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 :**

  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:**

  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

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.

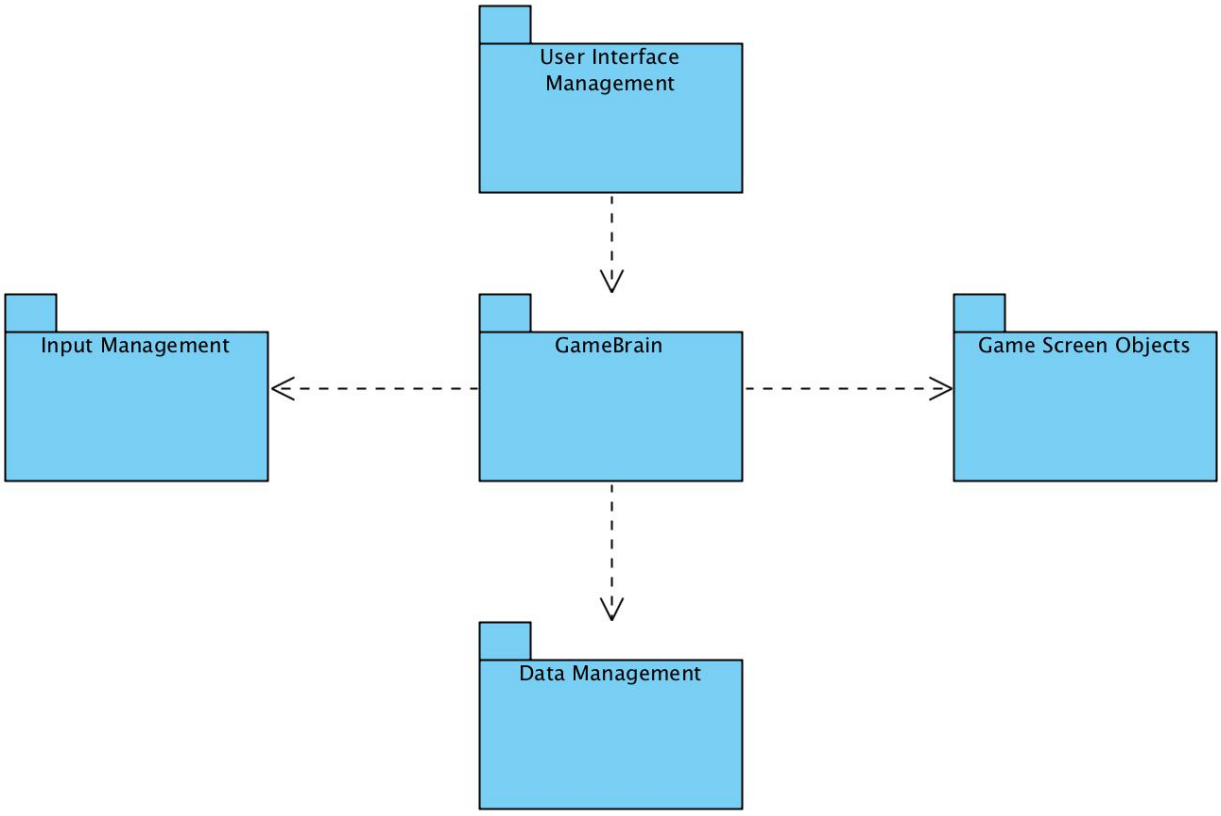

Figure 1 High Level Representation of Subsystem Decomposition.

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.

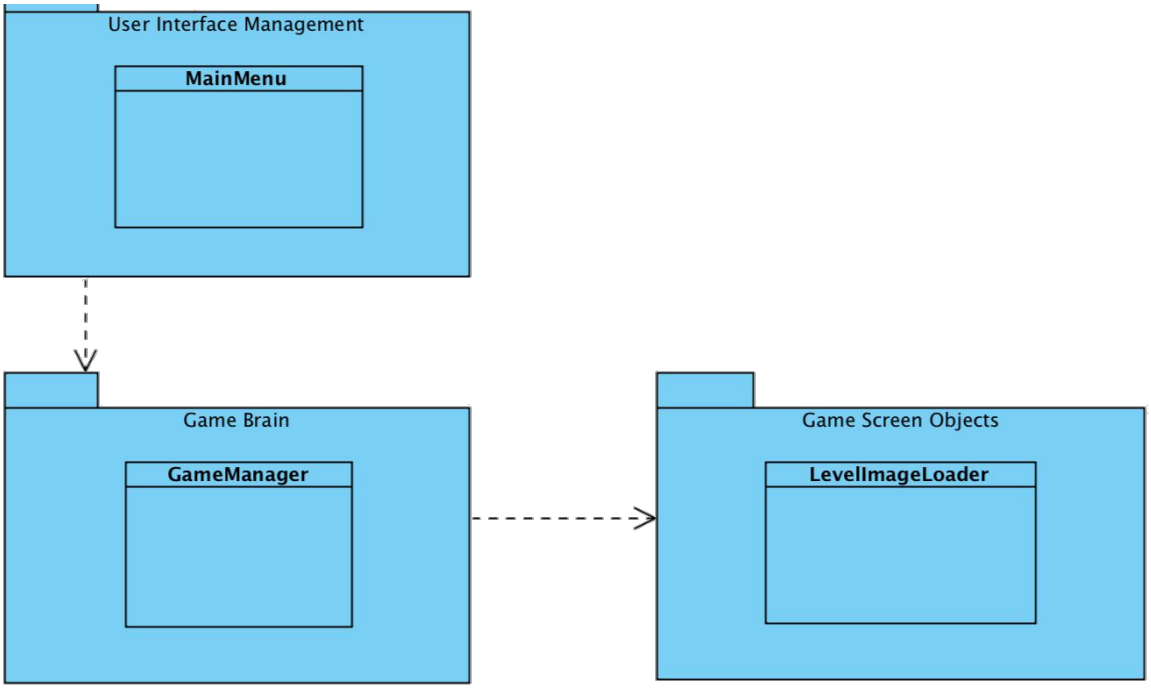

Figure 2 – Interaction between Tier 1 and Tier 2

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 “MainMenuPanel”, “DefaultPanel”, “LevelPanel”, “PauseGamePanel”, “GameFinihedPanel”, “MusicManager” and the “MainMenu”. The Game Logic subsystem consists of the “GameManager”, “LevelImageLoader”, “Camera” and the “CollisionDetection”. The Game Screen Elements subsystem consists of the “GameObject”, “Dot”, “Spike”, “Eraser”, “LetterBox”, “FadingLetterBox”, “Extra_Life”, “TimeBonus”, “TimePunishmentCircle”, “CheckPoint” class and also the “ObjectId” enumeration class.

