Bilkent Univeristy

Computer Engineering

CS319 – Section 2

Object Oriented Software Engineering

Section TA: Gulden Olgun

Supervisor: Bora Güngören

Design Report: Final Draft

OCTOBER 7TH, 2017

Bubble Popper A Project By Group 2-I

Serhat Hakkı Akdag

Orkun Alpar

Mustafa Mert Aşkaroğlu

Faaiz Ul Haque

Table of Contents

Table of Contents

Design Report

1. Introduction

1.1 Purpose of the system

Bubble shooter is a 2D arcade game that aims to allow users to have an enjoyable experience. The game consists of a very friendly user interface with convenient navigation and basic options. The menus will allow users to learn about the game, find information of the developers, adjust settings, view high scores and play the game. The game will have simple graphics to ensure simplicity and allow a smooth gameplay. Users can have a competitive experience while cooperating with each other and the system using the tools used to play the game. Players can develop skills such as communication, human reaction, and hand-eye coordination. Additionally players must use their game-skills which they will achieve over-time in order to complete the sequential levels as they rise in difficulty. The primary aim is to allow a stress-relieving and calm environment in which users can enjoy in their free time.

1.2 Design Goals

Design goals are an important criteria to establish before the actual implementation of our system. We can derive most of our design goals using our observations and statements we made in our analysis report of the game. It is important to have design goals as advanced as possible in order to achieve at least a decent final product. Most of the design goals can be linked with the nonfunctional and functional-requirements of the game. The design stage is important in creating an application that will allow optimal user experience.

Efficiency: By keeping a balance with the graphics we can obtain an efficient program although this may result in a poor looking game. However our main objective is to provide a smooth running game with minimal glitches and lags to provide a better user experience rather than having a fancy looking game.

Usability: By having a friendly-user interface with basic commands and an easy navigation system the usability of the application will be as convenient as possible. Users only require a mouse and a few keys on the keyboard to operate the game. Users can easily switch menus independent of which menu they are on.

For example, the user can pause the game and return to the main menu.

Adaptability and Portability: Since our game will be entirely run on Java Virtual Machine which provides cross-platform portability meaning the users do not have to be concerned regarding the operating system requirements as the game can function and be worked on in all JRE platforms. Although by using Java for both implementation of the games mechanics and graphics, we are eliminating the possibility of having benefits from other programming languages and other soft wares such as Python, Direct3d and C++.

Learnability: The game has a menu with very brief and basic instructions in simple English. Furthermore there will be diagrams to allow for aid in understanding. Users can easily grasp the concept of the game by even trying out a few levels and easily restarting the game. The controls are easy to learn as there are only a few keys used for each player that are near each other and simple to memorize.

Reliability: As mentioned in efficiency the game will focus on providing a smooth performance with low graphics. This is to ensure the system does not face issues such as lags or crashes. Furthermore, any type of glitches will be avoided at all costs by testing each of the different cases that may occur in a game. This will ensure a reliable game and have fair competition constantly when the game is played.

Extensibility and Modifiability: Since the game is all done in Java we will be using object oriented programming and make extensive use of features such as inheritance and extendibility. In order to make this process easier certain classes will have as less connections with each other as possible. The goal of the implementation should be to have main manager that uses many classes in one. Having this as a design goal is crucial in being too able to make the implementation procedure easier. Furthermore when the game is officially launched, any errors or future improvements could arise by users. With extensibility and modifiability as properties, the developers of the game can easily alter the game and add these updates with ease. Furthermore they can change the existing system to resolve issues that arise late in game-play.

1.3 Design Trade Offs

Reliability VS Graphics: In order to have a smooth running game with the reliability that it won't crash and run into bugs we have decided to create a game with relatively low graphics and without the use of high detailed imaging. This way game will be able to run on computers with low specifications, low memory, etc. But some of the fancier graphics that might be added will not be considered to achieve the reliability.

Usability VS Functionality: The functionality of the game will be quite simple with only a few keys required to operate and play the entire game. The instructions are clear and brief and the game is simple to learn and play. Users can navigate through the entire game using just their mouse. However, by making easy to use game some of the possible extra functionalities will not be added. This will ensure that players will not get confused by a lot of functionalities and instructions to use them.

2. Software Architecture

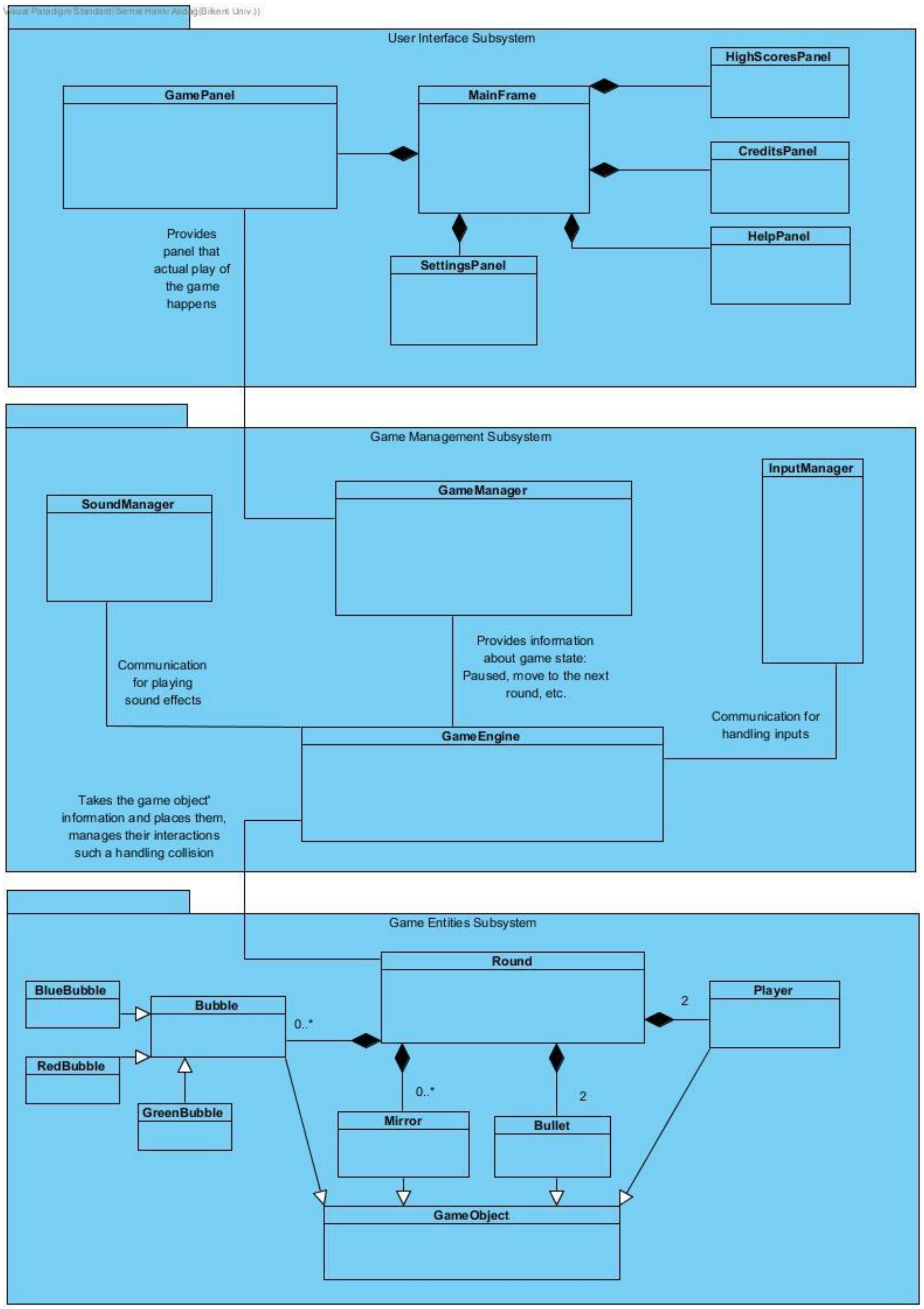
2.1 Overview

In this section we will provide a detailed summary with the aid of diagrams to explain the internal structures of our system. We decided to have a 3 sub-system type of organization, to reflect the idea of the MVC model (Model-View-Controller). This allows an abstract overview of our system and makes the implementing procedure easier. Us, as developers can understand how the internal features of the game will operate during gameplay. It will also allow users to grasp the game creation easier. Additionally, modifying the game when new versions will be created in the future, will be more convenient.

2.2 Subsystem Decomposition

The subsystem decomposition of the game will show how class interaction and its methods and elements operate in our project. The system is divided into three parts that are connected in specific ways. Each subsystem contains classes that independently work on a tasks cooperatively. However the subsystems are also joined and send necessary data to and from each other. This architecture type was chosen to ensure low coupling and high cohesion in our system which means if one error occurs in a certain part it won’t have much effect on other parts of the system, but still the code will be bonded together as much as possible

The organization of the system is clearly shown in the figure below.


