CS 319 - Object-Oriented Software Engineering

Design Report

Run, Dot Run!

Group 2-H

Beyza Tuğçe BİLGİÇ

Gökalp KÖKSAL

Gökçe ÖZKAN

Emine Ayşe SUNAR

Contents

1. Introduction

1.1. Purpose of the System

"Run Dot Run" is a game for entertainment which is mainly designed for kids and youth. It will consist of different levels. The levels will take short time, but will include some obstacles to pass and finish the game. The game will be easily understood, learned and used, while providing a user friendly UI.

1.2. Design Goals

a. Design Goals for End User

- **User-friendliness:**

  "Run Dot Run" will be a game that can be played by anyone at any age. However, because it is a game, our target audience will be consisted of mainly kids and youth. Therefore one of our goals is to provide a simple, attractive and well-organized interface that our target audience can easily understand and use. We will put a few buttons having basic features, such as starting the game and setting the volume of the music on the background, to the panels to decrease the complexity of the interface. Also, we will arrange the places of the buttons according to their priorities. For example, "Play Game" button, which is assumed to be the mostly used button on the main menu, will be placed in the middle of the screen, so that it will be less tiring for the user to use the tools.

- **Ease of Learning:**

  While playing games, people mostly don't prefer to spend time on learning how to play the game. Also, when it is regarded that there will be kids in our target audience, we should decrease the difficulty of understanding the game as much as possible. Therefore, one of the important goals of this project is to provide a game which can be easily learned, understood and adapted by the users. Otherwise the game can lose some of its users. In this scope, in "How to Play" panel, there will be short and clear description about the game and the first level will be very basic and easy for the users to understand what is going on in the game in general.

- **Ease of Use:**

  In a game, one of the possible disadvantages is the difficulty of controlling the game. Therefore, another important goal putting the end user at the core is ease of controlling the game. In "Run Dot Run", there will be clearly understood buttons, some of which will have images on themselves to make their functions more understandable, and the game will be controlled by some basic keys, such as left and right keys to move the dot, whose functions are easily recognizable.

Design goals for end user basically focus on providing a good UX to the audience.

b. Design Goals for Developer

- **Readability:**

  At the beginning of the project, we should consider that this game has more than one developers, and in the future there can be different developers working on this project. For a developer, the more the code is readable, the better and easier it is to work on it. Thus, creating a clear and readable API is one of our main goals regarding the maintainers.

- **Modifiability:**

  In this game, there are different levels and different components for those levels. We assume that in the future developments, it is highly possible to need some changes in the components and functions depending on the feedback of users or desires of the client. Therefore, one of our aims is to make modifications easy and less dangerous for the whole code.

- **Reuse of Components [4]:**

  For this project, we are more concerned about the reusability of the components inside the project, rather than in other projects. In different levels, we use different components and it is possible for the future levels that we will need to reuse them. That's why, in our project, we are designing our classes and components in a way that they their dependencies will have possibly the least effect in the others reusability.

- **Adaptability:**

  We want our program to be able to run in different machines as much as possible. Programs coded on Java can run on all CPU if Java interpreter exists. For this goal, we will use Java as our programming language, which is more portable than C and C++.

c. Design Goals for Client:

- **Flexibility [6]:**

  Our project will be adaptable to new requirements. We will try to make it least complex to add new features and functions to our design. For example, if there is a desire for new kind of obstacle, we will be able to add the new obstacle object directly to the system by connecting it to the GameObject class.

Trade-offs:

- **Usability vs. Functionality:**

  From the beginning to the end, we should assume that anyone at any age will play this game, because it is designed for entertainment of a wide target of audience. Therefore, regarding that our youngest user target will be kids, we should keep the game as simple, understandable and clear as possible. We don't need a high functionality for this project. As we need to increase the usability, the functionality will be decreased.

- **Performance vs. Memory:**

  In "Run Dot Run", we believe the performance is prior than memory, considering that in a game visuality, speed, effects etc. are important for attraction of the end user. Therefore, we focused on the performance rather than the memory. In order to increase the performance, we stored the objects of the game in a linkedlist, which take space in the memory while reaching the object in a faster way.

1.3. Definitions, acronyms and abbreviations

UI [1]: User Interface

UX [2]: User Experience

API [3]: Application Programming Interface

CPU [5]: Central Processing Unit

GameObject class: the class where all of the objects included in the game are connected

1.4. References

[1] http://searchmicroservices.techtarget.com/definition/user-interface-UI

[2] https://www.usertesting.com/blog/2015/09/16/what-is-ux-design-15-user-experience-experts-weigh-in/

[3]https://webcache.googleusercontent.com/search?q=cache:fFDAQUix3qsJ:https://en.wikipedia.org/wiki/Application_programming_interface+&cd=3&hl=tr&ct=clnk&gl=tr

[4] https://dzone.com/articles/reusable-components-in-java

[5] https://www.webopedia.com/TERM/C/CPU.html

[6] http://www.nada.kth.se/~karlm/prutt05/lectures/prutt05_lec7.pdf

2. Software Architecture

2.1. Subsystem Decomposition

Three-tier architecture is chosen since it fits for our project's structure. Three-tier architecture consists three parts: presentation, functional/process logic and data storage. Diagram that shows the project's three-tier principle is given below as Figure 1.