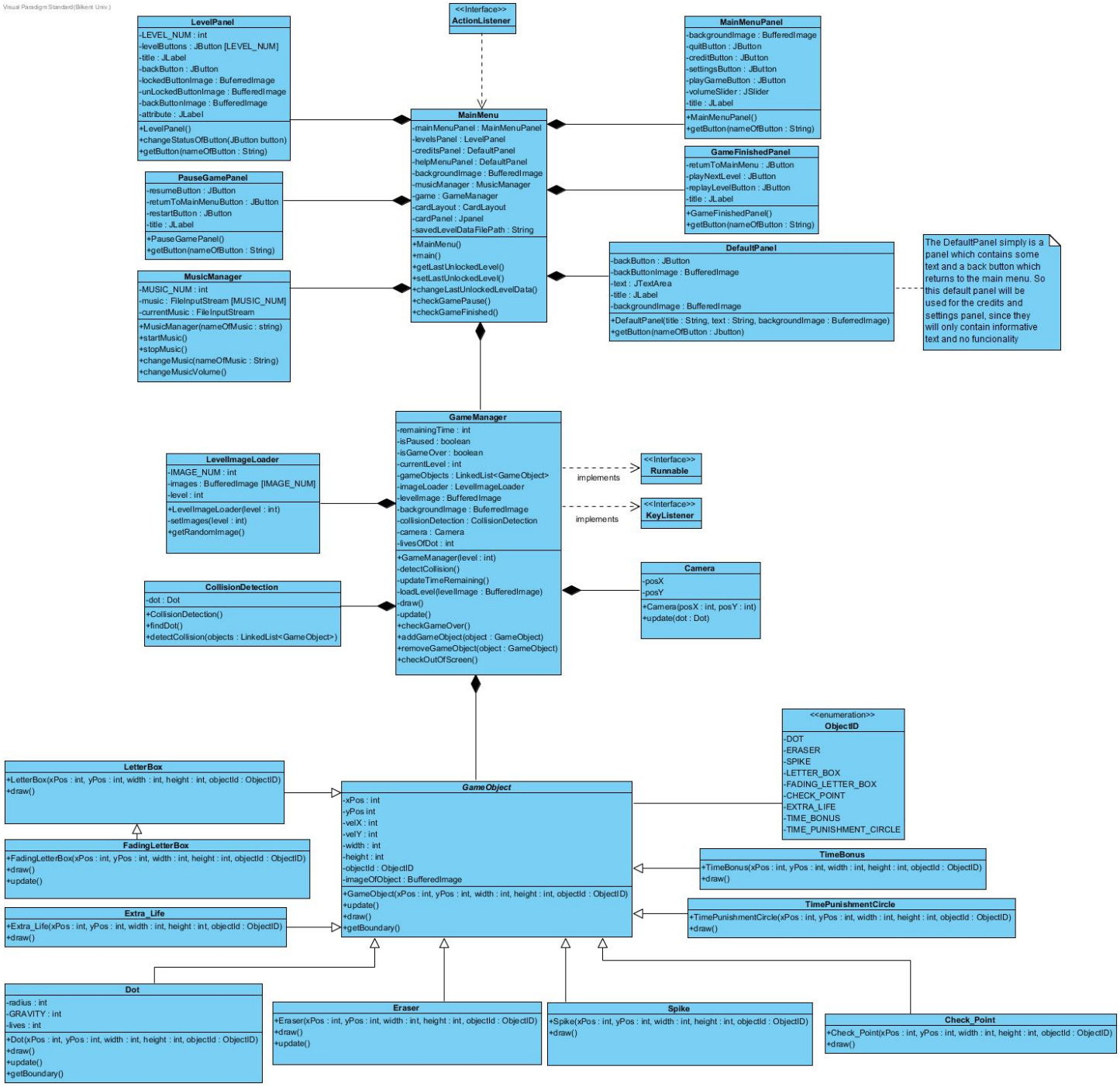

Figure 3 – Class Diagram

User Interface Subsystem:

This subsystem includes the classes which have the features of the interface of the game. It includes 7 classes which are, MainMenuPanel, LevelPanel, DefaultPanel, GameFinishedPanel, PauseGamePanel, MusicManager and MainMenu. MainMenu is the class that uses all the panels and MusicManager, and the GameManager to initiate the game. The ActionListener interface is to proceed the game according to the clicked buttons.

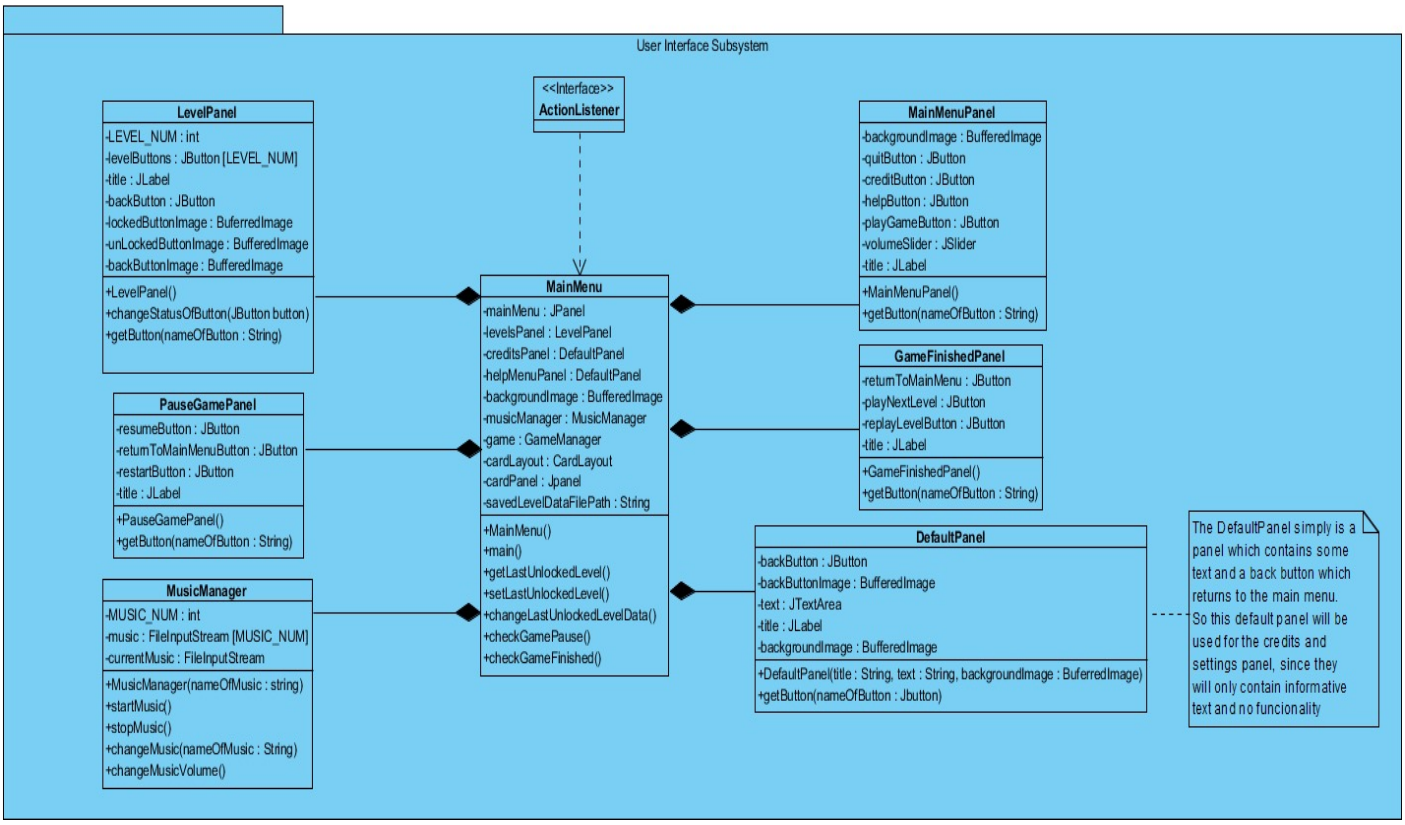

Figure 4 – User Interface Subsystem

MainMenuPanel Class:

MainMenuPanel is a panel which include 4 buttons and a slider. First button is helpButton to be placed at the upper left corner of the screen and it opens a defaultPanel including a text about the description of the game. Second button is playGameButton, which leads to the levelsPanel. Third button is creditButton and it shows the developers of the game in a defaultPanel. Fourth button is quitButton and when it is clicked, game is immediately quit. playGameButton, creditButton and quitButton will be placed in the middle of the screen respectively. The slider is placed at the upper left corner of the screen. The player will be able to change the volume of the bacgground music by scrolling the volumeSlider. MainMenuPanel will provide the user interface of the first screen to appear when the game is clicked.