Visual Paradigm Standard(Serhat Hakki Akdag(Bilkent Univ.))
User Interface Subsystem
GamePanel
MainFrame
HighScoresPanel
CreditsPanel
HelpPanel
SettingsPanel
Provides panel that actual play of the game happens
Game Management Subsystem
SoundManager
GameManager
InputManager
Communication for playing sound effects
Provides information about game state: Paused, move to the next round, etc.
Communication for handling inputs
GameEngine
Takes the game object' information and places them, manages their interactions such a handling collision
Game Entities Subsystem
Round
BlueBubble
RedBubble
GreenBubble
Bubble
0..*
Player
2
Mirror
0..*
Bullet
2
GameObject

Basic Subsystem Decomposition

2.2 Subsystem Decomposition (continued)

In the figure above we can clearly see the three layered model, in which we have the User-Interface at the top (View), the Game Management in the middle (Controller) and the Game Entities at the bottom (Model). When the game is executed the user is prompted to the menu which is why the user interface is the initial point of our system. From here the user can navigate through all possible menus in the game such as credits, highscores, settings, and playing the game. The mainframe is a class which will provide the layout for the main menu, which in turn contains all of the sub-menus. The GamePanel in the user interface subsystem is connected to the management subsystem. This is where the actual mechanics of the game are managed. It handles everything such as player movement, bubble collisions, sound management, game status and all other inputs and outputs that occur in the game. The GameEngine is the head class of this sub-system and it is also connected to the third sub-system, the Game Entities. The GameEngine collects all the necessary information required to manage the game from the Game entities subsystem. Such as the amount of lives, score, player location, bubble types and locations, etc.

2.3 Hardware/Software Mapping

This desktop game will require both a single mouse and keyboard to be played by two players. The game will be implemented entirely in a Java environment. The graphs will also be created using java's graphical user interface classes. We will be able to use Java's GPU acceleration during this process. Since the graphics will be basic and not high definition, a simple computer with a regular graphic card is sufficient for being able to run the game. An internet connection is not required since everything is processed offline.

2.4 Persistent Data Management

All game data related to our system will be stored on the hard-disk. Since the game is entirely run offline, we do not require an online server as a storage point. The overall storage required is low so we do not need a complex database. The map, sound effects, credits are all constants in the game and do not need to be altered.

The game map will not be changing as the background is stationary and such features will only be processed during run-time when the game is executed. We only need to update the highscore system that can be saved and loaded again once the game is run for the second time. We will use text files to store such data like high scores, lives, current in-game scores, etc. So our system should support such data types.

2.5. Access Control and Security

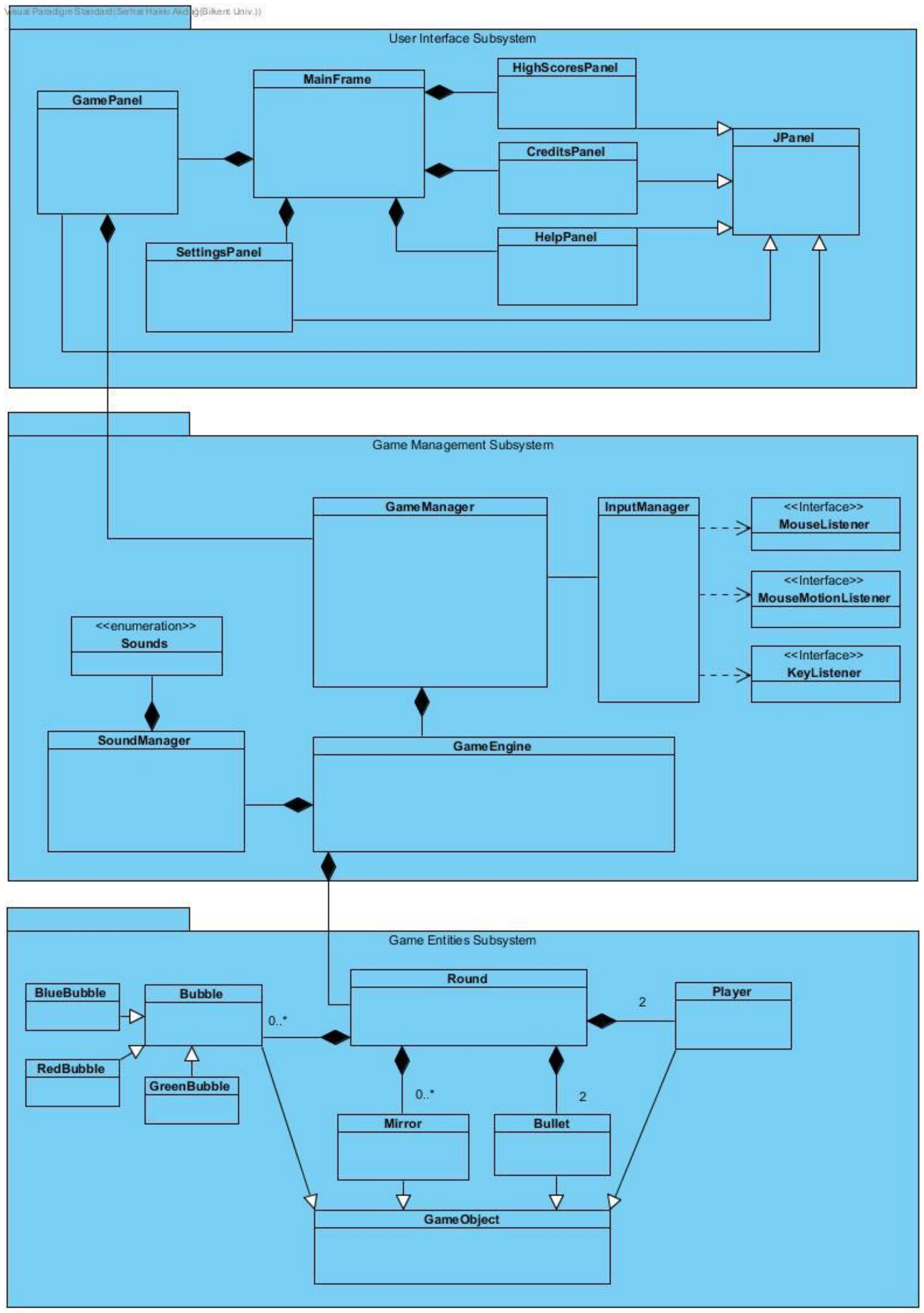
Our game has no issues with security, as the users the wish to play do not need to provide any personal data. There is no user authentication required, so no account is required for the game, any guest users can play as they please. Users may save a codename or their personal names if they wish upon reaching the high score list. The game can easily be accessed by anyone with a functional operating system. This enables a simpler implementation and keeps the game simple and easy encouraging more players.

2.6. Boundary Conditions

No installation process is required and the game will be executed on a simple .jar file. If a user re-opens the game when already launched once, the current process will terminate. If a file such as a text-file used to store data corrupts during or before game-play, the game will terminate and reset all its information. If a user clicks the ‘Quit’ option from the main-menu or clicks the ‘X’ button in the top right corner of the file the game will terminate. If a user clicks the ‘Play’ option from the main-menu the game will start, and the Controller sub-system will begin to operate. If a user is hit by a bubble, the round will reset or if the two players were on the last life, the game will be re-prompted to the main menu and the high-scores, if applicable, will be updated. If a user pops all bubbles the round will reset or if the two players were on the last round, the game will be redirected to the main-menu since the users have completed the game and again, if applicable, the high-scores will be updated. If the users press ‘P’ during game-play the game will pause, and if the users choose to they can return to the main-menu ending the current running process of the game, restarting from level one, with all lives and scores reset, the next time the ‘Play’ button is pressed.

3. Subsystem Services

3.1 Detailed System Design

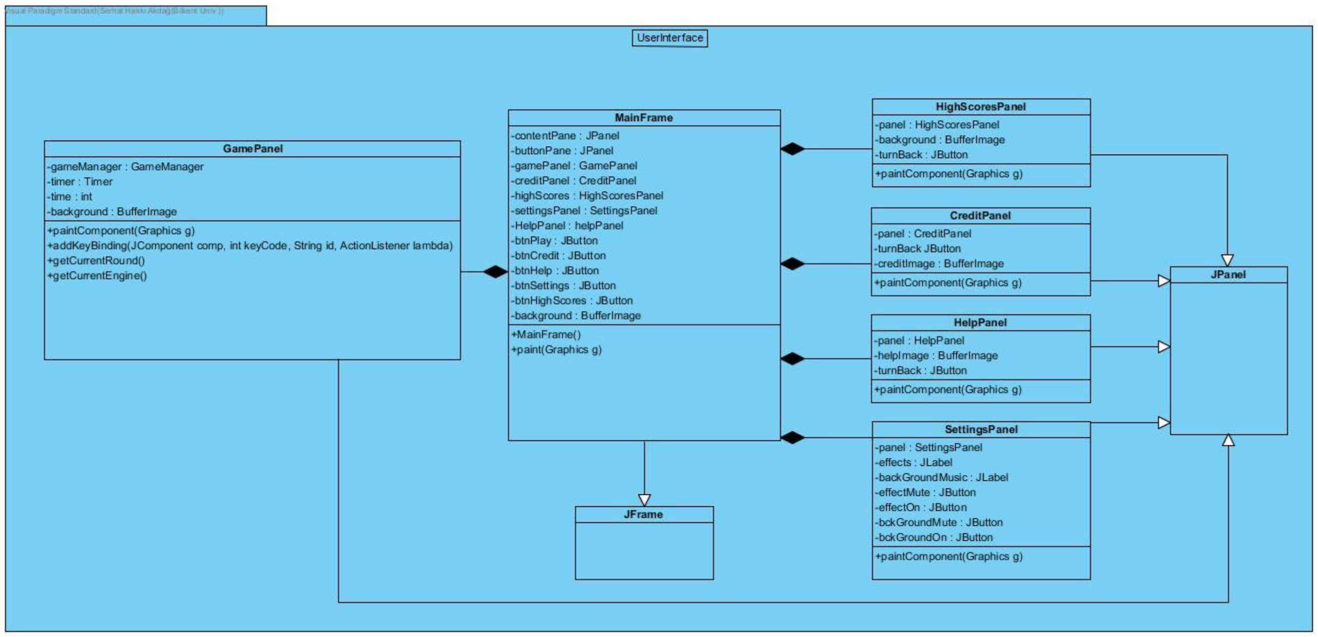


This diagram provides the detailed system containing all three parts in order to understand how the sub-systems are created. Below are each of the three sub-systems explained in further detail with their class functions and variables.