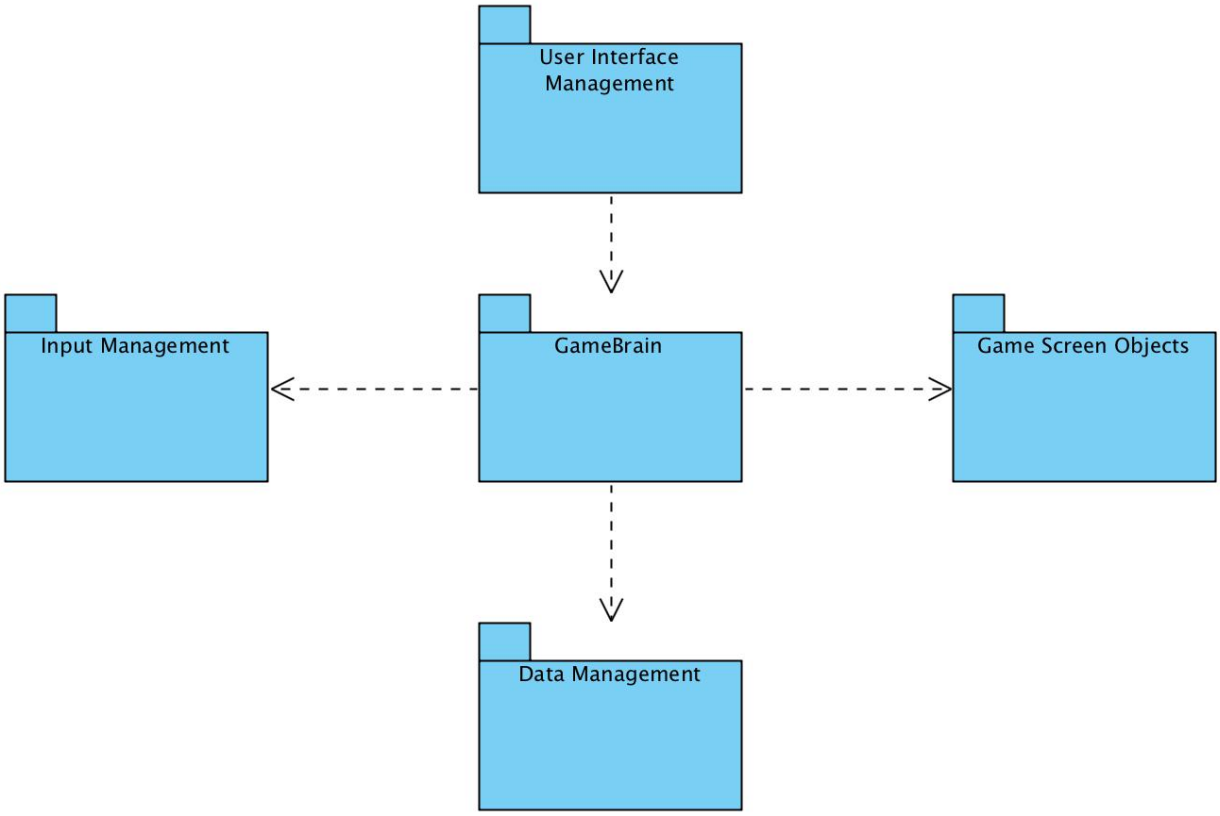

Figure 1 High Level Representation of Subsystem Decomposition.

User interface and its components are exist in the presentation tier. This tier is the first view that user faces with. Therefore, our MainMenu and all of its panels, buttons and components live at this tier. After user makes his/her actions on this tier, all of the signals get sent to the GameManager package to proceed.

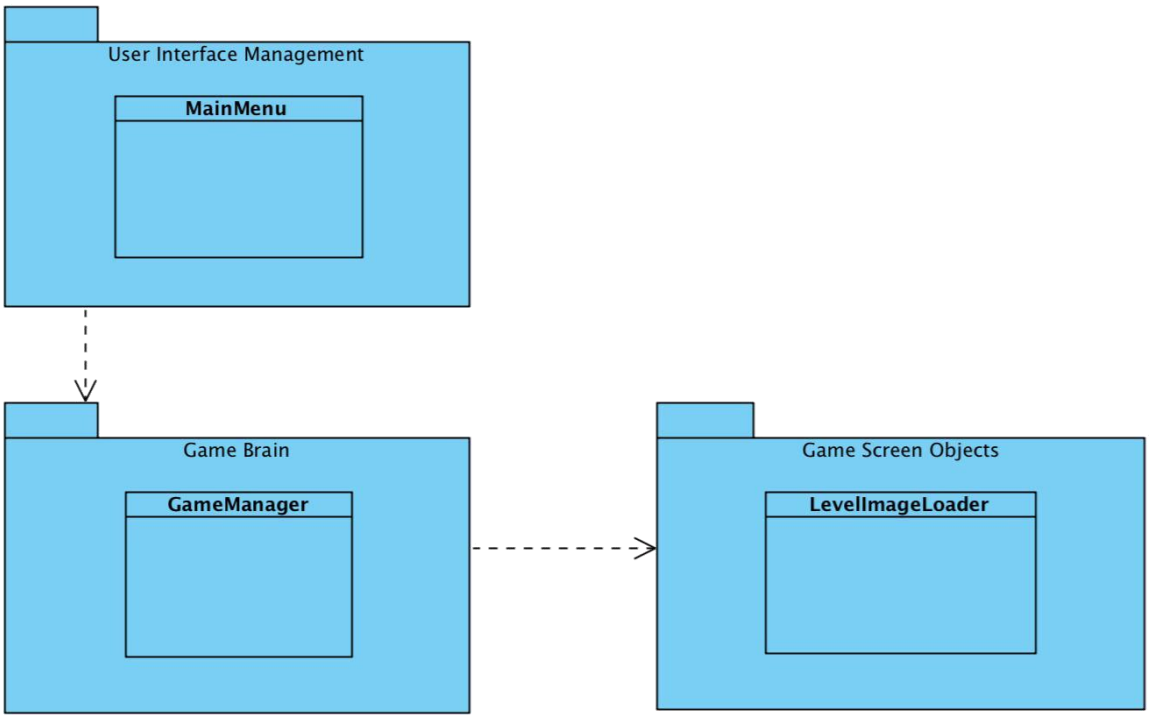

Figure 2 – Interaction between Tier 1 and Tier 2

To explain further actions, if the user clicks on the button "Play", program will go to our "LevelPanel" to let the user to select a level to play and here the logic tier is used. Then, with the help of "GameManagement" package and its helpers "LevelImageLoader", "Camera" and "GameObject" classes, the game will be set up with its logics. Afterwards, the user will start playing the game.

The data storage layer consists of classes that keeps the data needed for objects of the game. For instance, the coordinates of the player, status of collisions and time.

2.2. Hardware / Software Mapping

Software: Our project is being developed in Java as the programming language.

Therefore, Java Runtime Environment will be needed to run it.

Hardware: In terms of hardware requirements, user will use 3 buttons on keyboard to control the "Dot", in other words to control the player to left and right, and to jump. As a computer to run the program, most of the computers can handle our game since it is an old style, not complex game.

2.3. Persistent Data Management

The only data needs to be stored in our game is "LevelsCompleted" data, which stands for keeping the levels of the game completed by the user to let user know which levels he/she has completed. This data will be stored in users' hard drives. In addition, images of the levels will be

stored in a small file that will be in the game's files. Therefore no database will be constructed in our project to keep the track of the data.

2.4. Access Control and Security

The classes will be implemented in a way that only access to the game's logic system and the file system will be given to the necessary class GameManager. All the variables that should not be changed at any time will be defined as constant.

2.5. Boundary Conditions

If the players dies or the time is up, the game will restart. If the player finishes a level, a small panel and two buttons on it will be shown to let user to go either next level or Main Menu.