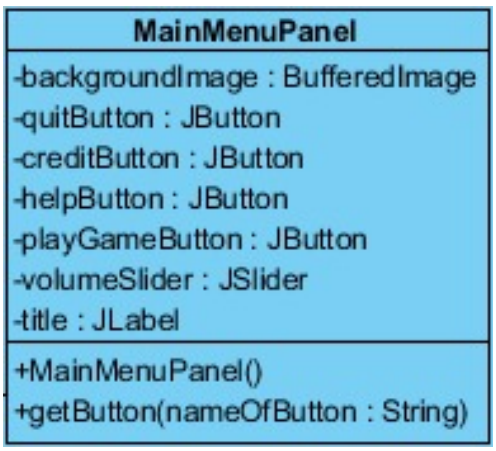

Figure 5 - Mainmenu Class

Attributes:

- **backgroundImage:** this attribute is a BufferedImage to be shown on the background of the main menu.
- **quitButton:** quitButton is one of the JButtons to be available on the main menu panel, which will be put on the bottom of the page. When it is called, the game will be quit.
- **creditButton:** creditButton is a JButton and when it is clicked, the developers of the game will be displayed on the screen.
- **helpButton:** helpButton is a JButton which will open a DefaultPanel including text of information and guidance for the game, when it is clicked.

- **playGameButton:** is a JButton. When it is clicked, LevelsPanel will appear.
- **volumeSlider:** is a JSlider to be placed at the upper left corner of the screen and will enable the player increase/decrease the volume palying on the background, which is called from MusicManager class.
- **title:** is a JLabel that includes the name of this panel.

Constructor:

- **MainMenuPanel():** MainMenuPanel() is the default constructor of this class. It takes and sets the backgroundImage; creates playGameButton, creditButton, quitButton, helpButton and volumeSlider on the page and place them on the panel, setting their dimensions and sets the related title to the panel.

Methods:

- **getButton(nameOfButton : String):** getButton takes the button of this panel as a JButton parameter, called nameOfButton. It is then used by ActionListener to understand if the button clicked by the player is the button of this panel or another panel. After ActionListener decides which panel's button is clicked by checking also the other panels' getButton methods, it proceeds the related panel.

DefaultPanel Class:

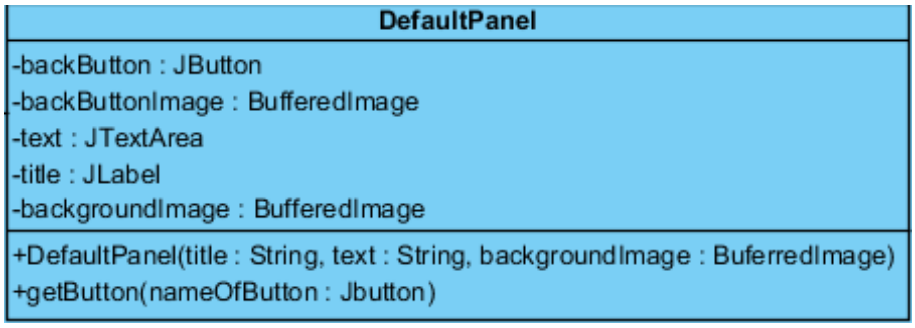

Figure 6 - DefaultPanel Class

Attributes:

- **backButton:** backButton is a JButton that will help the user go to the previous page when it is clicked.
- **backButtonImage:** is a BufferedImage. It will be placed on the backButton.
- **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.
- **backgroundImage:** is a BufferedImage. It will be placed on the backButton.

Constructor:

- **DefaultPanel(title: String, text: String, backgroundImage: BufferedImage):** this constructor creates a simple panel having a title on the page, a background image and a text. It takes the title, text and background image of the current page as its parameters, where title and text are String and backgroundImage is a BufferedImage.

Methods:

- **getButton(nameOfButton : JButton):** getButton takes the button of this panel as a JButton parameter, called nameOfButton. It is then used by ActionListener to understand if the button clicked by the player is the button of this panel or another panel. After ActionListener decides which panel's button is clicked by checking also the other panels' getButton methods, it proceeds the related panel.

LevelPanel Class:

LevelPanel is a panel having 5 buttons and title indicating it is the levels page. One of the buttons is backButton, which is to be available on every panel and returns to main menu. Other buttons correspond to 4 different levels. Each button will have an image on them showing whether that level is locked or unlocked. If the level is unlocked, the button will be

clickable, or else, it will not be clickable. At the very beginning, only the first level's button will be clickable and have unlockedButtonImage on it. Afterwards, as the new levels are completed successfully, their status of clickability will change by changeStatusOfButton method. As an unlocked level button is clicked, the game will be started by GameManager from the chosen level.

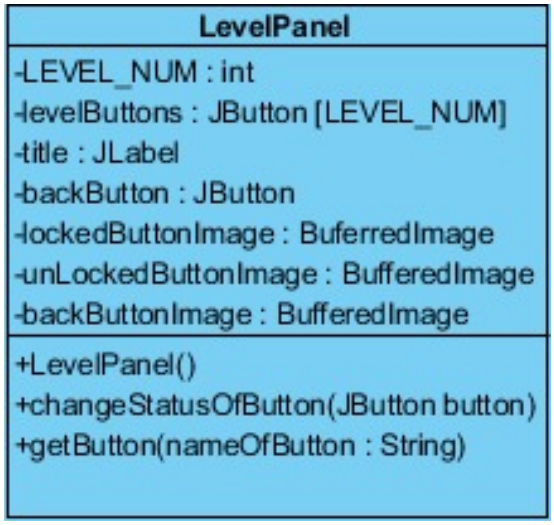

Figure 7 - LevelPanel Class

Attributes:

- **LEVEL_NUM:** is an integer for the number of levels. LEVEL_NUM will take 4 different values like 1, 2, 3 and 4, each corresponding to different levels.
- **levelButtons:** levelButtons is an array of JButton with size of LEVEL_NUM. Each button is for different levels and the game starts the level when any of these levelButtons is clicked if unlocked.
- **title:** title is a JLabel including the name of the panel.
- **backButton:** is the JButton to help the user go to the previous page.
- **backButtonImage:** is a BufferedImage. It will be placed on the backButton.
- **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(): is the** default constructor. It creates the buttons and put them on their places accordingly; sets button colors and images, sets background image and sets the title of the panel.

Methods:

- **changeStatusOfButton(JButton button):** Each level button have a status, either locked or unlocked. The unlocked levels' buttons will not clickable and have lockedButtonImage on. Therefore, if a level is finished successfully, the button of the next level will become clickable and its image will be changed to unlockedButtonImage from lockedButtonImage. changeStatusOfButton takes a JButton, called button for this method, whose current status is needed to be changed as a parameter.
- **getButton(nameOfButton : Jbutton):** getButton takes the button of this panel as a JButton parameter, called nameOfButton. It is then used by ActionListener to understand if the button clicked by the player is the button of this panel or another panel. After ActionListener decides which panel's button is clicked by checking also the other panels' getButton methods, it proceeds the related panel.

PauseGamePanel Class:

PauseGamePanel is a panel which will appear when the player clicks "P" during the game. It appears as a pop-up and includes resumeButton, returnToMainMenuButton, restartButton and a title indicating the game is paused. When resumeButton is clicked, game continues from where it is left. When returnToMainMenuButton is clicked, game finishes and MainMenuPanel is open. When the restartButton is clicked, same level restarts immediately.

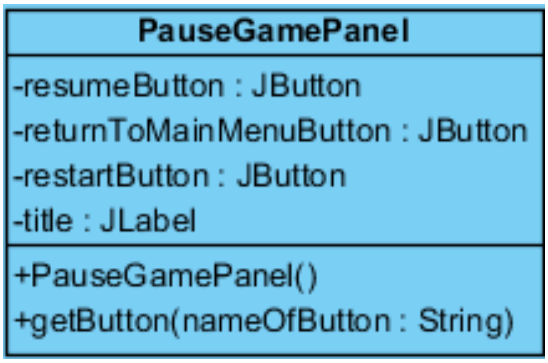

Figure 8 - PauseGame Class

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:

- **PauseGamePanel():** is the default constructor. It creates a pop-up including a title indicating the game is paused, restarts button, resumeButton and returnToMainMenuButton and sets the buttons clickable.

Methods:

- **getButton(nameOfButton: String):** getButton takes the button of this panel as a JButton parameter, called nameOfButton. It is then used by ActionListener to understand if the button clicked by the player is the button of this panel or another panel. After ActionListener decides which panel's button is clicked by checking also the other panels' getButton methods, it proceeds the related panel.