3.2 User Interface Subsystem

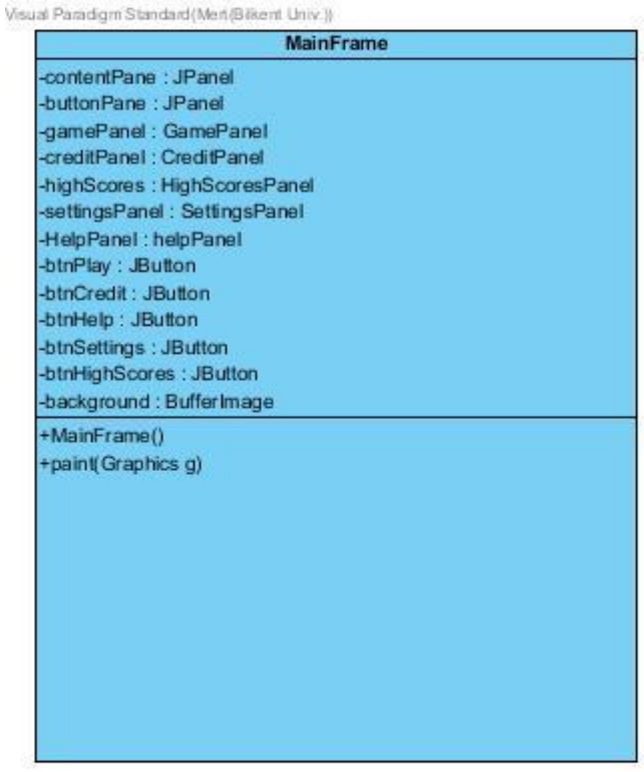
The user interface system shows the visualization of the program. The simple and convenient layout will allow users to access different menus of the game as they please. The system includes independent classes which work on their own specific tasks while being correlated to each other with certain connections. Depending on the user's inputs, different layouts of the interface will be prompted.

Below is the structure of the user interface subsystem and a brief explanation of how each class operates.



MainFrame Class

Main Frame is a class which is the only frame of our game. It includes buttons which have action performed method to arrange for switching panels according to useinput.



Attributes

private contentPane: JPanel Contains a panel which is activated by user in it.

private buttonPane: JPanel Contains buttons which enable user to change the activated panel according to user's input.

private settingsPanel: SettingsPanel: Contains all information regarding the settings of the game such as changing sound. Main menu extends this panel and this panel has a constructor used to initialize the required settings.

private creditsPanel: CreditsPanel: Contains all information regarding the contact information of the developers. Main menu extends this panel and this panel has a constructor used to initialize the required information.

private helpPanel: HelpPanel: Contains all information regarding the instructions and controls of the game. Main menu extends this panel and this panel contains a constructor used to initialize the controls and instructions

private highScorePanel: highScores: Contains all information regarding players names and their top scores. Main menu extends this panel and this panel contains a constructor used to initialize the high scores.

private gamepanel: GamePanel: It is the panel which can be activated by click on btnPlay. It contains our game interface and key binding to enable users to move characters and pop the balls.

private btnPlay: **JButton:** It is the button which has action listener to activate gamePanel and deactivate current panel in main frame.

private btnCredit: **JButton:** It is the button which has action listener to activate creditPanel and deactivate current panel in main frame.

private btnHelp: **JButton:** It is the button which has action listener to activate helpPanel and deactivate current panel in main frame.

private btnSettings: **JButton:** It is the button which has action listener to activate settingsPanel and deactivate current panel in main frame.

private btnHighScores: **JButton:** It is the button which has action listener to activate highScores and deactivate current panel in main frame.

Methods

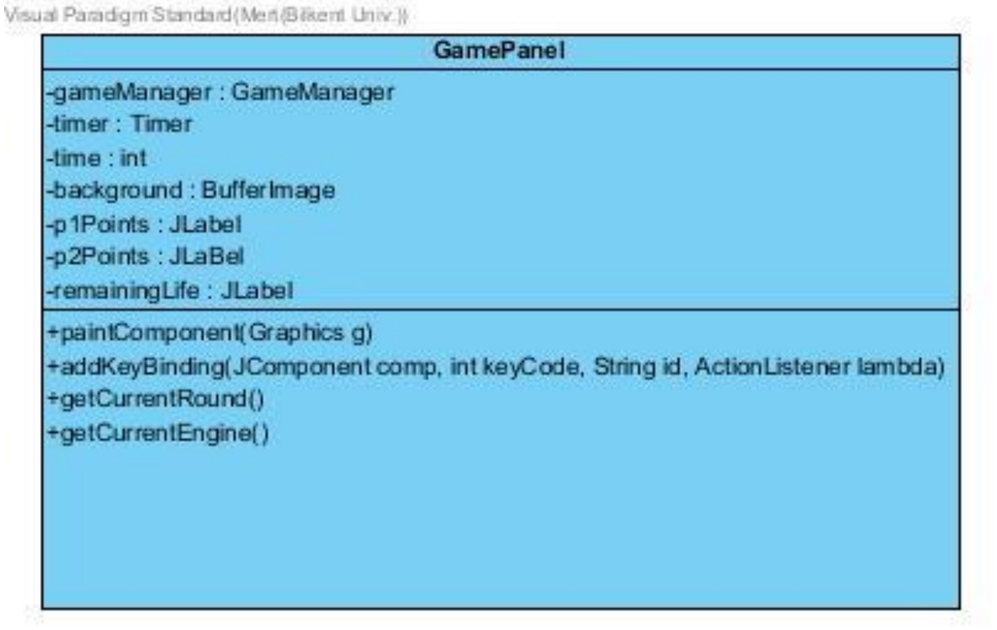
public MainFrame()

It is the constructor of MainFrame.

Public void paint(Graphics g)

It is doing required drawings on the frame

Game Panel Class


Visual Paradigm Standard(Mert(Bilkent Univ.))
GamePanel
-gameManager : GameManager
-timer : Timer
-time : int
-background : BufferImage
-p1Points : JLabel
-p2Points : JLaBel
-remainingLife : JLabel
+paintComponent(Graphics g)
+addKeyBinding(JComponent comp, int keyCode, String id, ActionListener lambda)
+getCurrentRound()
+getCurrentEngine()

Attributes

Private gameManger: GameManager: It is used to managing game features and occurrences which enable GamePanel to display game objects in correct way.

Private p1Points: **JLabel:** It is used to display the points of player one in the current game

Private p2PointsPanel: JLabel: It is used to display the points of player two in the current game

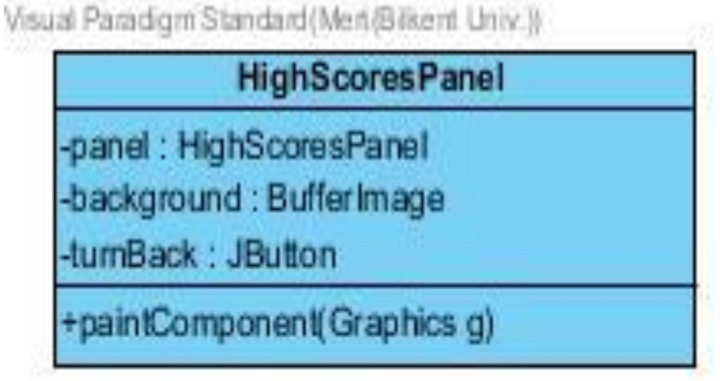
Private remaniningLife : JLabel: It isused to display the current lives shared amongst the two players in the current game

Private timer: Timer: It is used to do some executions depending upon time.

Methods

public void paintComponent(Graphics g): It is doing required drawings on the panel according to data which is come from game manager.

HighScorePanel

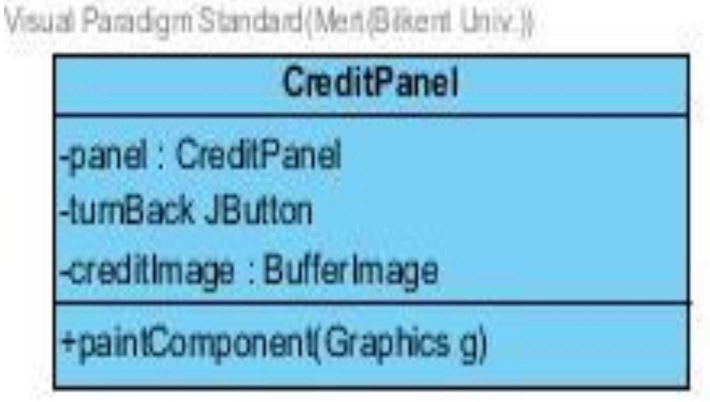

Visual Paradigm Standard(Mert(Bilkent Univ.))
HighScoresPanel
-panel : HighScoresPanel
-background : BufferImage
-turnBack : JButton
+paintComponent(Graphics g)

This class is used to keep high scores which is achieved by players

Attributes

Private turnback: **JButton:** It is used to turn back main menu.

