of the system which are suitable for running JVM.

1.3.2) Reliability

Considering the various inputs that can come from the users, our main purpose is to make sure that our program will not be crushed. After determining the possible error cases,

errors are going to be solved by using exceptions and giving notifications to the user in the case of occurrence of an error.

1.3.3) Modifiability

During the implementation stage it is inevitable to make some changes and modify the Project. Our existing functionalities will be open to some modifications. By minimizing the coupling, our aim is to make sure that a small change in the code does not affect the entire program in a considerable way.

1.3.4) Understandability

Since we have to consider the ones who uses and analyze the Project, we need to design our Project in such a way that it is understandable for anyone. By trying to express every detail in a clear and understandable way in our Project reports, we aim to accomplish this purpose. When it comes to implementation stage, we are aiming to provide some comments to express some details of our Project which will increase the understandability.

1.3.5) Extendibility

While designing a software, it is always significant to design a system which is suitable to extensions such as new functionalities (i.e. new Modes, new power-ups etc.) Providing a system which is suitable with extensions, our main purpose is to provide users with an entertaining adventure.

1.4) Trade-Offs

1.4.1) Functionality vs. Usability

Headball is easy and simple game in which players can control the footballers with arrow keys, left, up and right, or w, a and d. This game appeals players from different range of ages. Game doesn't so much functionalities and complexities in terms of playing; on the other hand, it appeals players with its simplicity. It can be easily understandable, playable. We have sacrificed more complex functionality; however, we get more funny game which doesn't require lots of mental and physical efforts.

1.4.2) Flexibility vs. Efficiency

We used open architecture (transparent layering) in which each layer can call operations from any layer. It provides run-time efficiency, while closed (opaque) architecture allows program to be more flexible and maintainable. We have sacrificed these features by choosing open architecture.

1.4.3) Performance vs. Portability

We have chosen Java for implementing our program. Java is portable language which can work on any computer with Java Virtual Machine. Java language is also known by our entire group members; writing codes wrote in different programming language and merging them would be so hard and time consuming for us, so we chose Java to implement our program. However, Java brings also some disadvantages. For instance, JAVA does not have a good running time performance compared to other languages. Game will also have 2d

graphics which allows game to be run by any computer with minimal requirements of hardware.

2) Software Architecture

2.1) Overview

This section will be mainly on structure of the system. In the first part of the report, we have identified design goals. Now, we are modeling system design as a set of subsystems. We have decomposed the overall system into manageable parts by using the principles of cohesion and coherence. This part and third part will include identification of subsystems, services, and their association to each other. Objects and classes in the previous analysis report are ‘seeds’ for our subsystems. And uses cases in previous report allow us to define services in this part.

2.2) Architecture Style

We have chosen the MVC (Model-View-Controller) architectural style. In this style, we classified subsystem into three different types.

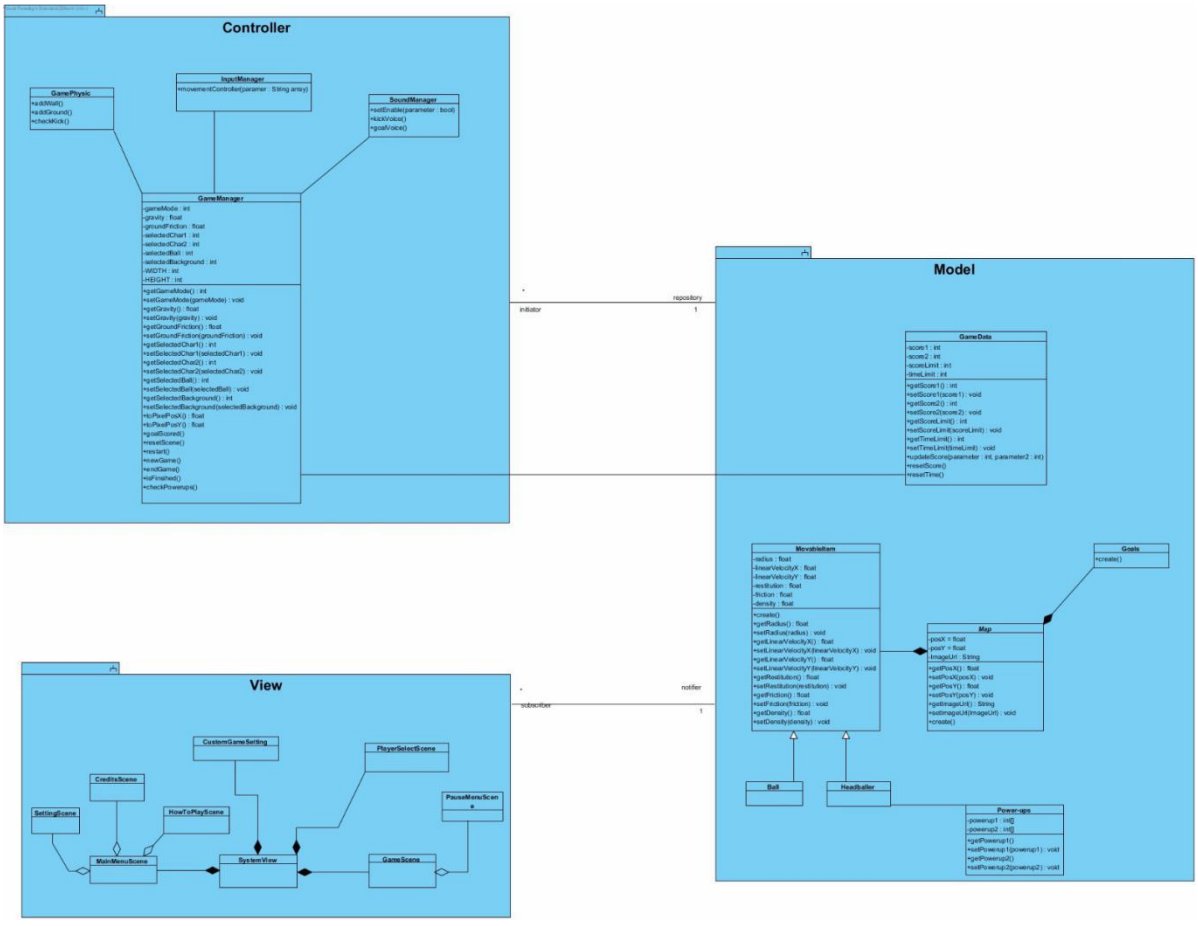
Model subsystem is responsible for application domain knowledge. This subsystem expresses the system’s behavior in terms of the application domain. View subsystem handle display related issues. Controller subsystem interacts with user, gets input and does the computations needed to be done behind and modifies both Model and View subsystems.