GameFinishedPanel Class:

GameFinishedPanel is a panel that displays when the level is finished successfully. It appears as a pop-up like PauseGamePanel and includes replayLevelButton, returnToMainMenu, playNextLevel buttons and a title indicating the game is finished successfully. When the button returnToMainMenu is clicked, MainMenuPanel appears. When playNextLevel button is clicked, the next level of the game starts immediately. If the user clicks replayLevelButton, the next level be unlocked, which means the success is recorded, but the same level will be started again instead of the new level

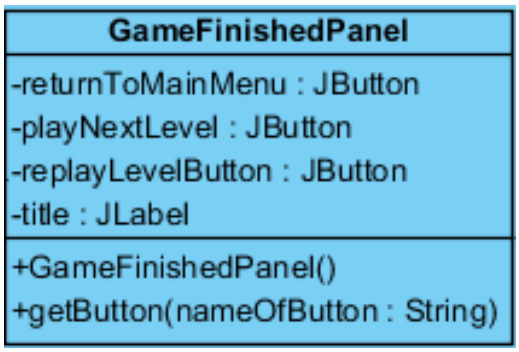

Figure 9 - GameFinishedPanel Class

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 of GameFinishedPanel class. It shows up as a pop-up on the game. It creates a title indicating the game is finished and creates replayLevelButton, returnToMainMenu and playNextLevel buttons, and set the buttons clickable.

Methods:

- **getButton(nameOfButton: String):** getButton takes the button of this panel as a JButton parameter, called nameOfButton. It is then used by ActionListener to understand if the button clicked by the player is the button of this panel or another panel. After ActionListener decides which panel's button is clicked by checking also the other panels' getButton methods, it proceeds the related action.

MusicManager Class:

During the game, music will play on the background and the MusicManager is to manage the music related part of the game. As the game clicked and MainMenuPanel is open, music starts. The same music continues as LevelsPanel, PauseGamePanel and GameFinishedPanel are open. Once the game starts, the music changes. There will be some different sound effects, which are also taken as music in the input file, that will start as the dot jumps, falls, collides etc. The player will not be able to start or stop the music, but s/he can only make it silent by changing the volume.

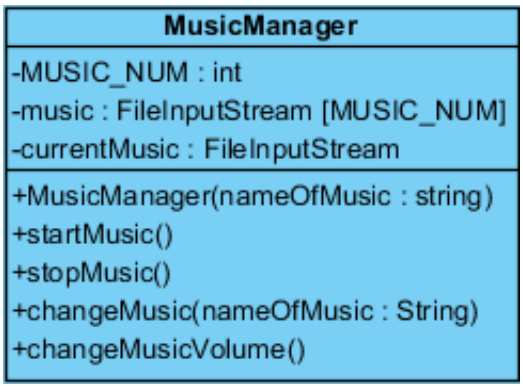

Figure 10 - MusicManager Class

Attributes:

- **MUSIC_NUM:** is an integer. MUSIC_NUM will correspond to a music to be displayed on the background of the game in the music file.
- **music:** is a JButton to pass the next level when the level is completed successfully.
- **currentMusic:** is a JButton which enables the user to play the same level again.

Constructor:

- **musicManager(nameOfMusic : String):** The constructor of MusicManager class takes the name of the music to be played on the background as a String parameter, named nameOfMusic. It sets the nameOfMusic as currentMusic.

Methods:

- **startMusic():** startMusic makes a music start to play. When the corresponding MUSIC_NUM is used in music fileInputStream array, that music starts. currentMusic will hold the currently playing music and as the music changes, currentMusic will shift to the new music.
- **stopMusic():** stopMusic method makes the currentMusic stop playing. It will be used to stop the music when a new music is to be started to play.
- **changeMusic(nameOfMusic: String):** changeMusic changes the currentMusic by starting(startMusic) a new music and stopping(stopMusic) the last one. This method is to be used to change the music as the game starts, or a new panel is opened during the game. For example, while currentMusic is music1 during the game, if the game is paused and PausePanel appear, it will change to music2. The new music which is wanted to play after changeMusic method will be taken as a String parameter holding the name of that music in the input file, which is nameOfMusic: String.
- **changeMusicVolume():** changeMusicVolume() method controls the colume of the currentMusic as the player uses the volumeSlider. If the player slides it left side, volume will decrease; else if s/he slides it to the right, volume will increase. If the user does not use volumeSlider and stop sliding, the volume will stay same where the slide is left.

MainMenu Class:

MainMenu class uses MainMenuPanel, LevelsPanel and DefaultPanel. It also uses GameManager to initiate the game. According to the flow of the game, it decides which panel is to be open, which methods to be called, which data to be saved etc.

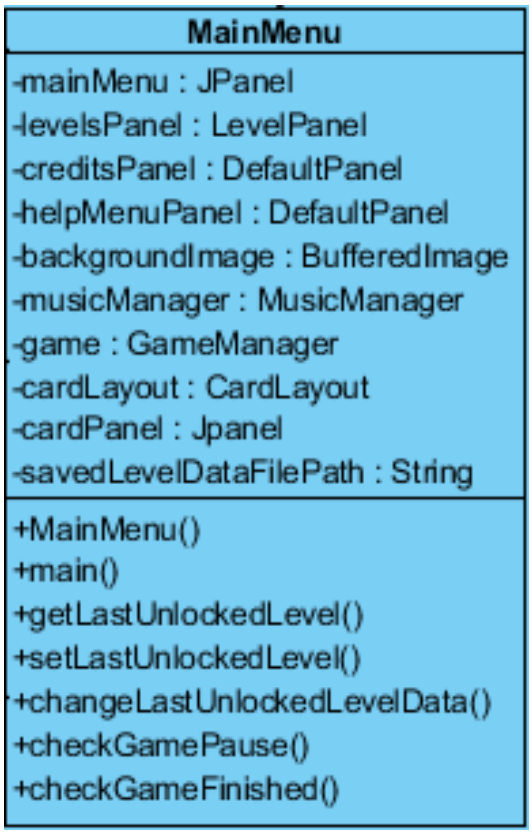

Figure 11 - MainMenu Class

Attributes:

- **mainMenu:** mainMenu is a MainMenuPanel 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**.**
- **helpMenuPanel:** helpMenuPanel is also a DefaultPanel, which will include a text of description of the game.

- **backgroundImage:** backgroundImage is a BufferedImage which will be seen on the background of the game.
- **musicManager:** musicManager is a MusicManager to control the music playing on the background of the game depending on which panel is open.
- **cardLayout:** is a CardLayout to provide the transition among all panels of the game, which are all put in one frame. It is put into cardPanel, which is a Jpanel.
- **cardPanel:** is a JPanel where all the panels (MainMenuPanel, LevelPanel, DefaultPanel etc.) are added into. It shifts the panels as their buttons are clicked, or any other necessary condition is provided to open that panel.
- **savedLevelDataFilePath:** this attribute is to hold a string for the lastly saved level's name.
- **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.
- **setLastUnLockedLevel():** this method sets the last unlocked level to keep which level the game is left and up to which level the others are unlocked.
- **getLastUnLockedLevel():** gives the last unlocked level.
- **checkGamePause():** returns true if the game is paused, and opens PauseGame panel.
- **checkGameFinished():** checks if the game is over and opens gameFinished panel.