3. Subsystem Services

The whole class diagram is given below. The three-tier subsystem architecture could be seen here more clearly. The User Interface subsystem consists of the “MainMenu”, “DefaultPanel”, “LevelPanel”, “PauseGamePanel” and the “GameFinihedPanel”. The Game Logic subsystem consists of the “GameManager”, “LevelImageLoader” and the “Camera”. The Game Screen Elements subsystem consists of the “GameObject”, “Dot”, “Spike”, “Eraser”, “LetterBox” and “FadingLetterBox” class and also the “ObjectId” enumeration class.

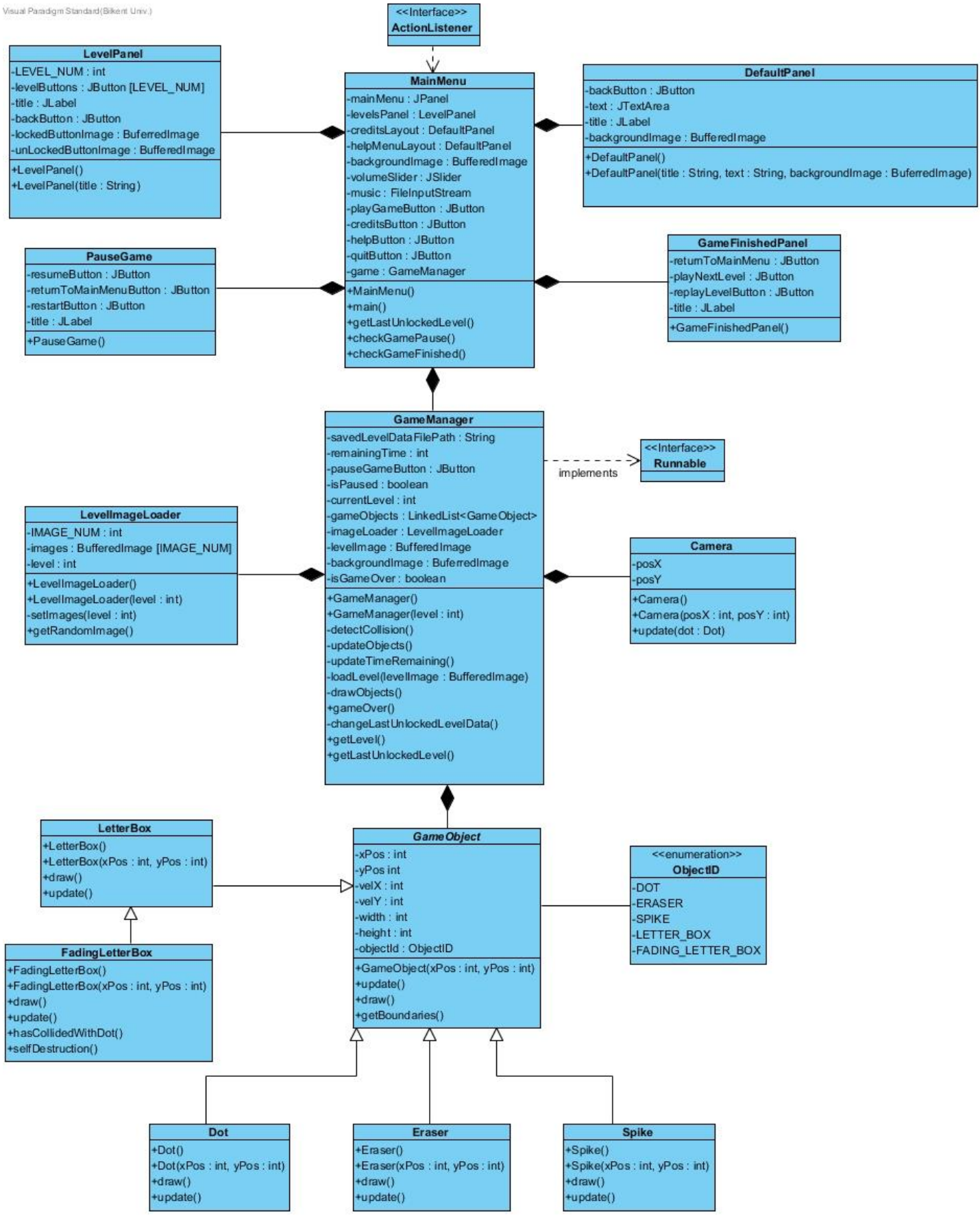

Figure 3 – Class Diagram

User Interface Subsystem:

This subsystem includes the classes which have the features of the interface of the game.