This style provides high coherence by allowing classes to perform similar tasks and to be related with each other, and provides low coupling by decreasing dependency between subsystems and providing that they have less information about each other. This style brings also some tradeoffs which were mentioned in the previous part.

2.2.1) Architecture Type

We have divided our subsystems horizontally into several independent subsystems. They have mutual knowledge about each other. While some of them have compile-time dependency, some of them have also run-time dependency. We also use open architecture in which each layer can call operations from any layer.

2.3) Subsystem Decomposition



2.4) Hardware / Software Management

Our game will be coded in JAVA mainly because everyone in the group has an experience with JAVA and furthermore JAVA has some very beneficial libraries to use. Images and sound effects will be stored in .png and .wav format.

On the hardware part, our game will need a keyboard and since our game's system requirements will be very low any PC with JAVA environment can run the game without any problems.

2.5) Persistent Data Management

The data for the game will be stored in the user's hard drive and the game will only store some basic stats like highest number of goals in a match, best victory etc. because of this no database will be used.

2.6 Boundary Conditions

2.6.1 Initialization

Our game will not need an installation to work. The game will work as a .jar executable file. When this executable clicked the game will be opened in a new window.

2.6.2 Termination

There are multiple ways to terminate our program. The most basic way is to simply click the "Exit" button in the main and pause menu. In addition to this you can close the program by clicking "X" at the top of the window or ending the task with task manager.

2.6.3) Failure

There are a few cases where there can be a failure, each of them will have an error message specific for that error. These failures consist of: missing program files, absence of JDK on the computer.

3) Subsystem Services

3.1) Modal

The modal component is handling ithe data of our program, where the controller manipulates it on what data is changed and what data is used for the game. After that modal updates the view component about what to display and what is changing during the execution stage. It has 7 classes. Game Data is keeping in-game variables and data. Movable item is a class which keeps the data of movable objects (ball and handballer). Ball and Handballer classes implements MovableItem. Headballer class uses power-ups to store powerup data. Goals class is checking goal statuses and is a structural object. Map class is the main class of modal which sends data to view.

3.2) View

The view controller is handling the user interface and displaying graphics. It takes the required data from Modal and displays it on the screen. Using JavaFX and JBox2d, it draws the scene and builds the world of the game. SystemView handles the main display, all classes

are connected to it. MainMenuScene class operates the main menu classes, with respect to their fxml classes. CustomGameSetting class displays the custom game settings and initiates the game accordingly to the options. Game scene displays the game and pause menu screen.

3.3) Controller

The Controller component is managing the game by manipulating the data on Model subsystem according to the user inputs. Controller controls the game values, balls, backgrounds, characters etc according to user's preferences and also controls the sounds. Controller component has 4 classes: Game Physic class controls the game environment as players move in the field, InputManager class gets the inputs from users, SoundManager class controls the sounds and GameManager class controls the variables and main flow of events in the game.