Game Logic Subsystem

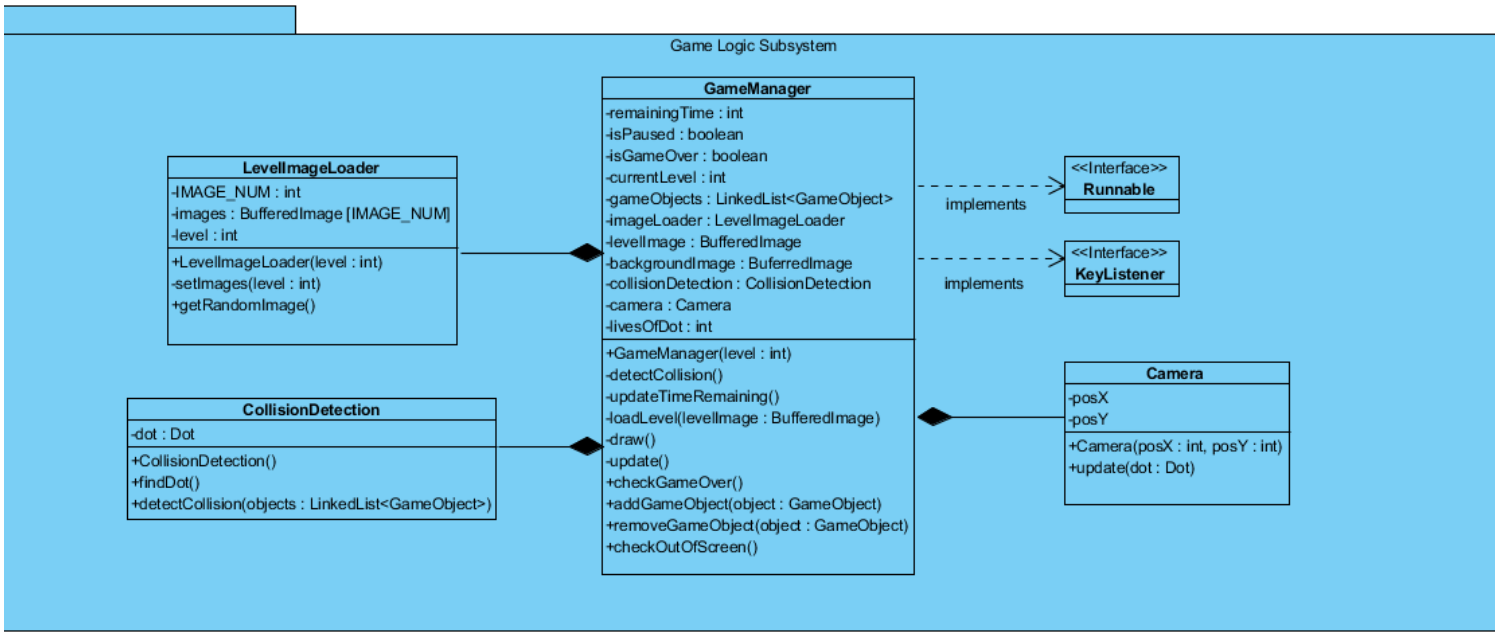

Figure 12 – Game Logic Subsystem

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. The game loop is started in this subsystem. The updating of the objects will also be done in this subsystem, more specifically in the GameManager class.

Game Manager Class

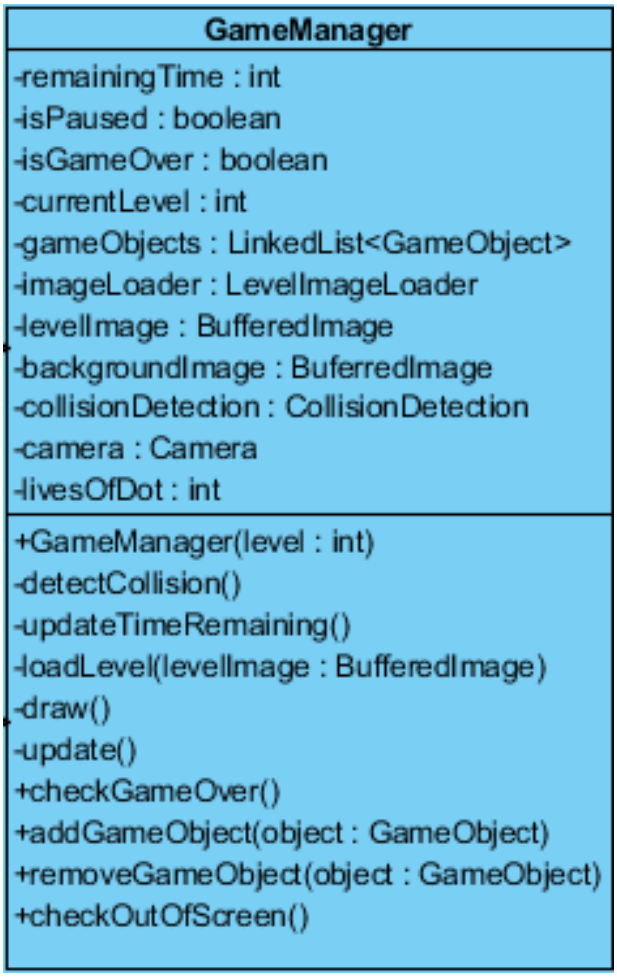

Figure 13 - GameManager Class

The GameManager class is the Façade class of the Game Logic subsystem, thus it is responsible for 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.

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 levelImage 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 GameObject, 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.

As the game continues, the game loop will be running and in the game loop the draw and update methods will be called continuously. The update method basically updates the positions of the

objects by calling the update method of each GameObject, later the draw method draws each GameObject's updated position by calling the draw method of each GameObject.

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

Attributes:

- **remainingTime:** This attribute keeps track of the remaining time. The user must finish the game in a given time, thus there must be an integer attribute which keeps track of the time.
- **isPaused:** A Boolean attribute which will keep the data of whether the game is paused or not. During the game, the user will be able to pause the game, this attribute will keep track of this information.
- **isGameOver:** This is a Boolean attribute which keeps track of whether the game is over or not. The value of this attribute is decided by checking the remaining time, the lives of the player and whether the player has reached the finish point.
- **currentLevel:** This is an integer attribute which holds the number of the level that is currently being played. This information is needed to load the game according to the level.
- **gameObjects:** This is a linked list of type GameObjects. The objects Dot, LetterBox, FadingLetterBox, Spike, Eraser, TimeBonus, TimePunishmentCircle, CheckPoint and ExtraLife will all be placed inside this linked list. A specific object will be reached by using a for loop. Keeping all of the GameObjects in a single linked list is an efficient way of keeping track of the objects.
- **imageLoader:** This is a LevelImageLoader object which will be used to load the image for a specific level. The number of the level is provided to this object to get a random image specific to that level from the LevelImageLoader class. This image will then be used to place the GameObjects according to the RBG values in the image (more explanation is given in the loadLevel method of the GameManager class).
- **levelImage:** This will be a BuferredImage which will be the image of the level. This image will be provided from the LevelImgeLoader class via the object imageLoader and 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 and will be the picture to be placed as the background picture.
- **collisionDetection:** This will be an object of the CollisionDetection class. This object will be used to call the collision() method of the CollisionDetection class in order to detect the collision between the Dot and other objects.

- **camera:** This is an object of the Camera class. This object is used so to enable the screen to move along with the Dot. So, the screen follows the Dot's movements.
- **livesOfDot:** This is the attribute which keeps track of the lives of the Dot. The Dot will initially have 3 lives. This information is given to the GameManager class to be able to show the remaining lives of the Dot in the user screen.

Constructor(s):

- **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. All of the initializations are done in the constructor. For example the initialization of the objects camera, gameObjects, imageLoader etc. are done here.

Methods:

- **detectCollision():** This method checks the collision between the Dot and the other GameObjects. In this method the detectCollision object (explained in the attributes section) is used to call the detectCollision() method of the DetectCollision class. The attribute gameObjects is passed to the method detectCollision() of the DetectCollison class, since in order to detect the collision of the Dot with the other objects, it must have the information of these objects.
- **updateTimeRemaining():** The method which will update the remainingTime attribute accordingly. In this method, the remaining minutes and seconds will be specified.
- **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 associated with that specific RGB value 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.
- **draw():** This method will draw each object in its updated location. This will be done by calling the draw() method of each GameObject. Also the backgroundImage, remaining lives of the Dot and the remaining time is drawn to the screen in this method.
- **update():** This method will update the positions of the GameObjects in the class. This will be done by calling the update method of each GameObject. Also all the other updates such as calling the updateTime(), checkGameOver() and checkOutOfScreen() methods are handled here. The update of the position of the camera is also done here, this is done by calling the update method of the Camera class.