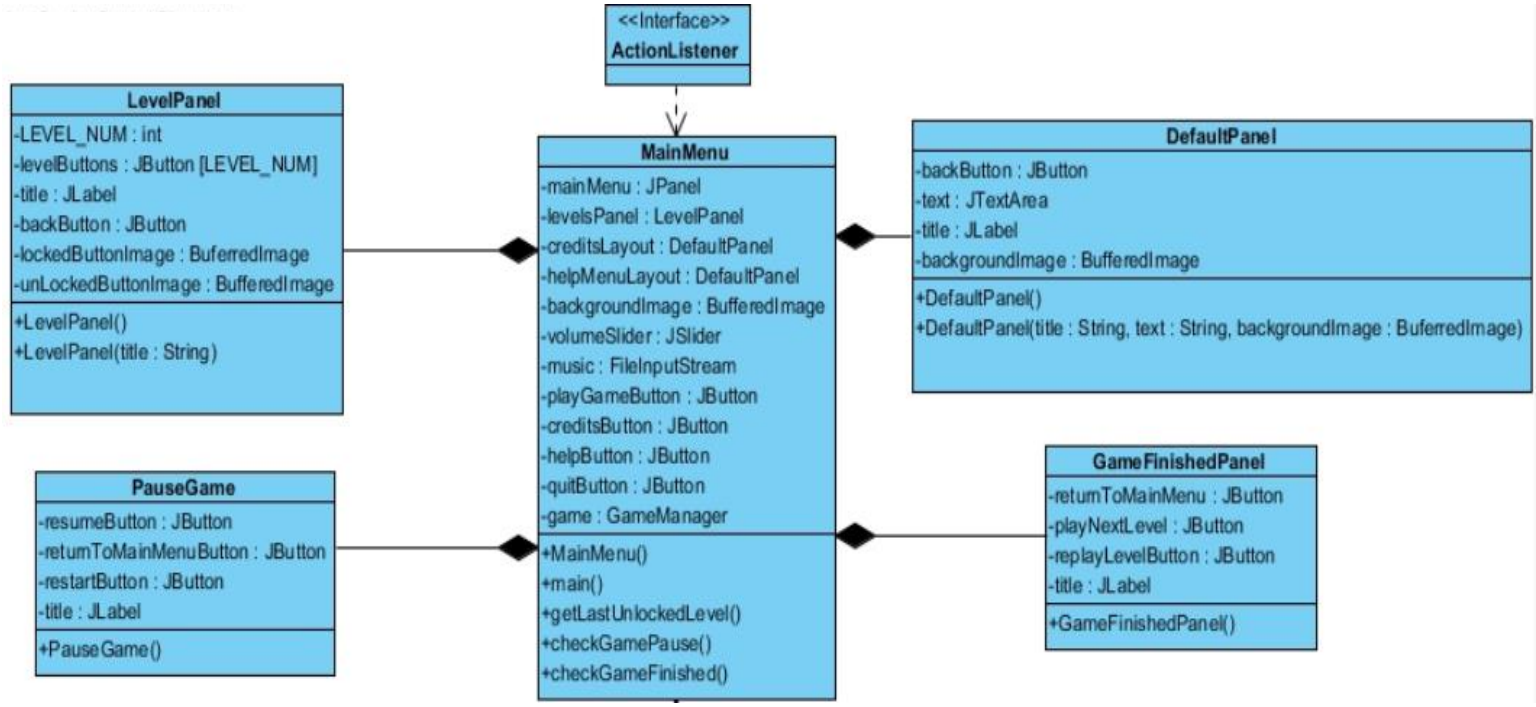

Figure 4 – User Interface Subsystem

MainMenu Class:

Figure 5 - Mainmenu Class

Attributes:

- **mainMenu:** mainMenu is a JPanel that will display the Main Menu contents.
- **levelsPanel:** levelsPanel is a LevelPanel which is a separate panel from mainMenu. It includes the features provided by LevelPalen class.
- **creditsLayout:** creditsLayout is a DefaultPanel, having the features of DefaultPanel class. In this panel, brief information about the developers of "Run Dot Run" will be shown as a text**.**
- **helpMenuLayout:** helpMenuLayout is also a DefaultPanel. It includes a text of description of the game.
- **backgroundImage:** backgroundImage is aBufferedImage which will be seen on the background of the game.
- **music:** music is a FileInputStream, which will be playing on the background of the game.
- **volumeSlider:** volumeSlider is a JSlider that enables the user control the volume of the background music
- **playGameButton**: playGameButton is a JButton that will direct the user to the levelsPanel when clicked.
- **creditsButton:** creditsButton is a JButton that will open creditsLayout.
- **helpButton:** helpButton is a JButton that will open helpMenuLayout.

- **quitButton:** quitButton is a JButton to quit the game.
- **game:** is a GameManager as the reference to the GameManager class.

Constructor:

- **MainMenu():** is the default constructor.

Methods:

- **main():** is the method to initiate the game.
- **setLastLockedLevel():** this method sets the last level to keep in which level the game is left and up to which level the others are unlocked.
- **getLastLockedLevel():** gives the last locked level.
- **checkGamePause():** returns true if the game is paused, and opens PauseGame panel.
- **checkGameFinished():** checks if the game is over and opens gameFinished panel.

DefaultPanel Class:

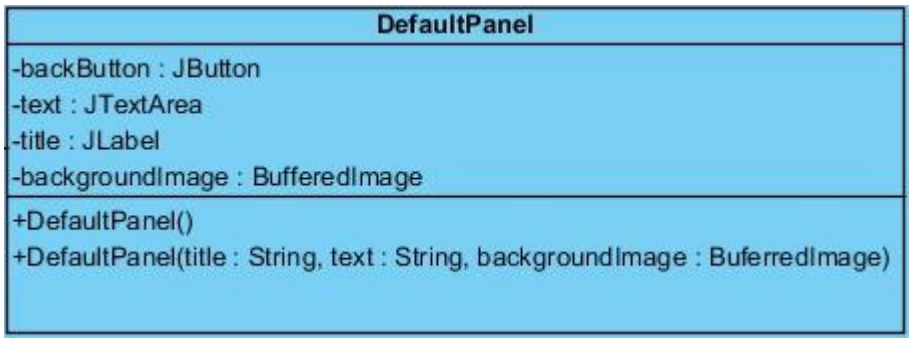

Figure 6 - DefaultPanel Class

Attributes:

- **backButton:** backButton is a JButton that will help the user go to the previous page when it is clicked.
- **text:** text is a JTextArea that will include some text depending on where it is called.
- **title:** title is a JLabel which will include the name of the current page.

Constructor:

- **DefaultPanel():** is the default constructor.

- **DefaultPanel(title: String, text: String, backgroundImage: BufferedImage):** this constructor takes the title, text and background image of the current page.

LevelPanel Class:

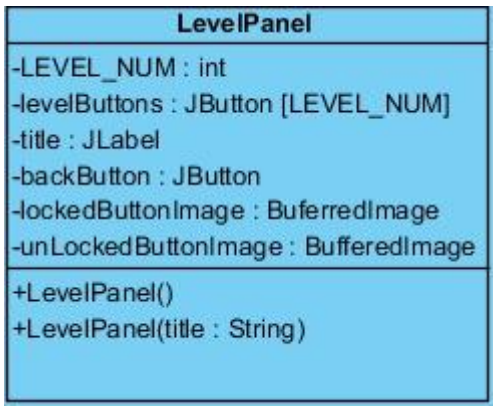

Figure 7 - LevelPanel Class

Attributes:

- **LEVEL_NUM:** is an integer for the number of levels.
- **levelButtons:** levelButtons is an array of JButton with size of LEVEL_NUM. Each button corresponds to different levels and starts the game when clicked.
- **title:** title is a JLabel including the name of the panel.
- **backButton:** is the button to help the user go to the previous page.
- **lockedButtonImage:** is a BufferedImage which will be placed on the levelButtons which are locked.
- **unLockedButtonImage:** is a BufferedImage which will be placed on the levelButtons which are unlocked.

Constructor:

- **LevelPanel():** default constructor.
- **LevelPanel(title: String):** this constructor takes the title as parameter.

PauseGame Class:

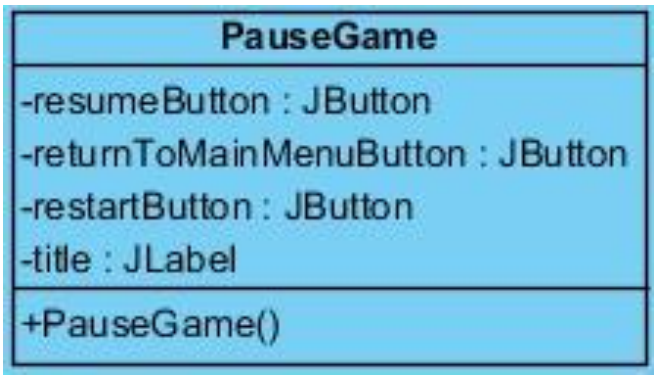

Figure 8 - PauseGame Class

Attributes:

- **resumeButton:** resumeButton is a JButton that returns the game from the pause panel.
- **returnToMainMenuButton:** is a JButton helping the user return to main menu page.
- **restartButton:** is a JButton which restarts the current level.
- title: is a JLabel for the title of the Pause page.