CreditsPanel

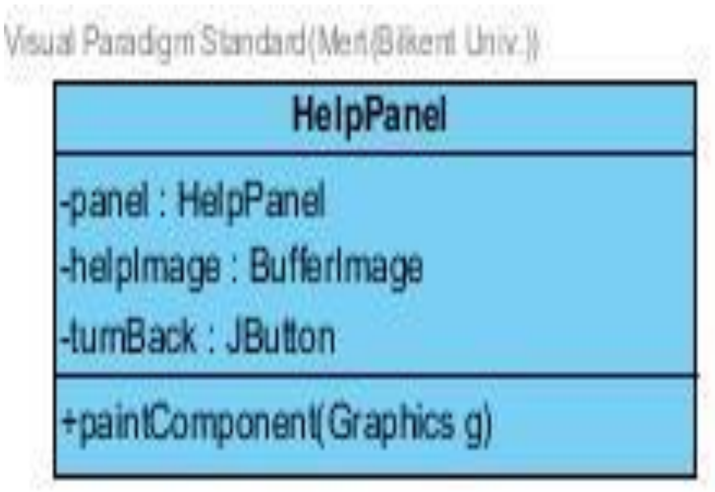

Visual Paradigm Standard(Mert(Bilkent Univ.))
CreditPanel
-panel : CreditPanel
-turnBack JButton
-creditImage : BufferImage
+paintComponent(Graphics g)

This class is used to keep credit image which is included information about authors

Attributes

Private turnback: **JButton:** It is used to turn back main menu

HelpPanel

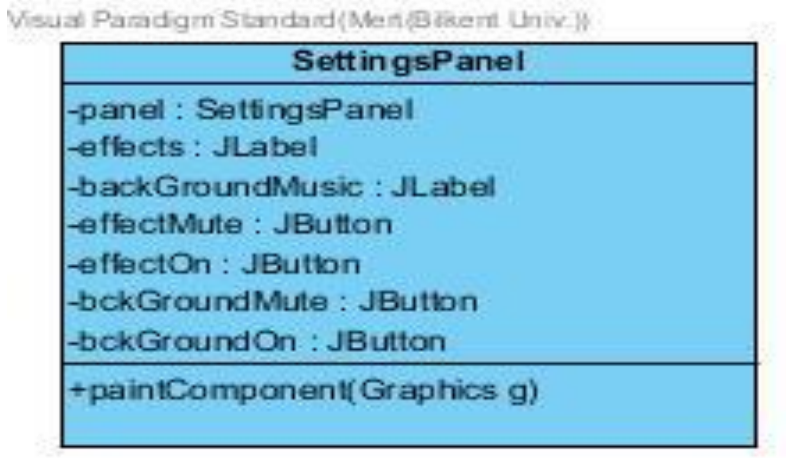

Visual Paradigm Standard(Mert(Bilkent Univ.))
HelpPanel
-panel : HelpPanel
-helpImage : BufferImage
-turnBack : JButton
+paintComponent(Graphics g)

This class is used to keep information which help user to learn how to play

Attributes

Private turnback: **JButton:** It is used to turn back main menu

SettingsPanel



Attributes

Private effectMute: **JButton:** It is used to make effect sounds mute, depend upon user wish.

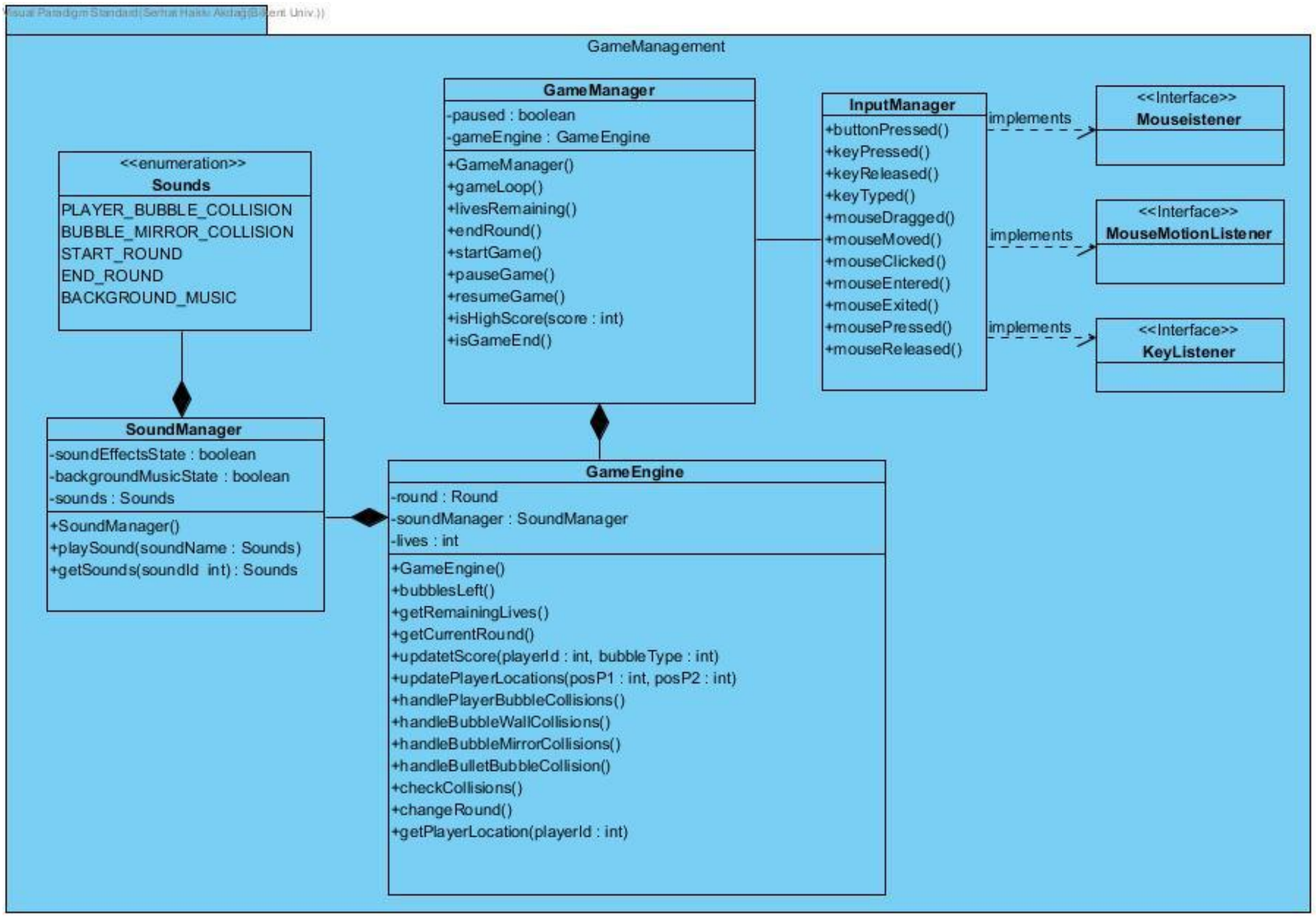
Private effectOn: **JButton:** It is used to provide game with effect sounds, depend upon user wish.

Private bckGroundOn: **JButton:** It is used to provide game with background sounds, depend upon user wish.

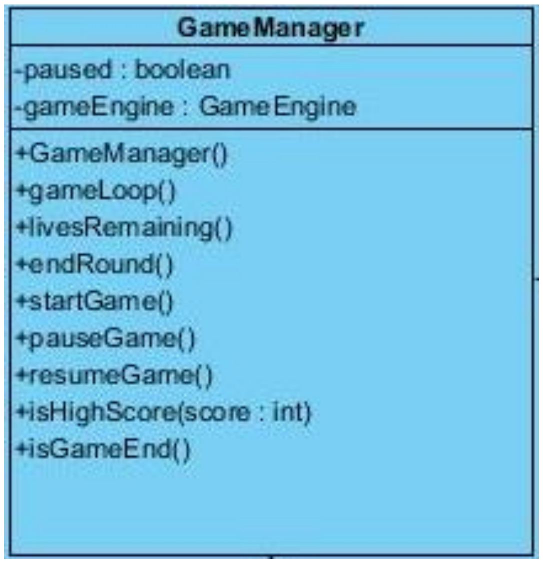
Private effectOn: **JButton:** It is used to make background sounds mute, depend upon user wish.

3.3 Game Management Subsystem

This is the subsystem that is responsible for managing game dynamics and logic. Namely, it manages data transfer between UI and game entities while checking and handling collisions, managing sound, updating player positions, etc. This subsystem includes 4 control classes as well as enumeration class for sound effects. The structure of this subsystem is:



GameManager Class:



Attributes: *private boolean paused ->* This attribute is used for holding true or false value for understanding whether game is paused or not

private gameEngine : GameEngine -> This attribute is a GameEngine object.

Constructors: *public GameManager() ->* Initializs the GameManager class at the first opening of the game.

Methods:
public void gameLoop() -> This method runs the loop that game is constantly updated

public Boolean endRound() -> This method checks whether certain round that is played ended by getting data from gameEngine class. Ending of the round is determined by checking if all the bubbles are popped and there are still lives left to continue the game.

public Boolean livesRemaining() -> This method controls whether players still have lives remaining to continue the game by getting appropriate information from Game Engine class.

public void startGame() -> This method resets all the data left from previous plays of the game such as lives, player points, etc. and gives a signal for starting of the new game.

public void pauseGame() -> This method prevents game loop to continue to do its work and makes game to wait on its current state. In addition, it creates a panel that informs players that game is paused.