- **checkGameOver():** This method will check whether the game is over by checking whether the time is finished, whether the lives of the Dot has finished and by checking whether the Dot has reached to the end of the sentence.
- **addGameObject(object: GameObject):** This is the method which is used to add GameObjects to the gameObjects linked list.
- **removeGameObject(object: GameObject):** This is the method which is used to remove GameObjects to the gameObjects linked list.
- **checkOutOfScreen():** This method checks whether the Dot has got out of the window limits of the screen, in other words, whether the Dot has fallen out of the screen. If the Dot falls out of the screen the life of the Dot is decreased by 1 and the Dot is placed to its starting position or to the last checkpoint that was reached.

LevelImageLoader Class

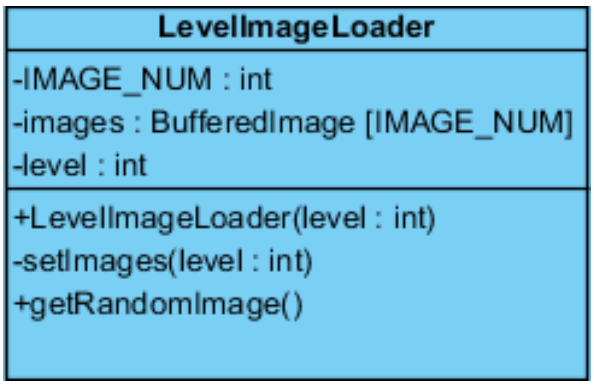

Figure 14 - LevelImageLoader Class

This class is responsible for 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. For example if this attribute is 5 then the images array will be of size 5, thus 5 images will be placed 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. The contents of the array will be filled by the setImages() method.
- **level:** This will be an int value which holds the number of the level. According to this number, a random image will be provided from the images array to the GameManager.

Constructors:

- **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 for 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

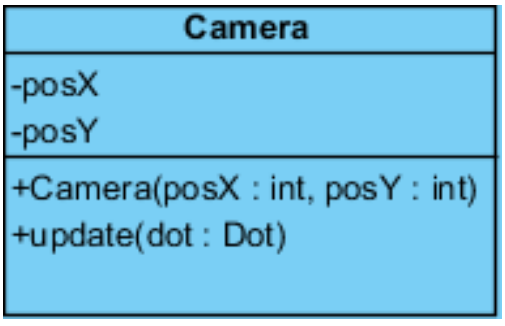

Figure 15 - Camera Class

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(posX, posY):** The constructor which initializes the 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. It needs the Dot's position since the movement of the camera depends on the Dot's position.

Collision

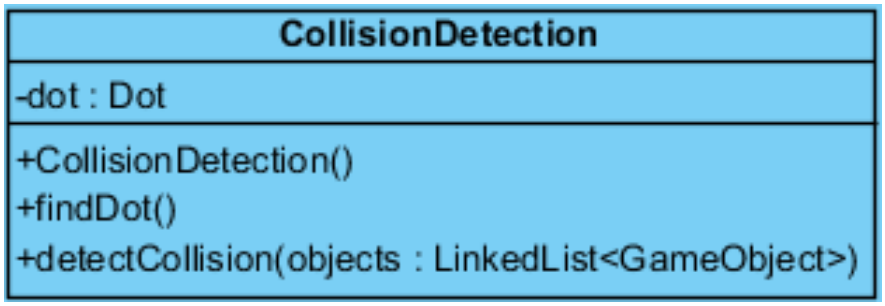

Figure 16 - CollisionDetection

This class is responsible for checking the collision between the Dot and the other objects. The detectCollision method of this class is called by the GameManager class. A Dot object is created here in order to update the position of the Dot object according to the detected GameObject. For example if the Dot collides with a Spike object then the detectCollision() method changes the position of the Dot to the starting point or to the last reached checkpoint.

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

Attributes:

- **dot:** This is an object of the Dot class. This object is created to be able to track information of the Dot object in the GameManager class. This is because once a collision is detected some changes to the Dot object must be made and in order to do this this Dot object must be created. This Dot object is the same Dot of the GameManager class because this Dot object is assigned to the Dot object from the objects linked list that is passed to the detectCollision method (so the assignment of the Dot is done inside the detectCollision method by calling the findDot method).

Constructors:

- **CollisionDetection():** This is the default constructor. This is an empty constructor which is only used when an object of CollisionDetecion is created. So it has no code inside.

Methods:

- **findDot(objects: LinkedList<GameObject>):** This method is used to assign the value of the Dot object of the CollisionDetection class. In order to detect collision and to update the position of the Dot, this class must have information of the Dot object, thus the objects linked list, which contains the Dot, is passed to this method so that the Dot could be found in this linked list and so that it would be assigned to the Dot object of this class. As a result of this, the Dot object in this class and the Dot object of the GameManager class will be the same objects.
- **detectCollision(objects: LinkedList<GameObject>):** This method is responsible from detecting the collision between the Dot object and the other GameObjects such as Spike, Eraser, LetterBox etc. In this method, first the Dot object is specified by calling the findDot() method. After this the linked list is iterated and it is checked whether the boundaries of the Dot and the other GameObejcts are intersecting. If an intersection is detected, in other words if a collision is detected, the Dot will be updated accordingly. For example, if an intersection is made with the ExtraLife object, an extra life is added to the Dot (if the Dot has the maximum number of lives, this will not be done).

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 "GameObject", "LetterBox", "FadingLetterBox", "Extra_Life", "Dot", "Eraser", "Spike", "CheckPoint", "TimePunishmentCircle" and "TimeBonus". This subsystem has also Enumaration named as "ObjectID" to indicate the types of different objects.