Constructor:

- **PauseGame():** is the default constructor.

GameFinishedPanel Class:

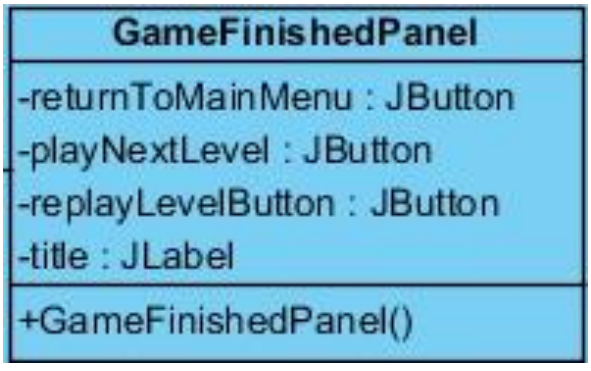

Figure 9 - GameFinishedPanel Class

Attributes:

- **returnToMainMenu:** is a JButton for the user to go back to the beginning page, which is the MainMenu.
- **playNextLevel:** is a JButton to pass the next level when the level is completed successfully.

- **replayLevelButton:** is a JButton which enables the user to play the same level again.
- **title:** is aJLabel for the title of GameFinishedPanel to be shown when the game is over.

Constructor:

- **GameFinishedPanel():** is the default constructor.

Game Logic Subsystem

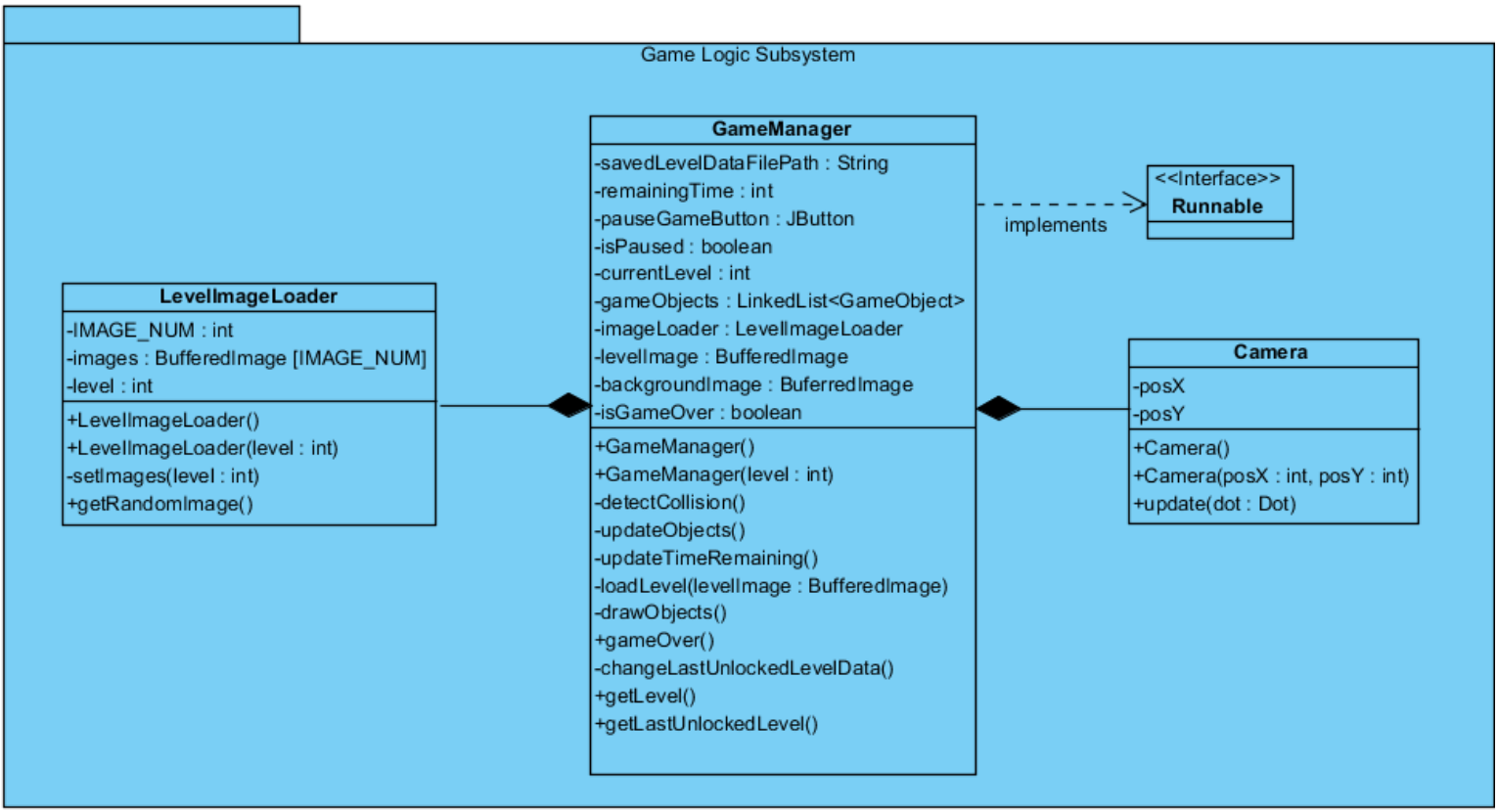

Figure 10 – Game Logic Subsystem

The figure illustrates the overall composition of the Game Logic Subsystem. The Game Logic Subsystem is responsible for handling and creating the objects of the game. In this subsystem, the positions of the objects will be specified and drawn in the screen.