4) Low-Level Design

4.1) Object Design Trade-Offs

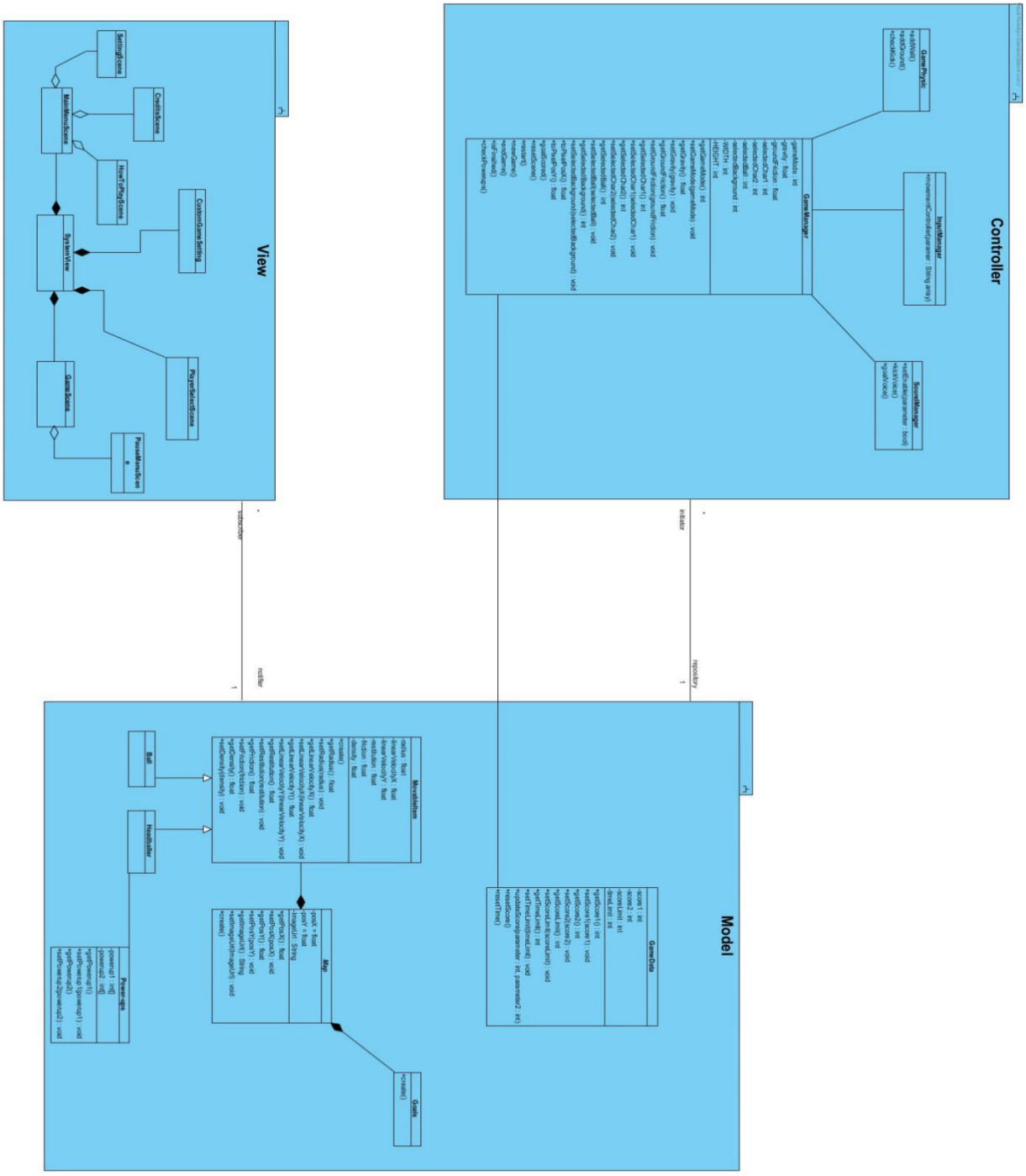
4.1.1)Functional Decomposition vs code length

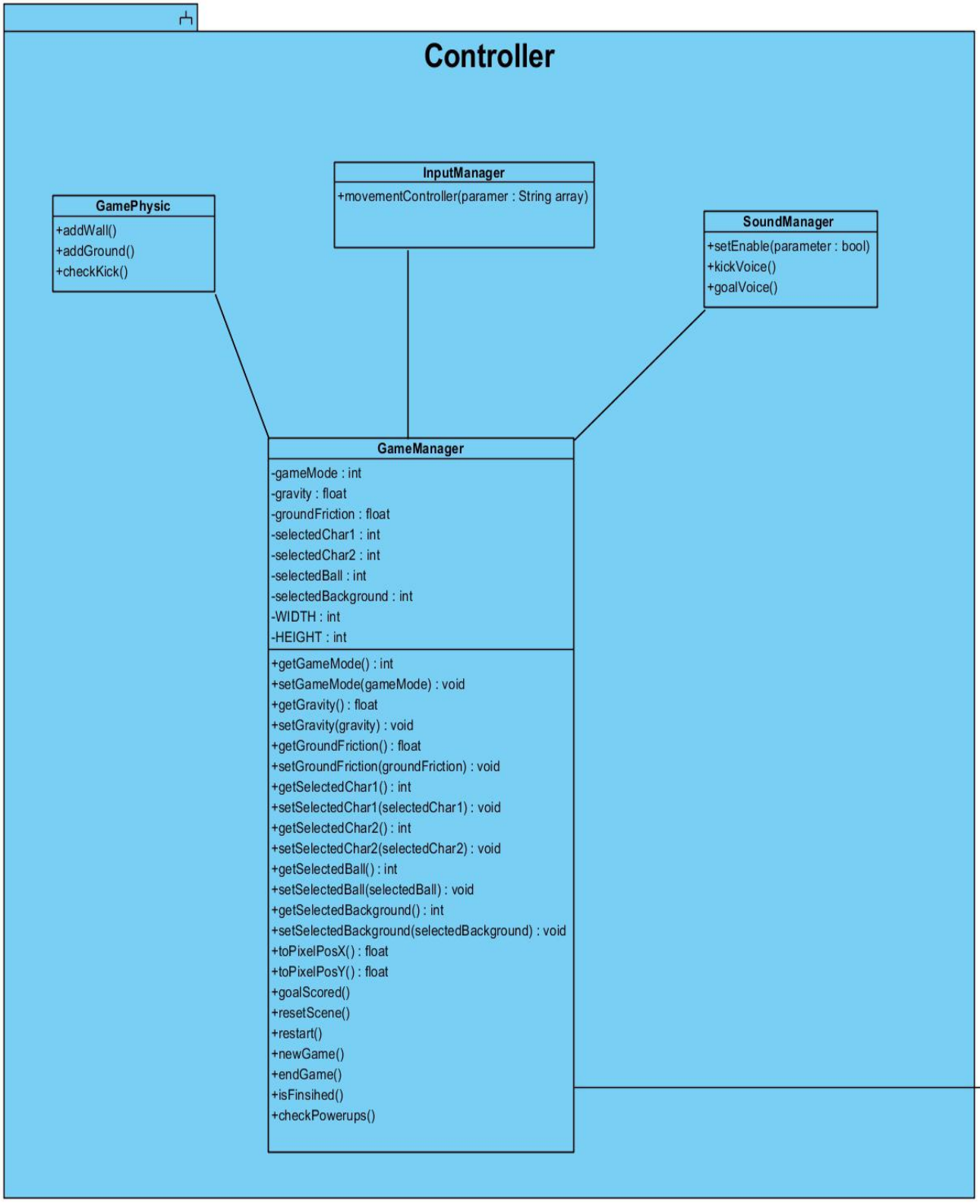
In our design report, we have aimed at dividing the code into individual modules. By doing so, maintenance and updation of the code can be done in a less strenuous manner and in a more efficient way, but at the cost of increased volume of lines of code.