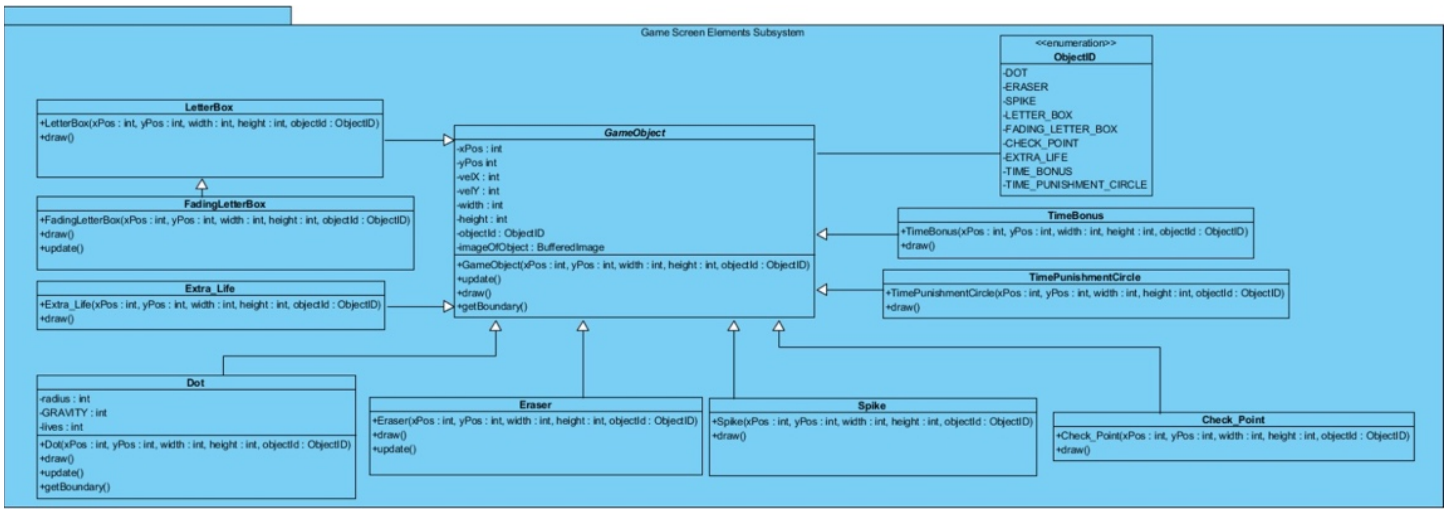

Figure 17 – Package Diagram of GameScreenElements

GameObject Class

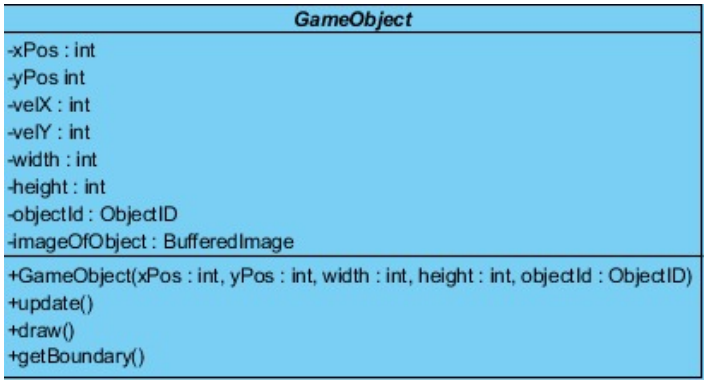

Figure 18 - GameObject Class

"GameObject" is an essential class for this subsystem. All of the objects have to keep an account of the position information, velocity information, size information, object type and the image of 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 *“getBoundary()”* since all objects may act differently to these operations. “GameObject” will be instantiated whenever the user attempts 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.
- **imageOfObject :** This will stand for the images of different object types.

Constructor:

- **GameObject(xPos : int, yPos: int, width: int, height: int, objectId: ObjectID) :** This will initialize a game object according to given position values, size information and the object type.

Methods:

- **draw() :** This is an abstract method. All objects will include their own implementations but generally it will achieve drawing the object according to its size and position information. By calling this method, each game object can be drawn in its updated location.
- **update() :** This is also an abstract method. It will manage the position of object according to its updated position information. This method will get the x and y coordinate of object’s position and replace these values with the new coordination.
- **getBoundaries() :** This is also an abstract method and will be responsible for returning the boundaries of an object according to necessary attributes such as position, width and height of different object types.

LetterBox Class

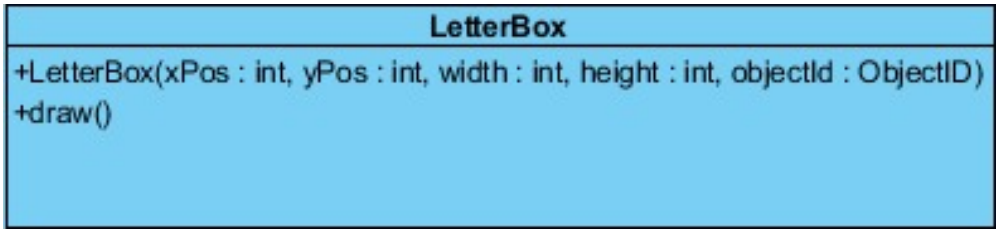

Figure 19 - LetterBox Class

The "LetterBox" class has the same attributes as the GameObject class but also includes a constructor for initializing LetterBox object and the implementation of *"draw()"* abstract method. 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 attributes of the LetterBox class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

Constructor:

- **LetterBox(xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) :** This is the constructor where the position information, size information and the object type specification will be given, thus the LetterBox will be instantiated according to "xPos", "yPos", "width", "height" and "objectId" values. This constructor will initialize a number of letters which are constructing obstacles the player will be encountered during the game.

FadingLetterBox Class

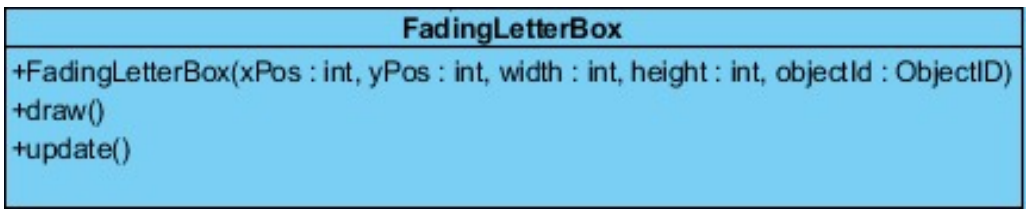

Figure 20 - FadingLetterBox Class

The “FadingLetterBox” class is a child class of “LetterBox” class and has the same attributes as its parent class but also includes the implementations of *“draw()”* and “*update()*” abstract methods. This class will be initialized after the player starts to play the level including fading letters. The FadingLetterBox object has the abilty to disappear when the collision with the Dot object is detected. Each object of this class is able to fade away in case of such a collision.

The attributes of the FadingLetterBox class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

Constructor:

- **FadingLetterBox(xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) :** This is the constructor where the position information, size information and the object type specification will be given, thus the FadingLetterBox will be instantiated according to “xPos”, “yPos”, “width”, “height” and “objectId” values.

Methods:

- **update() :** This is an abstract method of the GameObject class. It will manage the position of object according to its updated position information. This method will get the x and y coordinate of the FadingLetterBox object’s position and change these values when the collision with dot object is detected and FadingLetterBox object needs to be disappeared.

Extra_Life Class

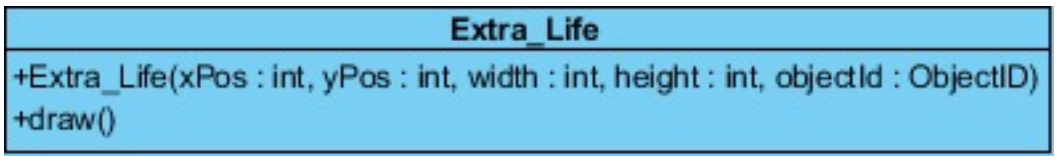

Figure 21 – Extra_Life Class

The “Extra_Life” class has the same attibutes as the GameObject class and the implementation of *“draw()”* abstract method exists in the class. The object of this class enables the player to get an extra life during the game. These “Extra_Life” objects will be created around the letters and the player can manage to collect these extra lives which give the player one more chance to play.

The attributes of the Extra_Life class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

Constructor:

- **Extra_Life (xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) :** This is the constructor where the position information, size information and the object type specification will be given, thus the extra life will be instantiated according to "xPos", "yPos", "width", "height" and "objectId" values. This constructor will create the extra lifes in specified positions where the player is able to reach and take it.

Dot Class

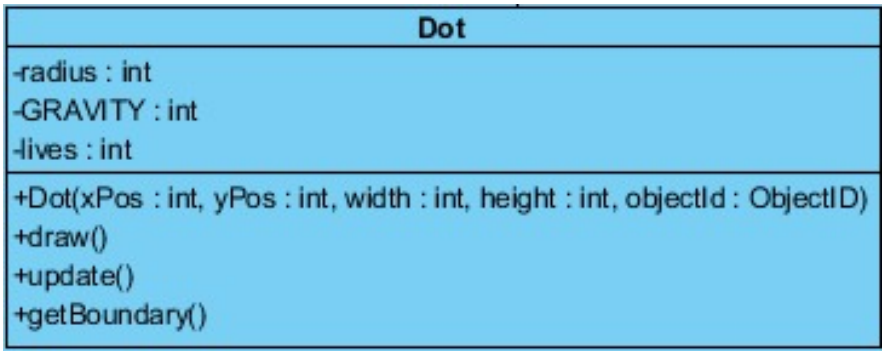

Figure 22 - Dot Class

The "Dot" class has the same attibutes as the GameObject class but also includes a radius, GRAVITY and lives as new attributes. The implementations of *"draw()"*, "*update()*" and "*getBoundary()*" 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.

The Dot class has the attributes of GameObject class which is defined before. Also, the description of abstract methods is given above. The new attributes of this class are described below to clarify the functionality of each attribute.

Attributes:

- **radius :** It represents the radius of dot object which will be drawn as a sphere. The draw method will use this attribute to specify the size of this object while drawing it.
- **GRAVITY :** This is the attribute which indicates the gravity of dot object because the dot is the only object that needs to be affected by the force of gravity while falling down. This object will jump between the letters without any collision with the ground then, it will be under influence of the gravity during the game.
- **lives :** This attribute keeps an account of the lives of the Dot. The Dot will initially have 3 lives. This information is given to the GameManager class to be able to show the remaining lives of the Dot in the user screen.