Game Manager Class

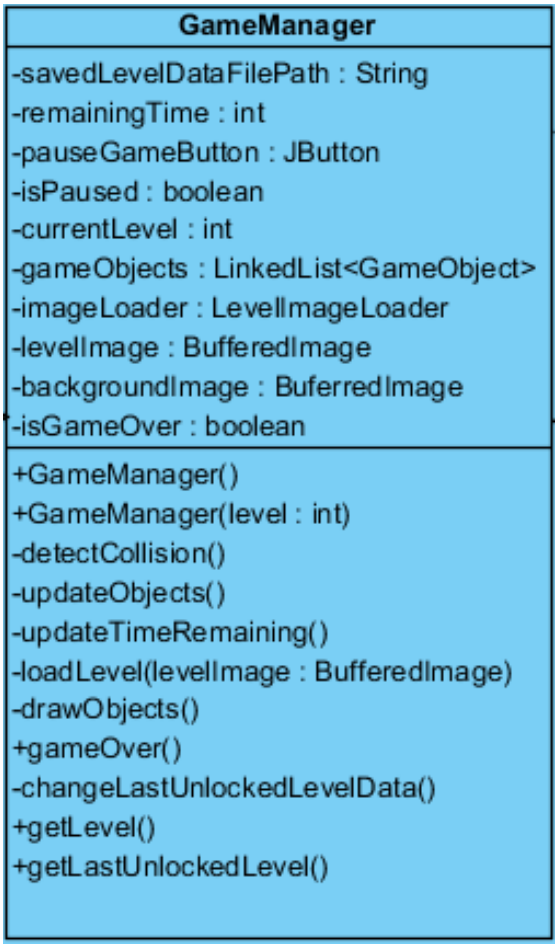

Figure 11 - GameManager Class

The GameManager class is the Façade class of the Game Logic subsystem, thus it is responsible of the creation of the objects that are going to be used in the game. A thread will be running in this class, so it will implement the interface "Runnable" in order to achieve this.

In the GameManager, the attribute savedLevelDataFilePath will be a string, which will contain the file's path of where the last unlocked level's data will be written. As the player unlocks levels, the last unlocked level's number will be written to this file by using the changeLastUnlockedLevelData() method. Also, the public method getLastUnlockedLevel() will be used by the MainMenu class (which was explained in the user interface subsystem) to give the last unlocked level data to the LevelsPanel class (which was also explained in the user interface subsystem).

The GameManager class will be initialized by giving the level's number to the constructer. By doing this, the GameManager will have the information of the level to be started. The image backgroundImage will be used to specify the locations of the objects such as Dot, letter boxes, spikes etc.

In this class, there will be a linked list of type GameObjects, which will hold all of the GameObjects, for example the Dot, letter boxes, fading letter boxes etc. All GameObjects will be added to this linked list to achieve efficiency.

A short description of the attributes and the methods in the GameManager class to clarify the functionality of each attribute and method:

Attributes:

- **savedLevelDataFilePath:** This is a string which specifies the path of the file that will be used to hold the last unlocked level information.
- **remainingTime:** This is an int which holds the remaining time.
- **pauseGameButton:** This is the button which will be pressed to pause the game during game play.
- **isPaused:** A Boolean attribute which will keep the data of whether the game is in pause.
- **currentLevel:** An int which will keep the current level number.
- **gameObjects:** This is a linked list of type GameObjects. The objects Dot, LetterBox, FadingLetterBox, Spike and Eraser will all be placed inside this linked list. A specific object will be reached by using a for loop.
- **imageLoader:** This is an LevelImageLoader object which will be used to load the image for a specific level.
- **levelImage:** This will be a BuferredImage which will be the image of the level. This image will be used to specify the positions of the objects in the class. (This is not the image that will be the background picture, it will be used to specify the locations of the GameObjects. Detailed information will be given when explaining the loadLevel method)
- **backgroundImage:** This will be a buferredImage will be the picture to be placed as he background picture.

Constructors:

- **GameManager:** This is the default constructor.
- **GameManager(level: int):** This is the constructor where the level number information will be given, thus the game will be initialized according to the level.

Methods:

- **detectCollision():** This method checks the collision between the Dot and the other GameObjects.
- **updateObjects():** This method will update the positions of the GameObjects in the class.

- **updateTimeRemaining():** The method which will update the remainingTime attribute accordingly.
- **loadLevel(levelImage: BufferedImage):** This method will take the attribute levelImage and specify the locations of the objects. This will be done by using the RGB values in the image. There will be different RGB values representing different GameObject types. This method will go through each pixel and when it comes across a specific RGB value, it will create and place the specific GameObject at this position. For example, whenever it comes across the RGB value (0, 0, 0) which is black, it will create and place a LetterBox object at the specific position.
- **drawObjects():** This method will draw each object in its updated location.
- **gameOver():** This method will check whether the game is over by checking whether the time is finished and by checking whether the Dot has reached to the end of the sentence.
- **changeLastUnlockedLevelData():** This method writes the last unlocked level's number into the file which has the path savedLevelDataFilePath. After the game is won this method will be used to update the data in this file.
- **getLevel():** This method will be used to get the current level number.
- **getLastUnlockedLevel():** This method will read the data from the savedLevelDataFilePath attribute and return the last unlocked level. This method will be public so that the MainMenu class could use this method and get the last unlocked level information to pass it on to the LevelsPanel.

LevelImageLoader Class

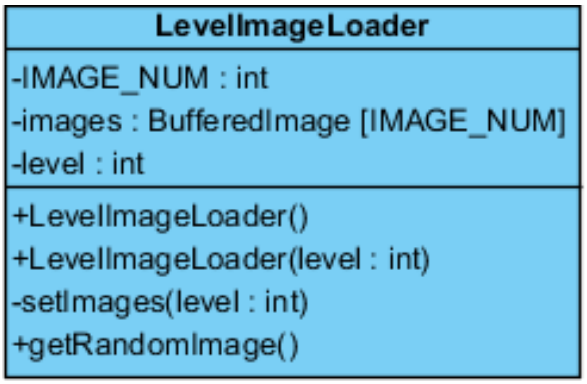

Figure 12 - LevelImageLoader Class

This class is responsible of providing the specific level image to the GameManager class. There will be a specific number of images for each specific level which will be held in an array of BuferredImages. A random level image will be given to the GameManager class. The reason for having different numbers and different types of level images is to present a variety of different game plays even for the same level for the player.

A short description of the attributes and the methods in the LevelImageLoader class to clarify the functionality of each attribute and method:

Attributes:

- **IMAGE_NUM:** This will be a final int value which will hold the number of images that will be held in the images array.
- **images:** This will be an array of size IMAGE_NUM, which will hold the different level images for the specific level.
- **level:** This will be an int value which holds the number of the level.

Constructors:

- **LevelImageLoader():** This is the default constructor.
- **LevelImageLoader(level: int):** This will initialize the array images according to the given level value. This will be done by using the setImages method.

Methods:

- **setImages(level: int):** This method is responsible of filling the images array when the constructor calls this method. This method will contain several images and according to the level, the images will be set in the images array.
- **getRandomImage():** This is a public method which is used by the GameManager class to get a random image from the images array and set it as the levelImage attribute of the GameManager class.