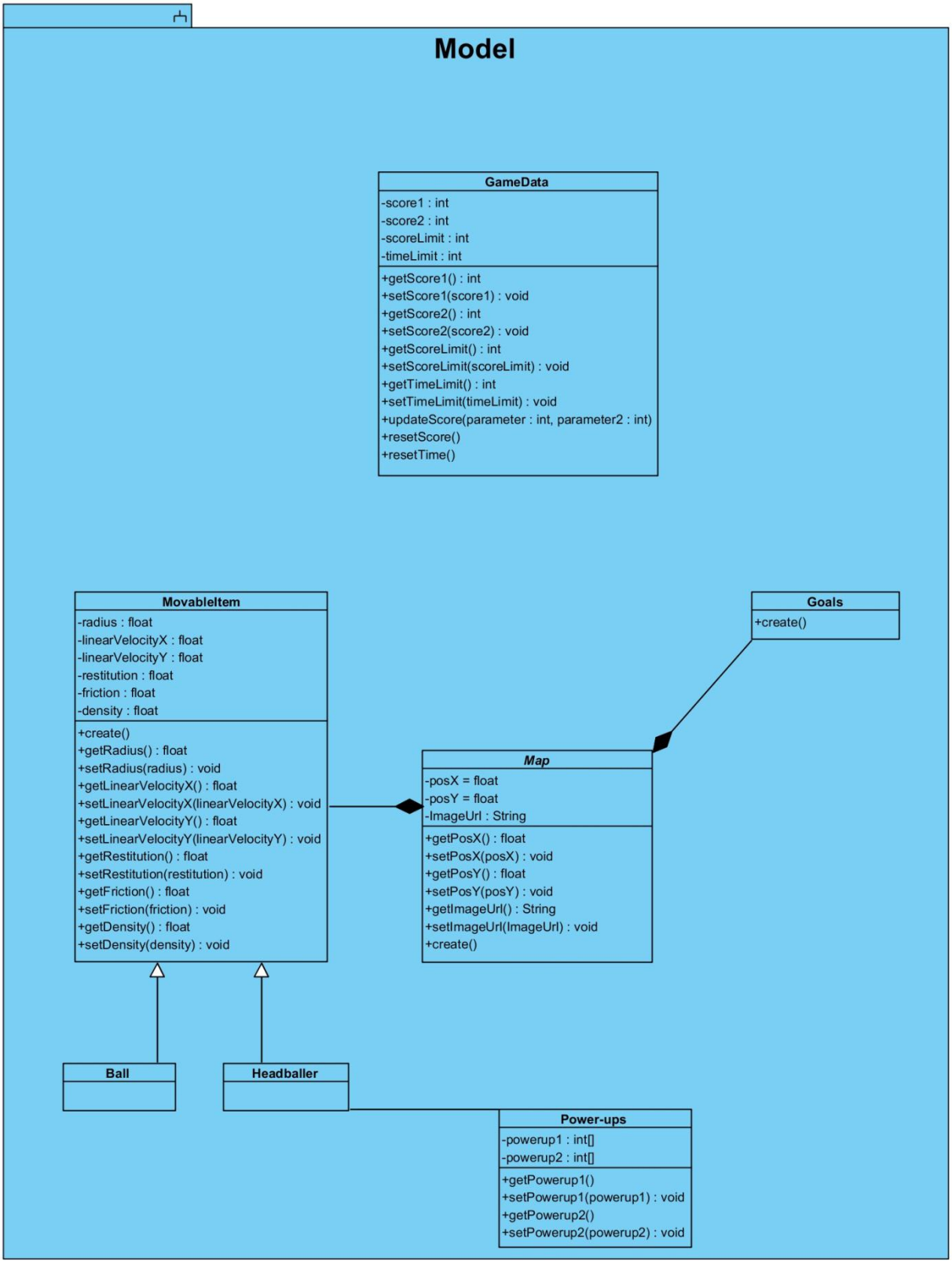
4.1.2)Performance vs complexity

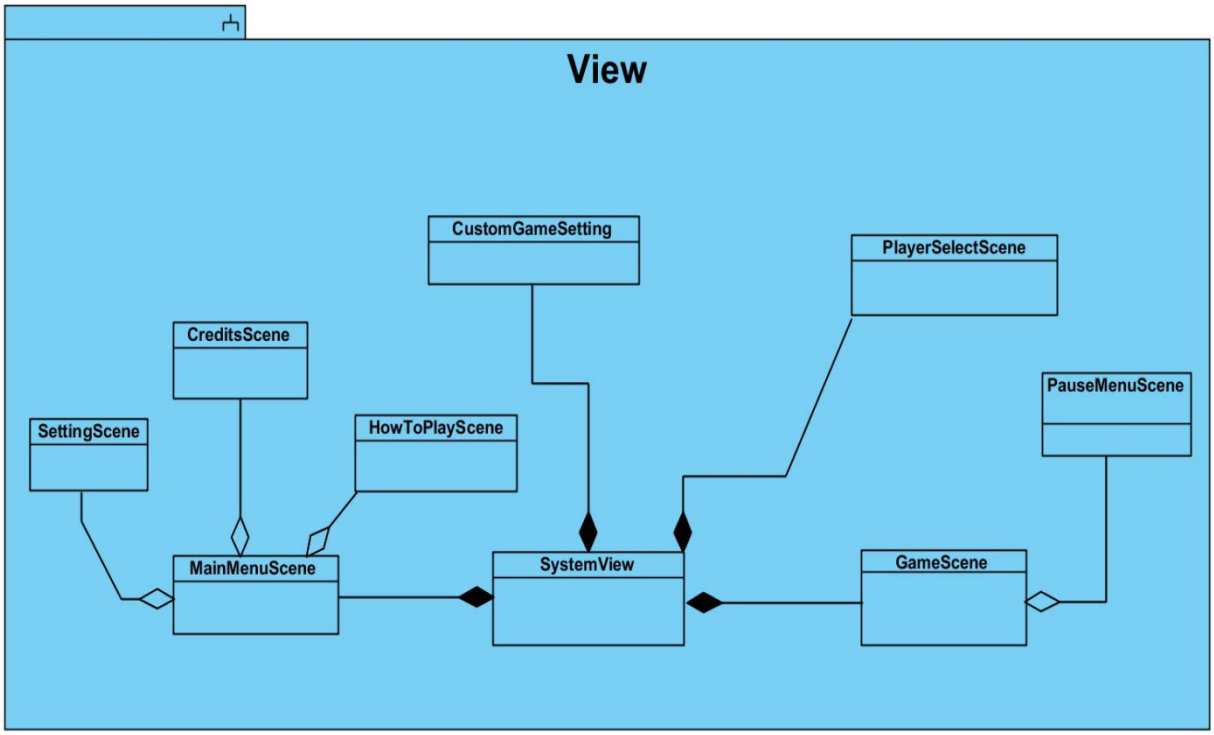
We have implemented the project by using MVC architecture. By doing so we have separated the game into three components in which we can seperate the Graphical User Interface (View) from database (Model) and control (controller) unit, where we can implement a more object oriented focused program and run the game in an efficient way. Though the adoption of this architecture the complexity of the game design would decrease which would ultimately ease the implementation of the game, as a complex design results in a complex and hard-to-do implementation.