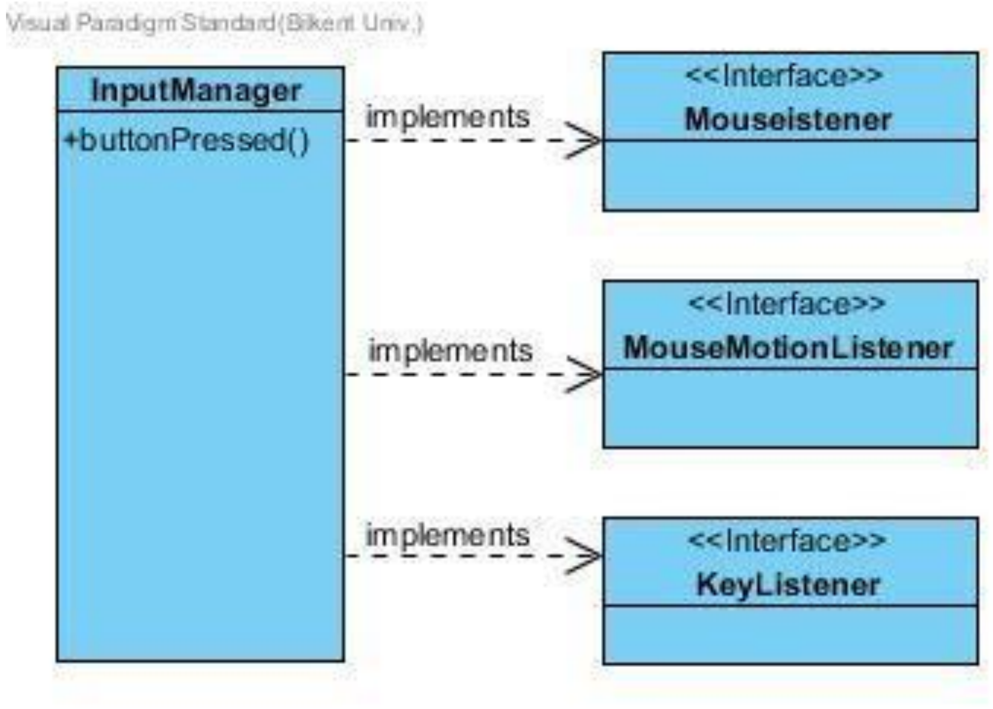
public void resumeGame() -> This method makes game loop to continue its work and makes the panel that informs about state of the game invisible.

public Boolean isHighScore(score :int) -> This method checks whether score made by last players of the game is eligible to be written to the top 10 high score list.

public void isGameEnd() -> This methods checks if the either all rounds ended or players have no lives left to play game.

InputManager Class:

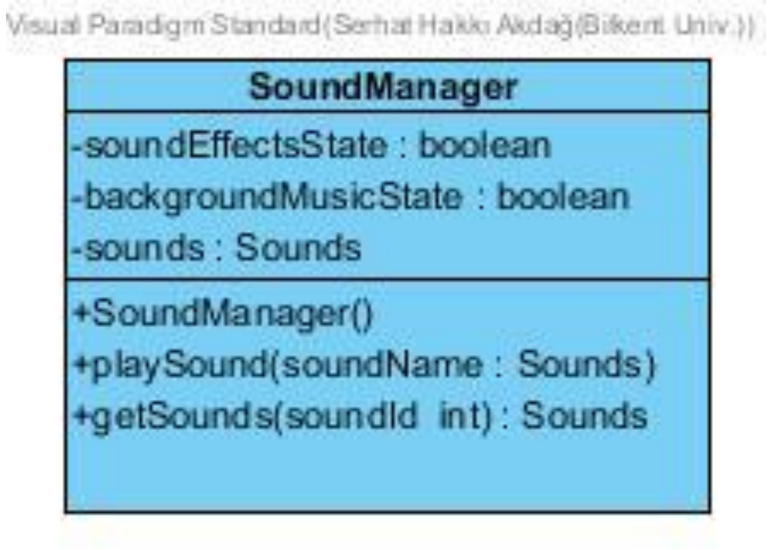
This class detects player actions from both mouse and keyboard. Players can navigate through main menu of the game by using mouse. Actual gameplay for both players is performed by keyboard. For each of the players 4 keys for movement and 1 for shooting will be used. In addition pausing and resuming the game will be done using the keyboard.



Methods:
public boolean buttonPressed() -> This method will be used for between round screen where program will wait for players for press any button. Method will return true if such action happens.

SoundManager Class:

This class will provide variety of sound effects and music. For example, when a bubble player collision happens, this class will take information from Game Engine class and it will play the corresponding sound effect.


Visual Paradigm Standard(Serhat Hakkı Akdağ(Bilkent Univ.))
SoundManager
-soundEffectsState : boolean
-backgroundMusicState : boolean
-sounds : Sounds
+SoundManager()
+playSound(soundName : Sounds)
+getSounds(soundId int) : Sounds

Attributes:

Sounds : Sounds -> This single property of SoundManager class is an instance of the enumerated sounds.

soundEffectsState : Boolean -> This attribute holds information for whether sound effects are closed by users from settings or not.

backgroundMusicState : Boolean -> This attribute holds information for whether background music is closed by users from settings or not.

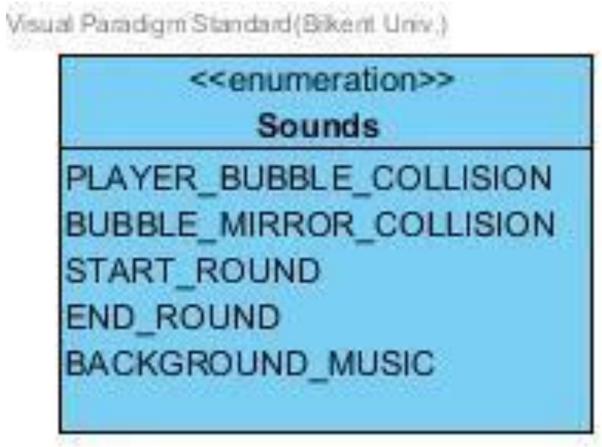
Constructors:

public SoundManager() -> This constructor initializes the object for this class and gives default values to its attributes.

Methods:

playSound(soundName : Sounds) -> This method is used to play desired sounds by desired times by getting information from Game Engine about collisions, their types, and round or game states. The sound that is played is chosen from the enumerations provided in Sounds class.

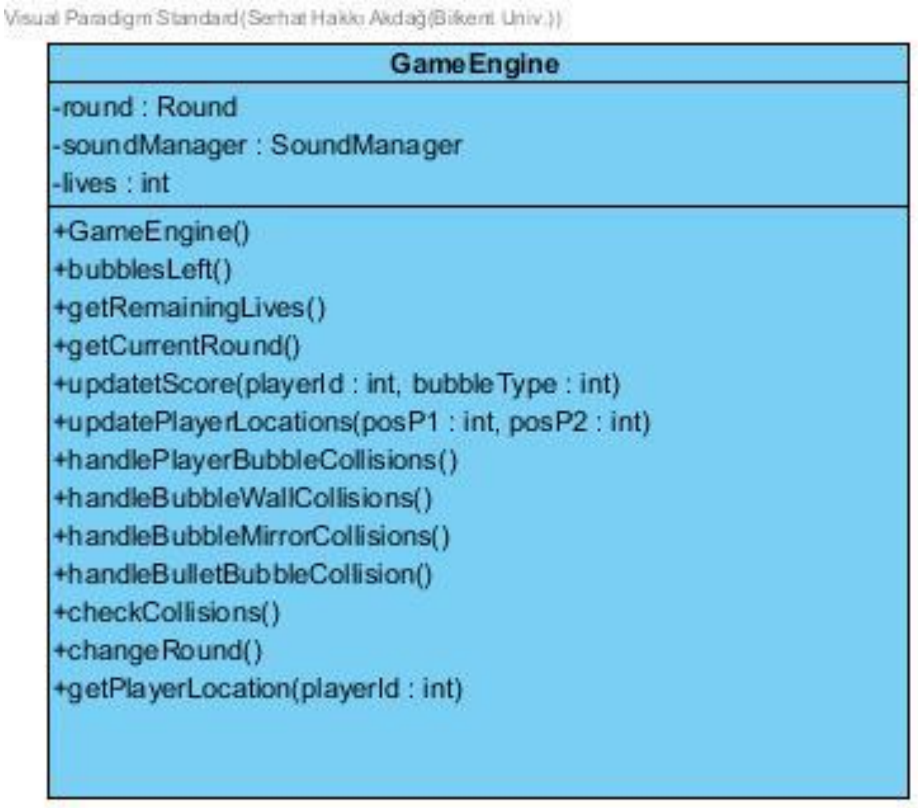
Sounds Class:
This enumeration will include various sounds effects and they will be enumerated to be used easily by soundManager class.



Sounds Class:
This enumeration will include various sounds effects and they will be enumerated to be used easily by soundManager class.

GameEngine class:

This class provides main structure for communication between users and game entities, actual game logic, and it is the backbone of actual playing of the game.