Camera Class

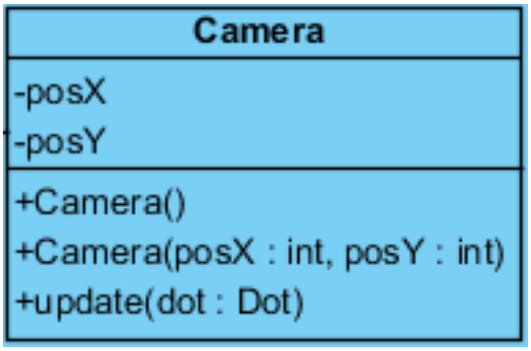

Figure 13 - Camera Class

The Camera class is responsible for the movement of the screen which is specified in the GameManager. The camera will have an initial position and the position of the camera will be updated according to the position of the Dot's position (As the Dot comes to the middle of the screen, the camera will move forward).

A short description of the attributes and the methods in the Camera class to clarify the functionality of each attribute and method:

Attributes:

- **posX:** The x coordinate of the camera's position.
- **posY:** The y coordinate of the camera's position.

Constructors:

- **Camera():** The default constructor.
- **Camera(posX, posY):** The constructor which initializes he camera class with the given x and y coordinates.

Methods:

- **Update(dot: Dot):** This method updates the position of the camera. It uses the position of the dot object to update the camera's position.

Game Screen Elements Subsystem

"Game Screen Elements Subsytem" plays a key role in declaring the objects which are shown in the screen while the game starts to running. There are many significant objects including "FadingLetterBox", "LetterBox", "Dot", "Eraser", "Spike" and "GameObject". This subsystem has also Enumaration named as "ObjectID" to indicate the types of different objects.

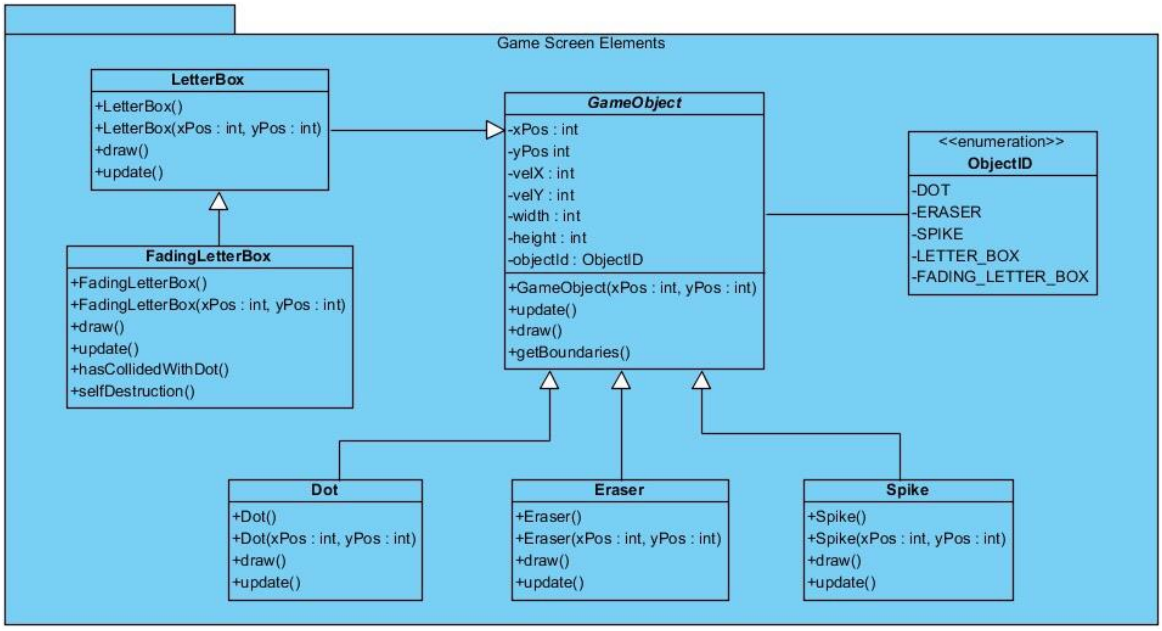

Figure 14 – Package Diagram of GameScreenElements

GameObject Class

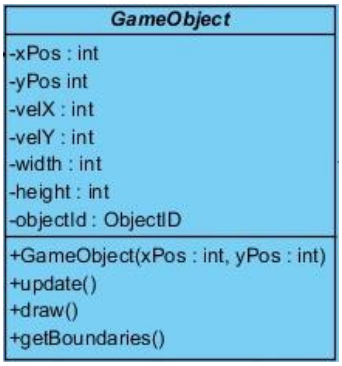

Figure 15 - GameObject Class

"GameObject" is an essential class for this subsystem. All of the objects have to keep an account of position information, velocity information, size information and object type hence, they use "GameObject" abstract class as a parent class which has all these attributes. This class stores the object type as "objectId" by using enumaration. Each object has a unique "objectId" defined as enum type. Also, there are abstract methods including "*update()*", "*draw()*" and *"getBoundaries()"* since all objects may act differently to these operations. "GameObject" will be instantiated whenever the user attemts to play the game.

A short description of the attributes and the abstract methods in the GameObject class to clarify the functionality of each attribute and method:

Attributes:

- **xPos :** The x coordinate of object's position.
- **yPos :** The y coordinate of object's position.
- **velX :** The velociy of object in the direction of x coordinate.
- **velY :** The velociy of object in the direction of y coordinate.
- **width :** The width of object.
- **height :** The height of object.
- **objectId :** The enum type of object.

Constructor:

- **GameObject(xPos : int, yPos: int) :** This will initialize a game object according to given position values.

Methods:

- **draw() :** This is an abstarct method. All objects will include their own implementations but generally it will achieve drawing the object according to its size and position information.
- **update() :** This is also an abstract method. It will manage the position of object accoridng to its updated position information.
- **getBoundaries() :** This is also an abstract method and will be responsible for returning the boundaries of an object according to some attributes such as position, width and height.

LetterBox Class

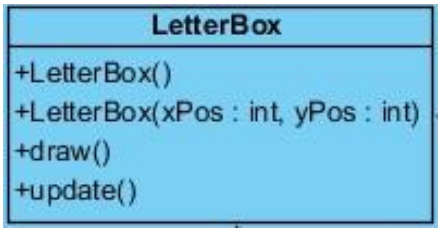

Figure 16 - LetterBox Class

The "LetterBox" class has the same attributes as the GameObject class but also includes two constructors, the implementations of *"draw()"* and "*update()*" abstract methods. This class will manage the game platform by drawing a number of letters and creating different sentences. The player has to jump over these letters without dropping between them. The LetterBox has a child class representing the fading letters which will be defined below.

The description of abstract methods is given above.

Constructors:

- **LetterBox() :** This is the default constructor.
- **LetterBox(xPos : int, yPos : int) :** This is the constructor where the position information will be given, thus the letter will be instantiated according to "xPos" and "yPos" values.