Constructor:

- **Dot (xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) :** This is the constructor where the position information, size information and the object type specification will be given, thus the dot will be instantiated according to "xPos", "yPos", "width", "height" and "objectId" values.

Methods:

- **getBoundary() :** This is an abstract method and will be responsible for returning the boundaries of the Dot object according to necessary attributes such as position, width and height of this object. Also, this method will help to check whether the dot is out of screen by calling this method and indicating the boundaries of dot object.

Eraser Class

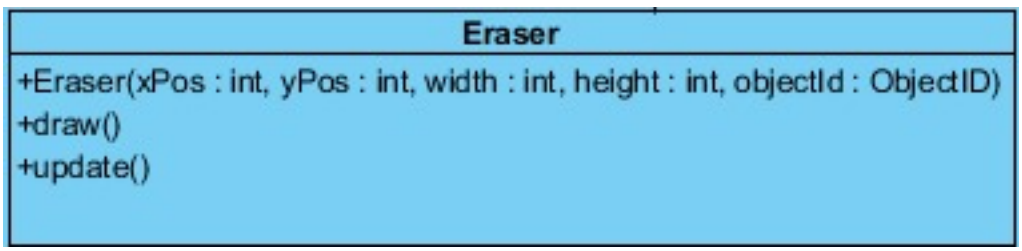

Figure 23 - Eraser Class

The “Eraser” class will be initialized when the player manages to play the level including the Eraser objects aimed at dropping 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.

The attributes of the Eraser class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

Constructor:

- **Eraser(xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) :** This is the constructor where the position information, size information and the object type specification will be given, thus the Eraser will be instantiated according to “xPos”, “yPos”, “width”, “height” and “objectId” values.

Spike Class

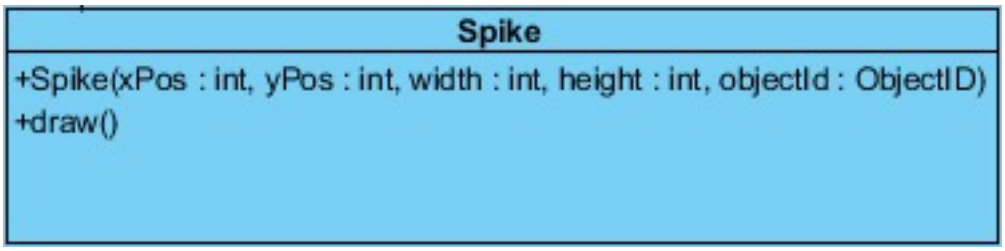

Figure 24 - Spike Class

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”.

The attributes of the Spike class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

Constructor:

- **Spike(xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) : :** This is the constructor where the position information, size information and the object type specification will be given, thus the spike will be instantiated according to “xPos”, “yPos”, “width”, “height” and “objectId” values.

CheckPoint Class

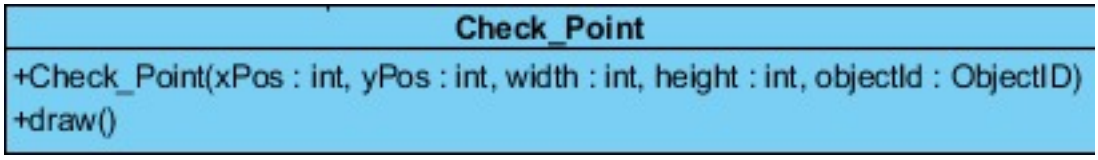

Figure 24 – CheckPoint Class

The "CheckPoint" class has the same attibutes as the GameObject class and the implementation of *"draw()"* abstract method. The objects of this class will be created in specified positions and these checkpoints can be reached by the player throughout each level. For instance, if the Dot falls out of the screen the life of the Dot is decreased by 1 and the Dot is placed to its starting position or to the last checkpoint that was reached. Hence, the checkpoint enables the player to finish each level without returning the starting point for its each fault.

The attributes of the CheckPoint class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

Constructor:

- **CheckPoint (xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) : :** This is the constructor where the position information, size information and the object type specification will be given, thus the CheckPoint will be instantiated according to "xPos", "yPos", "width", "height" and "objectId" values.

TimePunishmentCircle Class

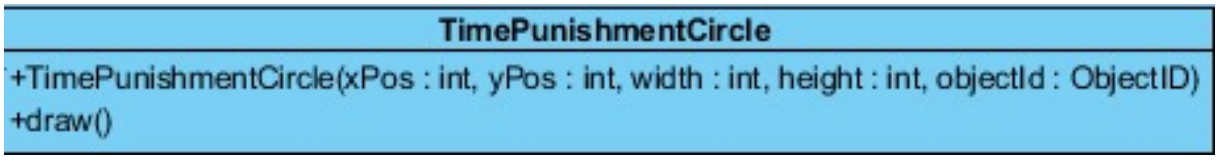

Figure 25 – TimePunishmentCircle Class

The "TimePunishmentCircle" has the same attibutes as the GameObject class and the implementation of *"draw()"* abstract method. The object of this class leads to a decrease in the

remaining time of each level if the player collides with this object. Therefore, it stands for an obstacle for the game. The dot must try not to encounter a TimePunishmentCircle object.

The attributes of the TimePunishmentCircle class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

Constructor:

- **TimePunishmentCircle (xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) :** This is the constructor where the position information, size information and the object type specification will be given, thus the TimePunishmentCircle will be instantiated according to "xPos", "yPos", "width", "height" and "objectId" values.

TimeBonus Class

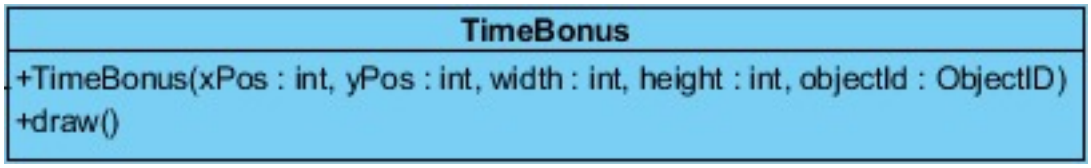

Figure 26 – TimeBonus Class

The "TimeBonus" has the same attibutes as the GameObject class and the implementation of *"draw()"* abstract method. This class provides a reverse situation for the player because TimeBonus object increases the remaining time of each level if the player encounters an object of this class.

The attributes of the TimeBonus class are the same as GameObject class which is defined before. Also, the description of abstract methods is given above.

Constructor:

- **TimeBonus(xPos : int, yPos : int, width : int, height : int, objectId : ObjectID) : :** This is the constructor where the position information, size information and the object type specification will be given, thus the TimeBonus will be instantiated according to "xPos", "yPos", "width", "height" and "objectId" values.

Enumaration (ObjectID)

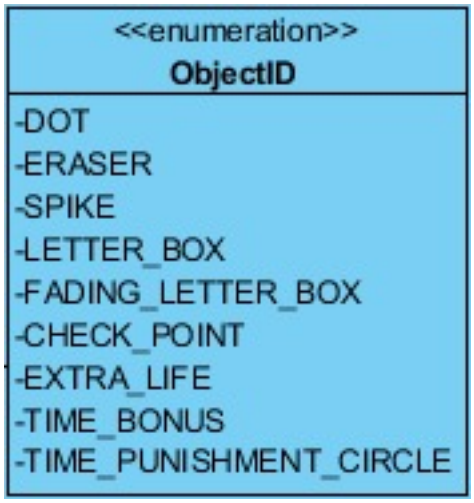

Figure 27 – ObjectID as Enum Types

"Game Screen Elements Subsytem" includes a lot of object types such as "Dot", "Eraser", "Spike", "LetterBox" and "FadingLetterBox", "CheckPoint", "ExtraLife", "TimeBonus" and "TimePunishmentCircle". In attempt to make our implementation organization easier, 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