4.2) Final Object Design





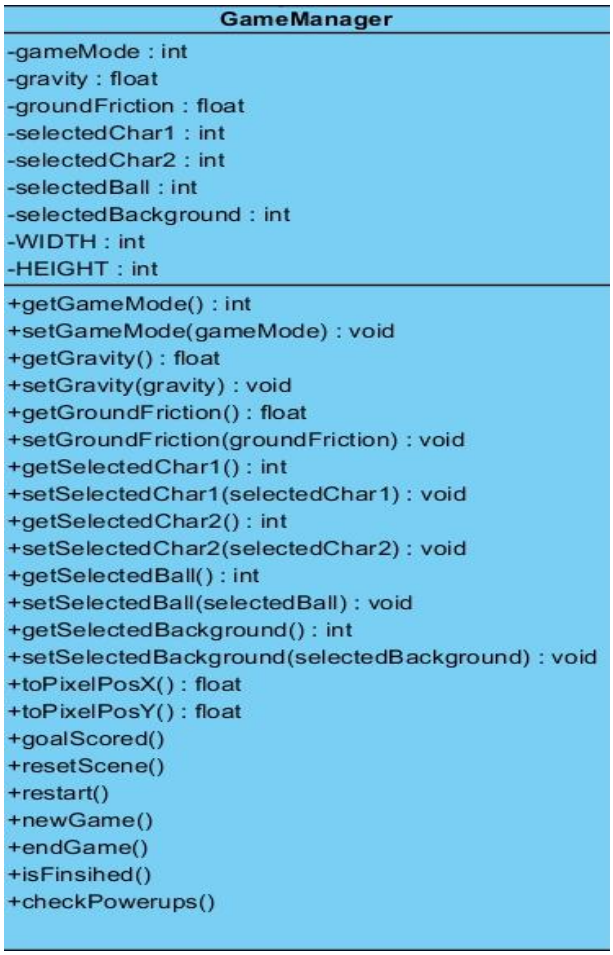

Model
GameData
-score1 : int
-score2 : int
-scoreLimit : int
-timeLimit : int
+getScore1() : int
+setScore1(score1) : void
+getScore2() : int
+setScore2(score2) : void
+getScoreLimit() : int
+setScoreLimit(scoreLimit) : void
+getTimeLimit() : int
+setTimeLimit(timeLimit) : void
+updateScore(parameter : int, parameter2 : int)
+resetScore()
+resetTime()
MovableItem
-radius : float
-linearVelocityX : float
-linearVelocityY : float
-restitution : float
-friction : float
-density : float
+create()
+getRadius() : float
+setRadius(radius) : void
+getLinearVelocityX() : float
+setLinearVelocityX(linearVelocityX) : void
+getLinearVelocityY() : float
+setLinearVelocityY(linearVelocityY) : void
+getRestitution() : float
+setRestitution(restitution) : void
+getFriction() : float
+setFriction(friction) : void
+getDensity() : float
+setDensity(density) : void
Goals
+create()
Map
-posX = float
-posY = float
-ImageUrl : String
+getPosX() : float
+setPosX(posX) : void
+getPosY() : float
+setPosY(posY) : void
+getImageUrl() : String
+setImageUrl(ImageUrl) : void
+create()
Ball
Headballer
Power-ups
-powerup1 : int[]
-powerup2 : int[]
+getPowerup1()
+setPowerup1(powerup1) : void
+getPowerup2()
+setPowerup2(powerup2) : void



4.3) Class Descriptions

4.3.1) Controller Package

4.3.1.1) GameManager Class



This class handles almost everything about the game. It provides connections between game data and display. It allows program to sets physical world behind the vision and to gives values to physical variable which affects how physical objects behave. It organizes and conducts main process of the game such as ending and starting game. In these methods all variables and objects are set to appropriate values and states according to process.

Attributes:

gameMode: This variable gets value from View according which mode player prefers to play.

gravity: This variable holds value of gravity of the world which is defined through the jbox2D library. During game this variable may get different values in different game modes. It needs to be set to different values and got by different methods.

groundFriction: This variable holds value of friction of the ground. During game this variable may get different values in different game modes. For example, with ice ground, its value would be lower than default friction. It needs to be set to different values and got by different methods.

selected … : These whole variables starting with selected get different values according to player preferences. When ball or headballer selection change, it gets different values and implies different attributes.

WIDTH and HEIGHT: They are both final values which can not be changed. They are set by programmer to default values according to screen size. Both world and game scene would be affected by values of these variables.

Operations:

toPixelPosX(): This method provides an algorithm which convert javafx pixel coordinates to world coordinates which is defined through jbox2d library utilized for visualizing physical motion and collision more accurately.

toPixelPosY(): This method provides an algorithm which convert javafx pixel coordinates to world coordinates which is defined through jbox2d library utilized for visualizing physical motion and collision more accurately.

goalScored(): That method checked whether goal is occurred for one of the goals. Then realize necessary operations after goal is detected. It updates score, calls resetScene() method and necessary voice methods.

resetScene(): It resets the scene appropriately. It re- locates the game object to their default places.

restart(): It is called when restart button is pressed. It re-initializes the game and game objects. It resets default values of physical attributions of the game object. It reset the game data such as score and time.

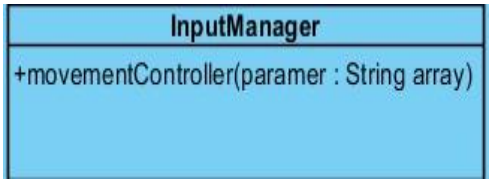
newGame(): It creates new world through the jbox2D library. And it puts two headballers and ball with preferred physical and visual attribution according to gameMode.

endGame(): It is called when game is over. It destroys the world which is created initially game object placed in. And other values set to default to prevent some confliction which may be arisen when user wants to play new game in different game mode.

isFinished(): This method checks whether game is over or not. It looks both time and score limit by getting current values of them from GameData class in the Modal. And it call endgame() method according to situation.

checkPowerups(): It checks which power-ups are activated. According to activated power ups, it changes and updates values of physical attributions of the game objects including ball and headballers.

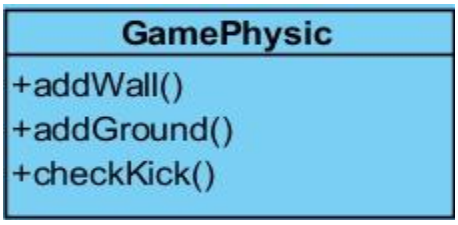
4.3.1.2) InputManager Class



Operation

movementController: It gets keyboard inputs from the listeners of the game scene and it realize appropriate movements and calls suitable method in GameManager to change and update physical attributions of the objects when it is needed.

4.3.1.3) GamePhysic Class



This class would be needed when world is being built and to detect occurrences of some of physical situations to call appropriate methods.

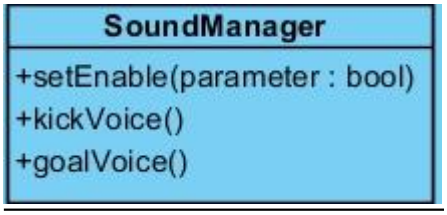
Operations

addWall() : This method is utilized when new game and world is built. It adds wall with given position, width and height to jbox2d world

addGround(): This method is utilized when world is built. It adds ground with given friction and height.

checkKick(): It checks whether there is collision or contact between one of headballers and ball and calls SoundManager methods.

4.3.1.4) SoundManager Class



Operations

setEnable(): It get value from the GameManager according to option value of sound preferred by the user and it enables or deactivates sound during the game.

goalVoice() and kickVoice() : They may both get activated(called) by GameManager to sound appropriate voice.

4.3.2) View Package

Here, methods are in FXML to control the GUI. So, instead of classes, we are demonstrating which class has which FXML files and show what they do.