FadingLetterBox Class

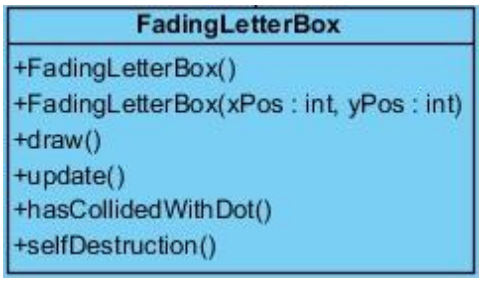

Figure 17 - LetterBox Class

The “FadingLetterBox” class is a child class of “LetterBox” class and has the same attributes as its parent class but also includes “*hasCollidedWithDot()*”and “*selfDestruction()*” methods as well as the implementations of *“draw()”* and “*update()*” abstract methods. This class will be initialized after the level including fading letters is played. The FadingLetterBox object has the abilty to disappear when the collision with the Dot object is detected by the “*hasCollidedWithDot()*” method. The “*selfDestruction()*” method enable the objects of this class to fade away in case of collision.

Constructors:

- **FadingLetterBox() :** This is the default constructor.
- **FadingLetterBox(xPos : int, yBox : int) :** This is the constructor where the position information will be given, thus the fading letter will be instantiated according to “xPos” and “yPos” values.

Methods:

- **hasCollidedWithDot() :** This method will detect the time when the Dot has collided a letter then, the letter will start to fade away.
- **selfDestruction() :** This method will enable the objects of this class to fade away in case of collision detected.

Dot Class

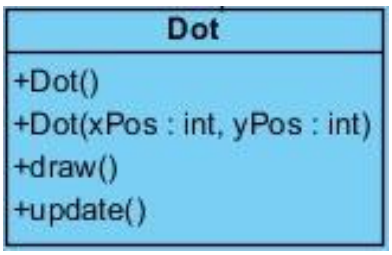

Figure 18 - Dot Class

The “Dot” class has the same attibutes as the GameObject class and the implementations of *“draw()”* and “*update()*” abstract methods exist in the class. The Dot object is as essential class for the game since it stands for the player who aims to finish the levels successfully. It will be created when the player starts to play first level.

Constructors:

- **Dot () :** This is the default constructor.

- **Dot (xPos : int, yBox : int) :** This is the constructor where the position information will be given, thus the Dot will be instantiated according to “xPos” and “yPos” values.

Eraser Class

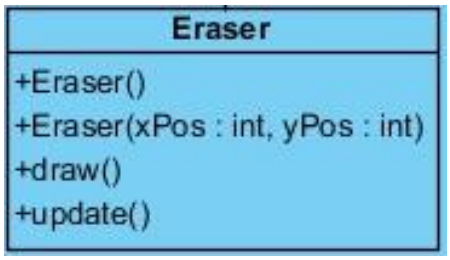

Figure 19 - Eraser Class

The “Eraser” class will be initialized when the player manages to play the level including the Eraser objects aimed at droping from top of the screen and hitting the Dot object as player. Hence, the Eraser object will take a part in exposing the player to some obstacles during the game.

Constructors:

- **Eraser() :** This is the default constructor.

- **Eraser(xPos : int, yBox : int) :** This is the constructor where the position information will be given, thus the Eraser will be instantiated according to “xPos” and “yPos” values.

Spike Class

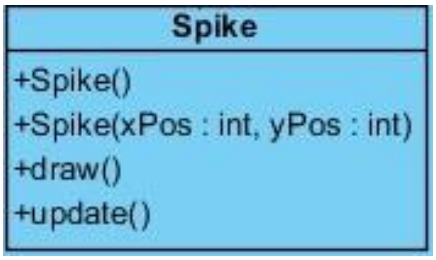

Figure 20 - Spike Class

The “Spike” class stands for another obstacle type during the game. It has the same attributes as the GameObject. The Spike object will be positioned according to its attributes including “xPos” and “yPos”.

Enumaration (ObjectID)

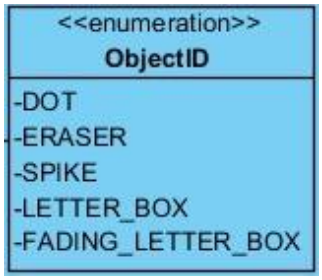

Figure 21 – ObjectID as Enum Types

"Game Screen Elements Subsytem" includes a lot of object types such as "Dot", "Eraser", "Spike", "LetterBox" and "FadingLetterBox". In attempt to make our implementation organization easy, we use enumaration types by specifying "ObjectID" related to each unique object. For instance, we can call "objectId.DOT" as objectId attribute of the Dot object when we define it in the "ObjectID" enumaration only once, since all objects use the GameObject as parent class having "objectID" attribute. With the advantage of enum types, the GameObject class can use different objects by calling only their unique "objectId".

Description of the Interactions between Classes According to the Use Cases

PlayGame: This is the main use case that user wants to start the game. When user clicks the "Play Game" button, LevelPanel that showing the levels of the game shows up on the screen. Here user will choose the level that he/she wants to play. After the level is chosen, "currentLevel" variable gets set and loadLevel() method gets called from the GameManager. Then, level gets loaded from LevelImageLoader class and drawObjects() method is called, and camera gets set on the player which is a dot and game loop starts. In this loop updateObjects() method updated the objects positions according to the user inputs to move the player around. After that, detectCollision() method checks whether the player hits any of the obstacles or gets falls down to the ground. If any collision gets detected between the player and the obstacles spike and eraser, the player will return to the beginning point to try again until the time expires. Therefore, during the game time will be checked with updateTimeRemaining() method. If there is no time left, isGameOver() method will return true and game will be over. A panel will be shown with two buttons on it to let user to choose either go back to the MainMenu or restart the level.

PauseGame: After the game starts and there will be a button “Pause” to pause the game and PauseGameMenu panel that has two buttons on it, “Resume” and “Exit”, will be shown when clicked. IsPaused variable in GameManager will be set to true when this button is clicked and will be set to false when “Resume” or “Exit” are clicked. Besides, updateTimeRemaining() method will also be stopped until the “Resume” is clicked.

ChangeVolume: This is the use case that player sets the volume using the volume bar shows up in the MainMenu class. When adjusting the volume, game will be pause on the background and it will continue when button “Exit” is clicked.

HowToPlay: HelpMenuLayout panel will be shown when button “HowToPlay” clicked. In the panel, the button information that user will need to move the player and the rules of the game will be shown.

Credits: CreditsLayout Panel will be shown when button “Credits” clicked. In the panel, the information about developers of this game will be given.

Quit: When the “Quit” button is clicked, program will save the levels that user has reached in a file that game’s file contains and terminates the program.