Visual Paradigm Standard(Serhat Hakkı Akdağ(Bilkent Univ.))
GameEngine
-round : Round
-soundManager : SoundManager
-lives : int
+GameEngine()
+bubblesLeft()
+getRemainingLives()
+getCurrentRound()
+updatetScore(playerId : int, bubbleType : int)
+updatePlayerLocations(posP1 : int, posP2 : int)
+handlePlayerBubbleCollisions()
+handleBubbleWallCollisions()
+handleBubbleMirrorCollisions()
+handleBulletBubbleCollision()
+checkCollisions()
+changeRound()
+getPlayerLocation(playerId : int)

Attributes:

Private round : Round-> By using object of the class Round, GameEngine class gets informations on the objects that are going to make up certain round such as bubbles, mirrors, etc. The information transferred from the Round includes positions of objects which will be used to determine and react to various type of collisions happened in game.

Private soundManager : SoundManager -> This is the object of the class Sound Manager which will be used to communicate and provide different sounds effects and background music.

Private lives : int -> Lives of the players is stored in this class and not in the player objects as players share a life pool.

Constructors:

public GameEngine(Round round) -> This constructor initializes the Game Engine object.

Methods:

public Boolean bubblesLeft() -> checks whether any bubbles left in the round or not.

public int getRemainingLives() -> This method returns the remaining total lives of the players.

public Round getCurrentRound() -> This method returns the current round that's being played.

public void updateScore(int playerID, int bubbleType) -> This method gets a playerID and an enumerated bubble type. Then, updates the score of that very player accordingly.

public void updatePlayerLocations(posP1: int, posP2 : int) ->This method is used to update locations of the characters that players control using inputs taken from the keyboard. As characters are only moving in the + and – x directions, y coordinates are do not needed to be specified here.

public int getPlayerLocation(int playerID) -> This method gets a playerID as an argument and returns the location of that player.

public void handlePlayerBubbleCollisions() -> This method finds and handles collisions between Bubble and Player objects. Positions and properties of this objects are stored in the game entities subsystem by Round object. By information taken from game entities, this method finds and performs needed actions when player and bubble collides.

public void handleBubbleWallCollision() -> Similar to handlePlayerBubbleCollision, this method also communicates with the game entities and handles the reflections of the bubbles from walls in the game which corresponds to all 4 sides of the screen.

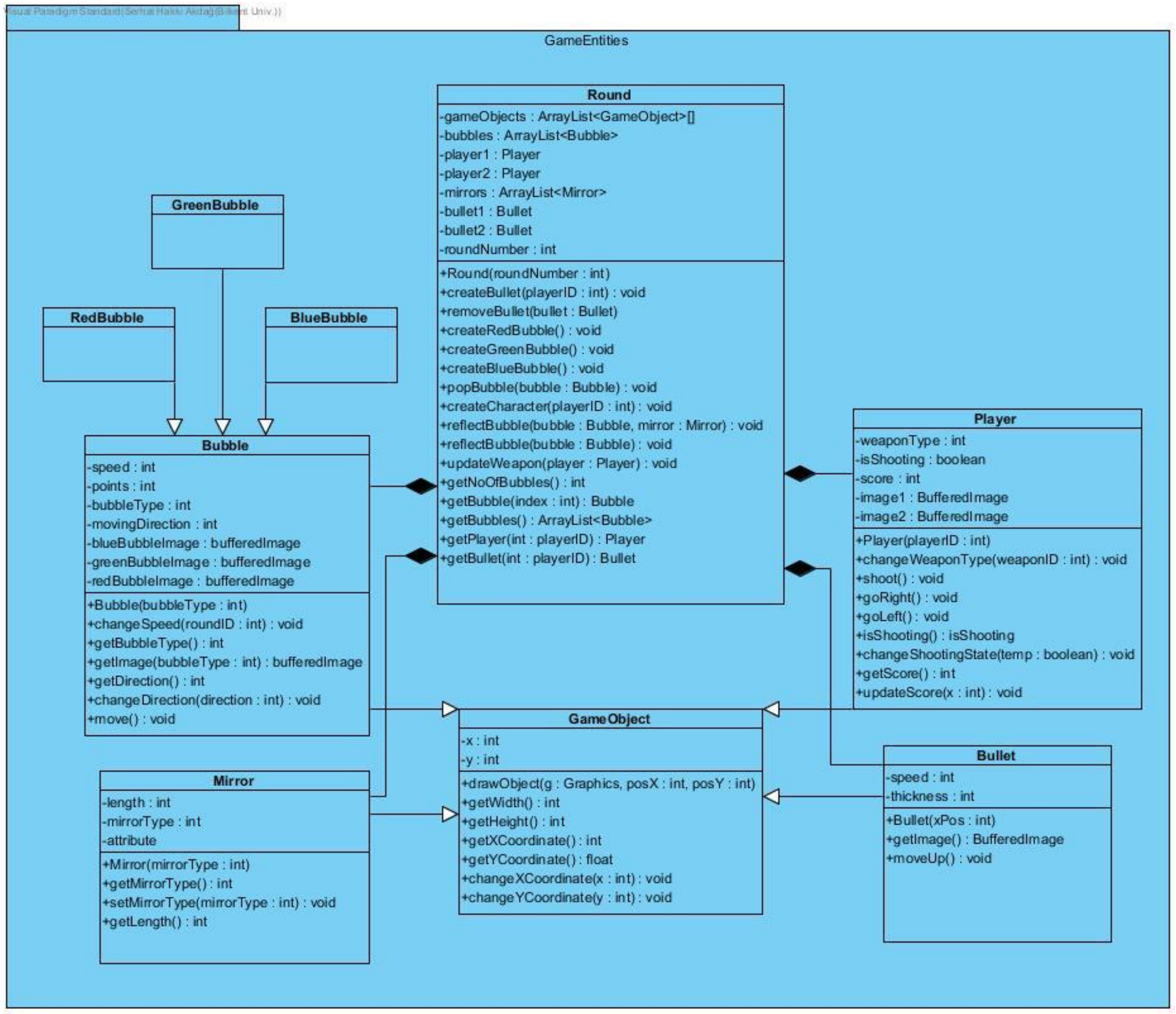
public void handleBubbleMirrorCollision() -> Similar to handleBubbleWallCollision, this method is used to handle collisions between Bubbles and Mirrors that are created and placed at the start of each round.

public void handleBulletMirrorCollision() -> This method is used to pop the bubbles that are successfully shot by players and increment their points.

public void checkCollisions() -> This methods checks and handles all collisions using 4 handler methods given above.

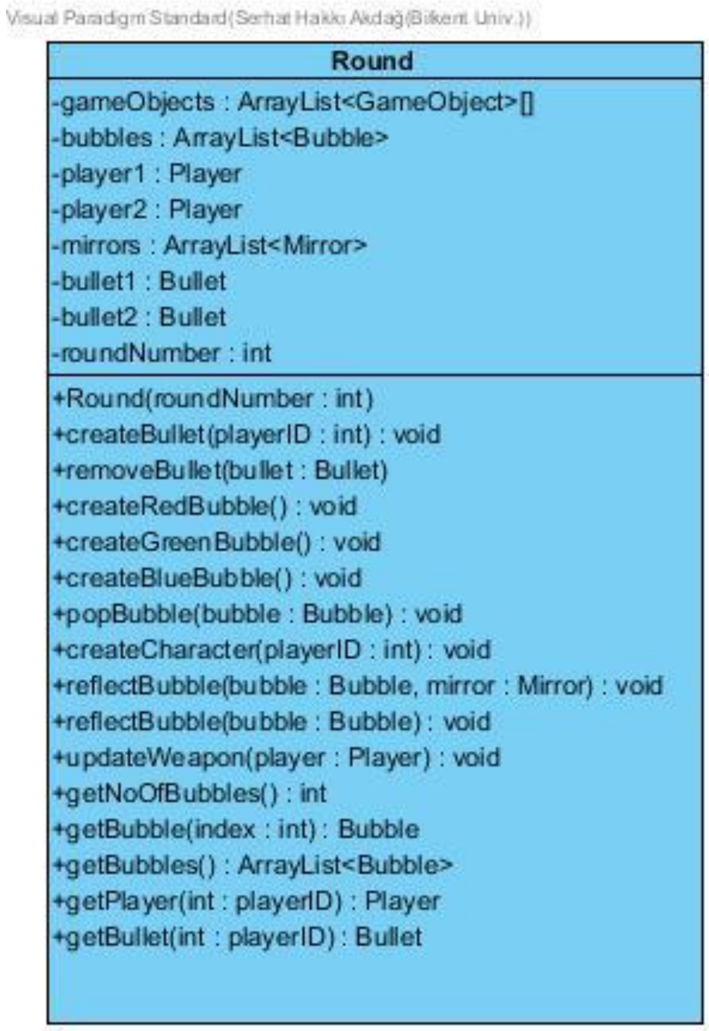
public void changeRound() -> This method increments round and makes game to move on to the next round by giving signals for creation of the objects, placements of them, etc.