4.3.2.1) Main_Menu_Controller Class

Operations:

`@FXML private void openPlayerSelect (ActionEvent event):` Changes the scene to player select menu.
`@FXML private void openOptions (ActionEvent event):` Changes the scene to options menu.
`@FXML private void openCredits (ActionEvent event):` Changes the scene to credits menu.
`@FXML private void openHowToPlay (ActionEvent event):` Changes the scene to how to play menu.
`@FXML private void exit (ActionEvent event):` Exits the program.

4.3.2.2) PlayerSelect_Controller Class

Attributes:

`@FXML private ToggleGroup playerOne:` The radio button group for the first player.

`@FXML private ToggleGroup playerTwo:` The radio button group for the second player.

`@FXML private ToggleGroup gameMode:` The radio button group for the game modes.

Operations:

`@FXML private void Start(ActionEvent event):` Starts the game by sending the player and game modes selections to game manager and changing the scene.

`@FXML private void goBack(ActionEvent event):` Changes the scene to main menu.

4.3.2.3) Options_Controller Class

Operations:

`@FXML private void goBack(ActionEvent event):` Changes the scene to main menu.

4.3.2.4) HowToPlay_Controller Class

Operations:

`@FXML private void goBack(ActionEvent event):` Changes the scene to main menu.

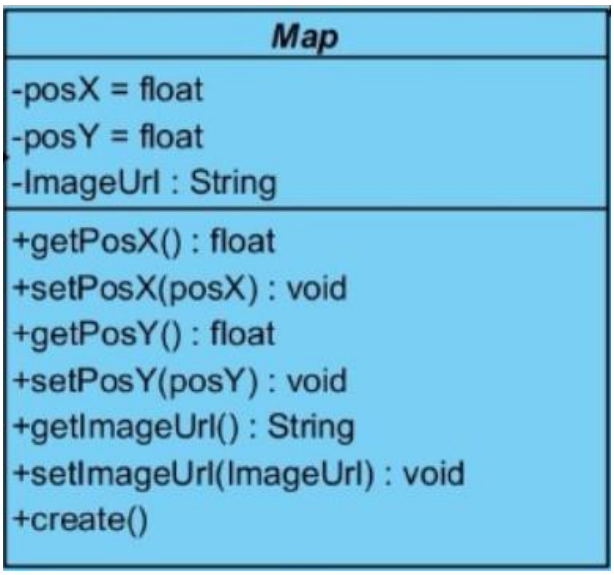
4.3.2.5) Credits_Controller Class

Operations:

`@FXML private void goBack(ActionEvent event):` Changes the scene to main menu.

4.3.3) Modal Package

4.3.3.1) Map Class



Map class is an abstract class which handles map data for Goals and MovableItem classes.

Variables:

Posx : position data of an object on X axis is kept here.

Posy : position data of an object on Y axis is kept here.

I**mageUrl :** url links of images of objects are kept here.

Methods:

getPosX() : gets the object's position on X axis.

getPosY() : gets the object's position on Y axis.

setPosX(posX) : void : updates the position of an object on X axis by getting the posX value.

setPosY(posY): void : updates the position of an object on X axis by getting the posX value.

getImageUrl() : gets the proper url of an image.

create(): abstract method for building each object to the scene.

4.3.3.2) Ball Class

Ball class extends MovableItem, methods are same but the variables are different for each ball.

4.3.3.3) Headballer Class

Headballer class extends MovableItem, methods are same, like ball but headballers have different variablers and attributes.

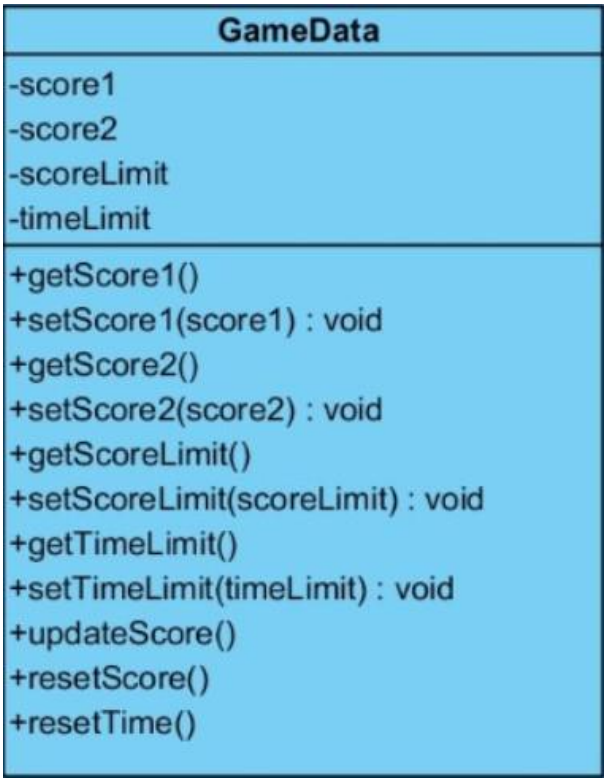
4.3.4.4) Goals Class

Goals class is for creating goals on the game scene. They are invisible objects on the map and are fixed to their locations. This enables us to use the top pole of the game efficiently .

Methods:

create(): This is the method for creating goals on the scene.

4.3.4.5) GameData Class



Variables:

score1 : Score of the first player

score2 : Score of the second player

scoreLimit : Highest achievable score by a player is kept here.

timeLimit : Duration limit of the game is kept here.

