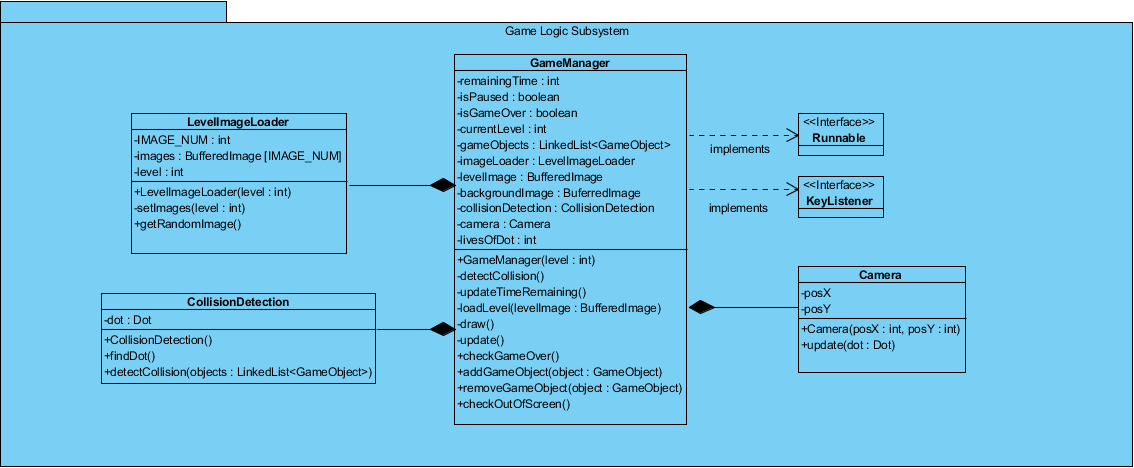
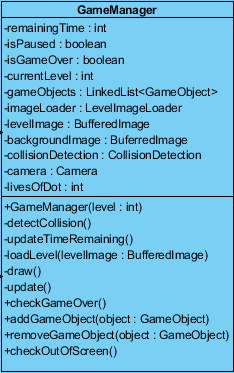
## Game Logic Subsystem



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



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):**

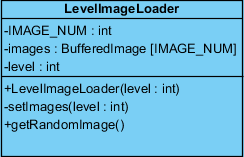
* **GameManager(level: int):** This is the constructor where the level number information will be given, thus the game will be initialized according to the level. All of the initializations are done in the constructor. For example the initialization of the objects camera, gameObjects, imageLoader etc. 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

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.

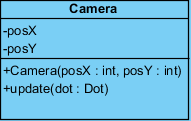
**Constructors:**

* **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 responsiblefor 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



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.

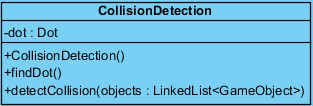
**Constructors:**

* **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



CollisionDetection Class

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).