3.4 Game Entities Subsystem


Visual Paradigm Standard(Serhat Haldu Akdag(Bilkent Univ.))
GameEntities
Round
-gameObjects : ArrayList<GameObject>[]
-bubbles : ArrayList<Bubble>
-player1 : Player
-player2 : Player
-mirrors : ArrayList<Mirror>
-bullet1 : Bullet
-bullet2 : Bullet
-roundNumber : int
+Round(roundNumber : int)
+createBullet(playerID : int) : void
+removeBullet(bullet : Bullet)
+createRedBubble() : void
+createGreenBubble() : void
+createBlueBubble() : void
+popBubble(bubble : Bubble) : void
+createCharacter(playerID : int) : void
+reflectBubble(bubble : Bubble, mirror : Mirror) : void
+reflectBubble(bubble : Bubble) : void
+updateWeapon(player : Player) : void
+getNoOfBubbles() : int
+getBubble(index : int) : Bubble
+getBubbles() : ArrayList<Bubble>
+getPlayer(int : playerID) : Player
+getBullet(int : playerID) : Bullet
GreenBubble
RedBubble
BlueBubble
Bubble
-speed : int
-points : int
-bubbleType : int
-movingDirection : int
-blueBubbleImage : bufferedImage
-greenBubbleImage : bufferedImage
-redBubbleImage : bufferedImage
+Bubble(bubbleType : int)
+changeSpeed(roundID : int) : void
+getBubbleType() : int
+getImage(bubbleType : int) : bufferedImage
+getDirection() : int
+changeDirection(direction : int) : void
+move() : void
Player
-weaponType : int
-isShooting : boolean
-score : int
-image1 : BufferedImage
-image2 : BufferedImage
+Player(playerID : int)
+changeWeaponType(weaponID : int) : void
+shoot() : void
+goRight() : void
+goLeft() : void
+isShooting() : isShooting
+changeShootingState(temp : boolean) : void
+getScore() : int
+updateScore(x : int) : void
GameObject
-x : int
-y : int
+drawObject(g : Graphics, posX : int, posY : int)
+getWidth() : int
+getHeight() : int
+getXCoordinate() : int
+getYCoordinate() : float
+changeXCoordinate(x : int) : void
+changeYCoordinate(y : int) : void
Mirror
-length : int
-mirrorType : int
-attribute
+Mirror(mirrorType : int)
+getMirrorType() : int
+setMirrorType(mirrorType : int) : void
+getLength() : int
Bullet
-speed : int
-thickness : int
+Bullet(xPos : int)
+getImage() : BufferedImage
+moveUp() : void

This sub system is responsible for the game objects that are created during the game-play. It is linked to the Game Manager subsystem as it uses the objects to manage the functions of the game. This system is responsible for the actual individual objects are their specifications. The main class here is Round which initializes all items in the beginning of a specific round, the round class inherits gameobject classes which is responsible for all objects created and destroyed throughout the game.

Round Class



```
-gameObjects : ArrayList<GameObject>[]
-bubbles : ArrayList<Bubble>
-player1 : Player
-player2 : Player
-mirrors : ArrayList<Mirror>
-bullet1 : Bullet
-bullet2 : Bullet
-roundNumber : int
+Round(roundNumber : int)
+createBullet(playerID : int) : void
+removeBullet(bullet : Bullet)
+createRedBubble() : void
+createGreenBubble() : void
+createBlueBubble() : void
+popBubble(bubble : Bubble) : void
+createCharacter(playerID : int) : void
+reflectBubble(bubble : Bubble, mirror : Mirror) : void
+reflectBubble(bubble : Bubble) : void
+updateWeapon(player : Player) : void
+getNoOfBubbles() : int
+getBubble(index : int) : Bubble
+getBubbles() : ArrayList<Bubble>
+getPlayer(int : playerID) : Player
+getBullet(int : playerID) : Bullet
```

This is the main class of the sub-system controlling all properties of each round in the game. It extends the gameObject class as well which extends all objects of the game.

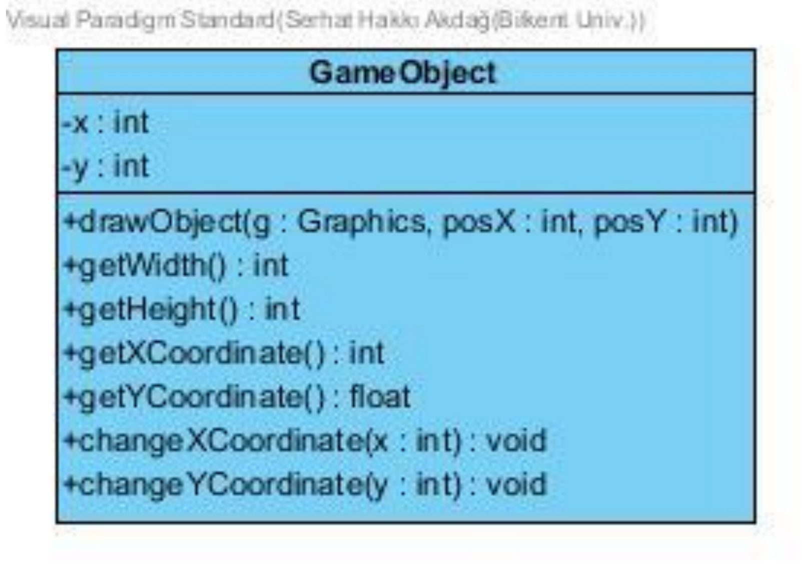
Attributes:

gameObjects: ArrayList<GameObject>[]: An array containing all the game objects. **Bubbles: ArrayList<Bubble> :** An array containing all type of bubbles in a current round.
Player1: Player: Stores information on player 1
Player2: Player: Stores information on player 2
Mirrors: ArrayList<Mirror>: An array containing all types of mirrors, with mirror objects.
Bullet1: Bullet: Stores information such as type of bullet of player 1's bullets
Bullet2: Bullet: Stores information such as a type of bullet of player 2's bullets.
RoundNumber: int: Stores the current round number the players are on.

<u>Methods:</u>
Round(RoundNumber rNumber): Initiates the current round the player is on, initializes all settings such as scores, lives, weapons, bubbles and mirrors depending on scores and levels.
CreateBullet(playerID : int) : Bullet: Shoots bullets depending on the weapon of the player
RemoveBullet(bullet : Bullet): Removes bullet depending on what it collides with
CreateRedBubble() : Bubble: Creates the coloured bubble depending on what level the players are on. (Same case for green and Blue bubble method creations)
PopBubble(Bubble : Bubble): Removes bubble when its hit by a bullet
createCharacter(playerID: int) : Player: Initializes the two characters in level 1
reflectBubble(bubble : Bubble, mirror: Mirror): Changes direction and manages the reflection of a bubble based on the mirror type that it collides with.
reflectBubble(bubble : Bubble): Changes direction and manages the reflection of the bubble based on the way it collides with the boundaries(walls) of the game.
updateWeapon(player : Player): Updates the weapon of a player after every round if they have reached the required score cap. Weapons start from pistols and are upgraded to shotguns then AK-47s.
getNoOfBubbles() : int: Returns number of current bubbles in a round.
getBubble(index : int) : Bubble: Returns the type of bubble according to its index in the bubble array list.

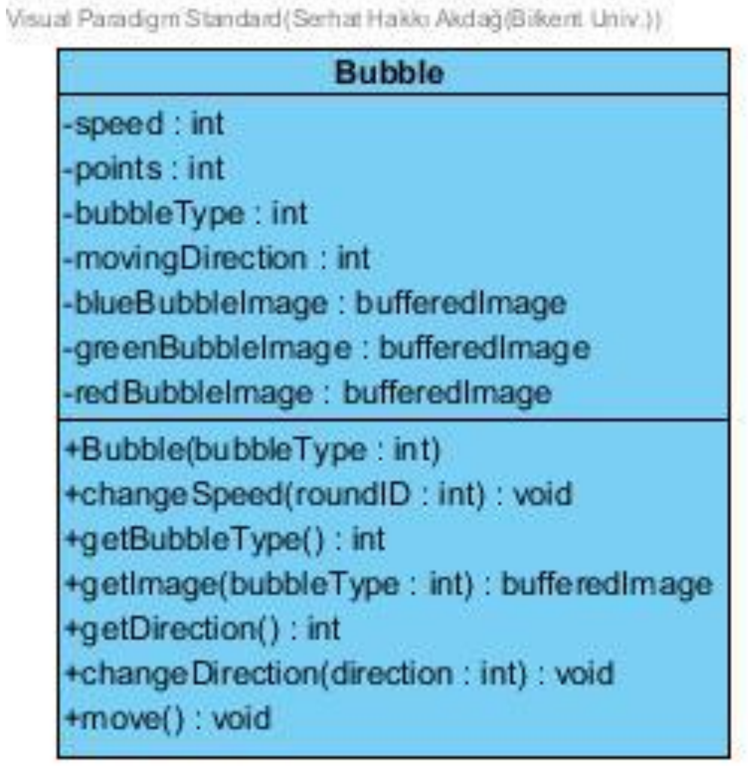
getBubbles(): ArrayList<Bubble>: Returns all current bubbles in a round as an array list
getPlayer(int : playerID) : Player: Gets the player data according to their IDs
getBullet(int : playerID) : Bullet: Gets the type of bullet depending on which player you specify.

Game Object Class:
Responsible for all objects that are seen during game play.