Methods:

getScore1(): int : returns the score of the first player

setScore1(score1):void : Sets the score value to the score of the first player.

getScore2(): int : returns the score of the second player

setScore2(score2):void : Sets the score value to the score of the second player.

getScoreLimit():int : Returns the highest score possible.

setScoreLimit(scoreLimit):void : Sets the value of scoreLimit.

getTimeLimit():int : Returns the duration limit of the game.

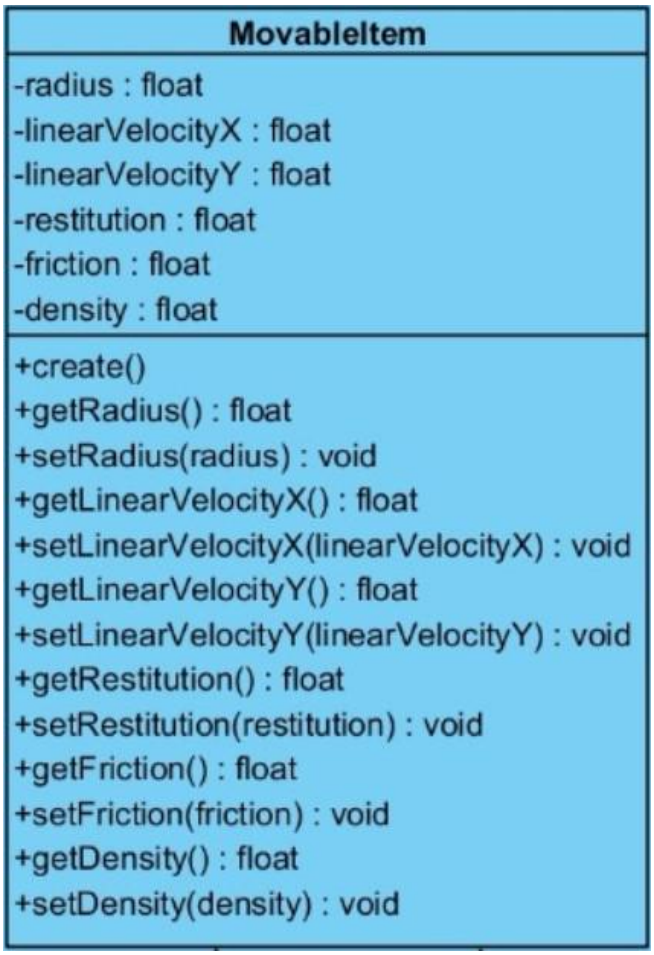
setTimeLimit(timeLimit):void : Sets the timelimit before starting the game.

updateScore(parameter:int,parameter2:int) : Changes the scoreboard whenever one of the players score.

resetScore() : Resets the scores.

resetTime() : Resets the time.

4.3.4.6) MovableItem Class



Variables:

Radius:float : Since we define the head of the players and ball as circle, we keep radius of the circle.

linearVelocityX:float : Velocity in the horizontal direction

linearVelocityY:float : Velocity in the vertical direction.

restitution:float : This is the restititution of objects, which affect the bounciness (i.e. ball)

friction:float : friction of objects in the scene is kept here.

density:float : density is an important coefficient about jumping calculations and kept here.

Methods:

create() : Creates the objects when the method is invoked.

getRadius():float : returns the radius value for the circles.

setRadius(radius):void : Sets the radius value to the taken parameter.

getLinearVelocityX():float : returns the velocity in the horizontal direction.

setLinearVelocityX(linearVelocityX):void : Sets the horizontal velocity value to taken parameter.

getLinearVelocityY():float : Returns the velocity value in the vertical direction.

setLinearVelocityY(linearVelocityY):void : Sets the vertical velocity to taken parameter.

getRestitution():float : returns the restitution of an object.

setRestitution(restitution):void : sets the restitution of an object.

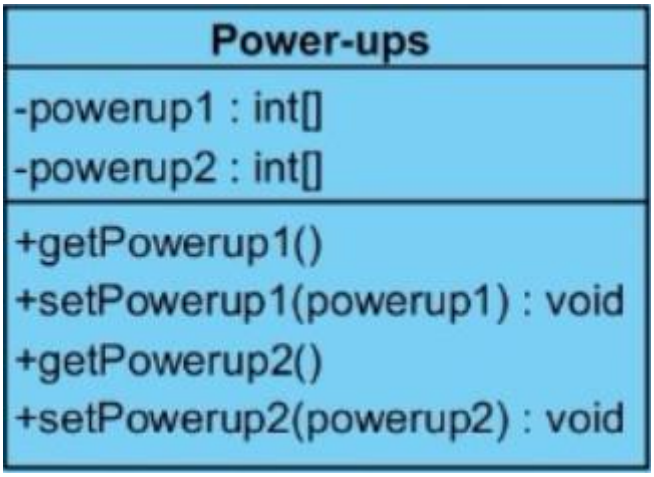
getFriction() : Returns the friction value.

setFriction(friction):void : Sets the friction value to taken parameter.

getDensity() : Returns the density value

setDensity(density):void : Sets the density value to taken parameter

4.3.4.7) Power-ups Class



Variables:

Powerup1: int[] : it is a two dimentional array for player 1 which powerups are available.

Powerup2: int[] : it is a two dimentional array for player 2 which powerups are available.

Methods:

getPowerup1() : returns the powerup statusfor player 1.

setPowerup1(powerup1):void : sets the status of player1's powerups.

getPowerup2() : returns the powerup statusfor player 2.

setPowerup2(powerup2):void : sets the status of player2's powerups.