Visual Paradigm Standard(Serhat Hakkı Akdağ(Bilkent Univ.))
GameObject
-x : int
-y : int
+drawObject(g : Graphics, posX : int, posY : int)
+getWidth() : int
+getHeight() : int
+getXCoordinate() : int
+getYCoordinate() : float
+changeXCoordinate(x : int) : void
+changeYCoordinate(y : int) : void

<u>**Attributes:**</u>
X: int: X coordinate of the game object
Y: int: Y coordinate of the game object
<u>**Methods:**</u>
DrawObject(g: Graphics, posX : int, posY : int): Based on the level the objects will be drawn using java graphics and their positions will also be provided in coordinates to the method. The coordinates will be randomized for the bubbles. Players will be drawn at a constant y coordinate but their x coordinate will be a variable that can change as users wish to move. The guns will have same coordinates as players. The mirrors will also be randomly positioned, and perhaps in difficult places depending on difficulty of level.
GetWidth() : int : Gets the width of the object
GetHeight() : int : Gets height of the object
GetX/Ycoordinate() : float: Gets specified coordinate of the object (X/Y)
ChangeX/Ycoordinate(x/y : int) : void: Changes X or Y coordinate with arguments

Bubble Class:


Visual Paradigm Standard(Serhat Hakkı Akdağ(Bilkent Univ.))
Bubble
-speed : int
-points : int
-bubbleType : int
-movingDirection : int
-blueBubbleImage : bufferedImage
-greenBubbleImage : bufferedImage
-redBubbleImage : bufferedImage
+Bubble(bubbleType : int)
+changeSpeed(roundID : int) : void
+getBubbleType() : int
+getImage(bubbleType : int) : bufferedImage
+getDirection() : int
+changeDirection(direction : int) : void
+move() : void

Responsible for all bubble instantiations in a round. Extends the three different type of bubbles (red, blue, green) which have attributes of points(10,20,50)

<u>**Attributes:**</u>

Speed: int: Current speed of the bubble (dependant on type of bubble)

Points: int: The points attributed to the different type of bubble

BubbleType: int: Represents the different type of bubble: green, red or yellow depending on the integers.

MovingDirection: int: Represents the way the direction of the bubble's movement

blueBubbleImage/greenBubbleImage/redBubbleImage: bufferedImage: The image of the different type of bubbles

<u>**Methods:**</u>

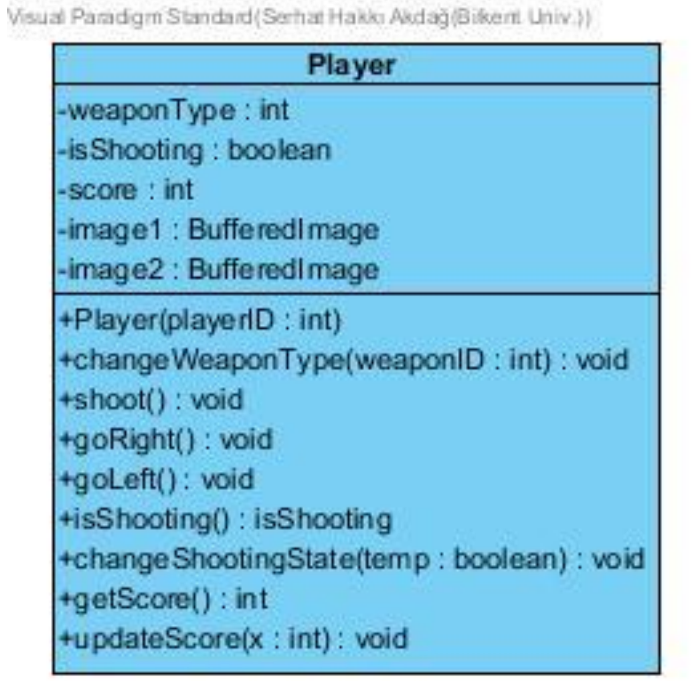
Bubble(bubbleType : int): Type of bubble corresnponding to a specific integer

ChangeSpeed(roundID : int) : void: Alters speed depending on type.

getBubbleType() : int: Gets the current bubble's type

getImage(bubbleType : int) : Returns the image of the bubble corresponding to the parameters specified
getDirection(): int : Returns the bubble's direction of movement
changeDirection(direction : int) : void: Can change the direction of the bubble with a specified parameter
move(): void: Moves the bubble by using current direction of it specified as a parameter of the class Bubble

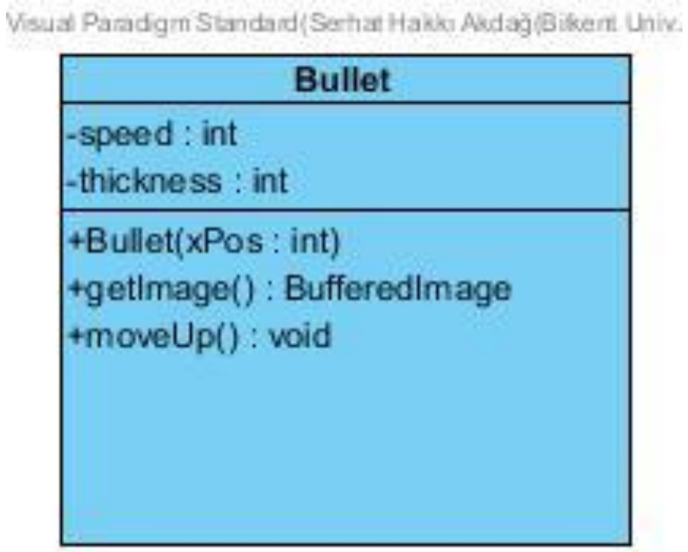
Player Class:
Responsible for the two players, their weapons, scores and locations.


Visual Paradigm Standard(Serhat Hakkı Akdağ(Bilkent Univ.))
Player
-weaponType : int
-isShooting : boolean
-score : int
-image1 : BufferedImage
-image2 : BufferedImage
+Player(playerID : int)
+changeWeaponType(weaponID : int) : void
+shoot() : void
+goRight() : void
+goLeft() : void
+isShooting() : isShooting
+changeShootingState(temp : boolean) : void
+getScore() : int
+updateScore(x : int) : void

<u>Attributes:</u>
weaponType : int : Type of weapon distinguished by an integer value.
Score : int: Current score of each player
isShooting : boolean: True or false value stored depending on whether a specific player is currently shooting or not.
image1/image2: **BufferedImage**: Stores the images of player 1 and 2's graphics.
<u>Methods:</u>
Player(playerID : int): Plyaer 1 or 2 creation

changeWeapon(weaponID : int) : void : Changes weapon based on scores after each round.
changeWeaponType(weaponID : int) : void: Changes the players weapon type with a specified parameter usually after a round is completed.
Shoot(): void: Shoots the bullets from a gun of a player
goRight(): void: Increases the value of the player's X coordinate
goLeft(): void: Decreases the value of the player's Y coordinate
isShooting() : isShooting: Returns the boolean attribute isShooting, which says whether the player is shoorting or not
changeShootingState(temp : boolean) : void: While player changes from one state of shooting to not shooting it is temporarily false or true accordingly
getScore() : **int:** Returns the player's score
updateScore(x : **int) : void:** Updates a player's score after popping a bubble

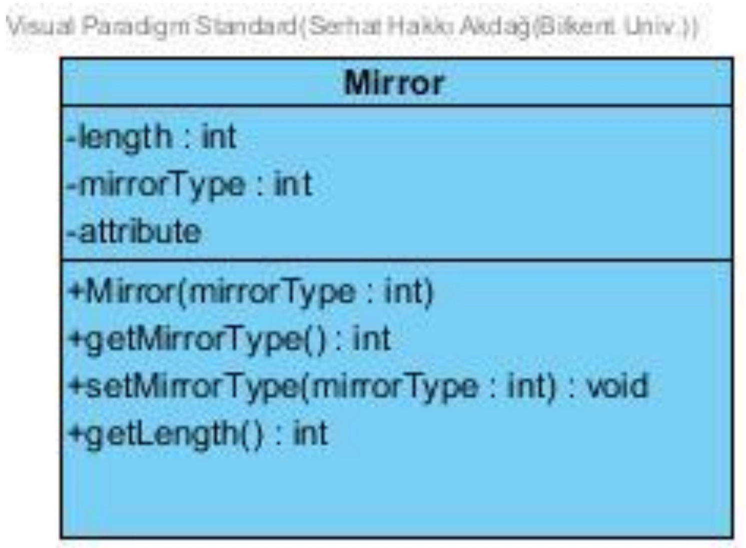
Bullet Class:
Responsible for all bullet instanations as they are shot by players through user-inputs


Visual Paradigm Standard(Serhat Hakkı Akdağ(Bilkent Univ.
Bullet
-speed : int
-thickness : int
+Bullet(xPos : int)
+getImage() : BufferedImage
+moveUp() : void

__Attributes:__
Speed: int: Speed of bullet dependant on weapon
Thickness : int: Size of bullet dependant on weapon
__Methods:__
Bullet(player ID: int, speed : int, thickness : int): Creates bullets dependant on player and their scores.

getImage() : BufferedImage: Returns the image of a bullet
moveUp() : void: Increases the value of the Y coordinate

Mirror Class:
Responsible for all mirror instantiations in the game.


Visual Paradigm Standard(Serhat Hakkı Akdağ(Bilkent Univ.))
Mirror
-length : int
-mirrorType : int
-attribute
+Mirror(mirrorType : int)
+getMirrorType() : int
+setMirrorType(mirrorType : int) : void
+getLength() : int

Attributes:
Length: int: Returns the length of the mirror
mirrorType: int: Returns the type of the mirror represented by integers
Methods:
Mirror(): Creates a mirror
getMirrorType(): int: Returns the type of mirror represented by an integer
setMirrorType(mirrorType : int) : void: Sets the mirror type with a specified parameter.
getLength() : int: Returns length attribute of the mirror