

Bilkent University

Department of Computer Engineering

|  |
| --- |
|  |

**CS 319 - Object-Oriented Software**

**Engineering Project Report**

*Color Shooter: The Spectrum Adventurer*

System Design Report

Group 3-F

Hasan Selim Yağcı

İrem Ural

Erdem Adaçal

Alper Mehmet Özdemir

Course Instructor: Bora Güngören

Design Report

Oct 21, 2017

**Table of Contents**

**1. Introduction**

**1.1. Purpose of the system**

**1.2. Design Goals**

**1.2.1. Performance Criteria**

**1.2.2. Maintenance Criteria**

**1.2.3. End User Criteria**

**1.2.4. Trade Offs**

**2. Proposed Software Architecture**

**2.1. Overview**

**2.2. Subsystem Decomposition**

**2.3. Architectural Styles**

**2.3.1 Layers and Partitioning**

**2.3.2 Model View Controller**

**2.4. Hardware / Software Mapping**

**2.5. Persistent Data Management**

**2.6. Access Control and Security**

**2.7. Boundary Conditions**

**3. Subsystem Services**

**3.1. Design Pattern : Facade Design**

**3.2. View Subsystem**

**MainView Class**

**PauseMenuPanel Class**

**MenuListener Class**

**MainMenuPanel Class**

**CreditsPanel Class**

**HelpPanel Class**

**ChangeSettingsPanel Class**

**DifficultySelectionPanel Class**

**GamePanel Class**

**3.3 Controller Subsystem**

**GameManager Class**

**SoundManager Class**

**GameStateManager Class**

**LevelFactory Class**

**3.4 Model Subsystem**

**GameMap Class**

**GameObject Abstract Class**

**Player Class**

**Enemy Abstract Class**

**Walker Class**

**Patroller Class**

**Jumper Class**

**Floater Class**

**PowerUp Class**

**Bullet Class**

**Block Class**

**ColorId Enum Class**

**ObjectId Enum Class**

1. ***Introduction***

**1.1. *Purpose of the System***

Color Shooter: The Spectrum Adventurer is a 2D platform-shooter single player game. When other platform games are taken into consideration, Color Shooter players experience a unique experience throughout the gameplay with its high quality graphics and two special features which are color system and exclusive jump property. Although with these distinct features the complexity of the game increases, it provides an entertaining gameplay. Color Shooter aims to test player's reflexes and increment their decision making strategy while providing a user friendly environment and high quality performance.

**1.2. *Design Goals***

The following design goals are mostly established during the requirements elicitation and analysis stage in non functional requirements. However it is further explained in detail in this section considering three main criterias.

**1.2.1. *Performance Criteria***

**1.2.1.1. *Response Time***

In game industry, response time is one of the most important design criterias and in Color shooter, we aim to respond player inputs immediately. As player enters the input from keyboard or clicks on the screen, the game will adapt everything in the system accordingly immediately. GUI will be changed smoothly without players recognition.

**1.2.1.2.*Fluid Gameplay***

In most commercial games there is a target of 60 frames per second to achieve smooth games. We aim to achieve that even in low end computers.

**1.2.2. *Maintenance Criteria***

**1.2.2.1. *Extendibility***

System design should consider the extendibility of the project, especially in games new features or functionalities can be added to the system. In Color Shooter, a login system might be added in the future with appropriate database system. Moreover, additional platforms and power-ups can be defined. Hence in order to allow these changes, game design should be adaptable and easy to add new functionalities.

**1.2.2.2. *Portability***

Portability allows companies to reach several players which uses variety of platforms. Hence it is one of the significant aspects while developing a software. In order to achieve this goal, we will take advantage of Java’s portability.

**1.2.3. *End user Criteria***

**1.2.3.1. *Usability***

Color Shooter will give priority to reach several players with different age groups, therefore it is important for us to develop an easy to understand system. The user experience is important for us. The user friendly interface and several help buttons in the game will help the player to comprehend the features of the game. The smooth graphics which does not disturb eyes, and colorful images will create an user friendly interface.

**1.2.4. *Trade Offs***

**1.2.4.1. *Usability vs. Functionality***

As the game game is suitable for players with different age groups, it is important for us to develop and easy to use system. However, sometimes it can create an obstacle to add new features to the game. Hence, when we add a new functionality to our system, we guarantee that it will be a user friendly property. It will entertain the player. Although some features in our system might seem complex such as color system or jumphobia property, we make sure that they will be implemented in a user friendly way. For instance if the player does not prefer to play in jumphobia mode, they can select easy option while starting a game. Additionally, the help buttons which will explain the game features will be available all the time in the game.

**1.2.4.2. *DeliveryTime vs. Quality***

The project should be delivered at the end of December, hence as the developers we need to implement the system in a short time. Although we had a limited time, we aim to finish it on time in an organized manner with a well-planned schedule. Although some systems can be delivered with some small bugs, we hope to figure them out beforehand and deliver an error free software.

**1.2.4.3. *Memory vs. Performance***

As games are event-driven softwares, performance of the system is extremely significant, therefore we prefered performance over memory. Performance of the system can be explained as the response time of the software. As Color Shooter will include huge number of images, sounds and musics, space and performance should be considered while including them. In order to increment the response time, we will use .wav files instead of .mp3 formatted musics and sounds. The mp3 files decrement the performance of the system a lot, a certain appreciable time occurs when the game is started. However, as .mp3 files are compressed, it will allocate less memory. It is one of the examples that we prefered performance and sacrificed the memory in our software.

**2. *Proposed Software Architecture***

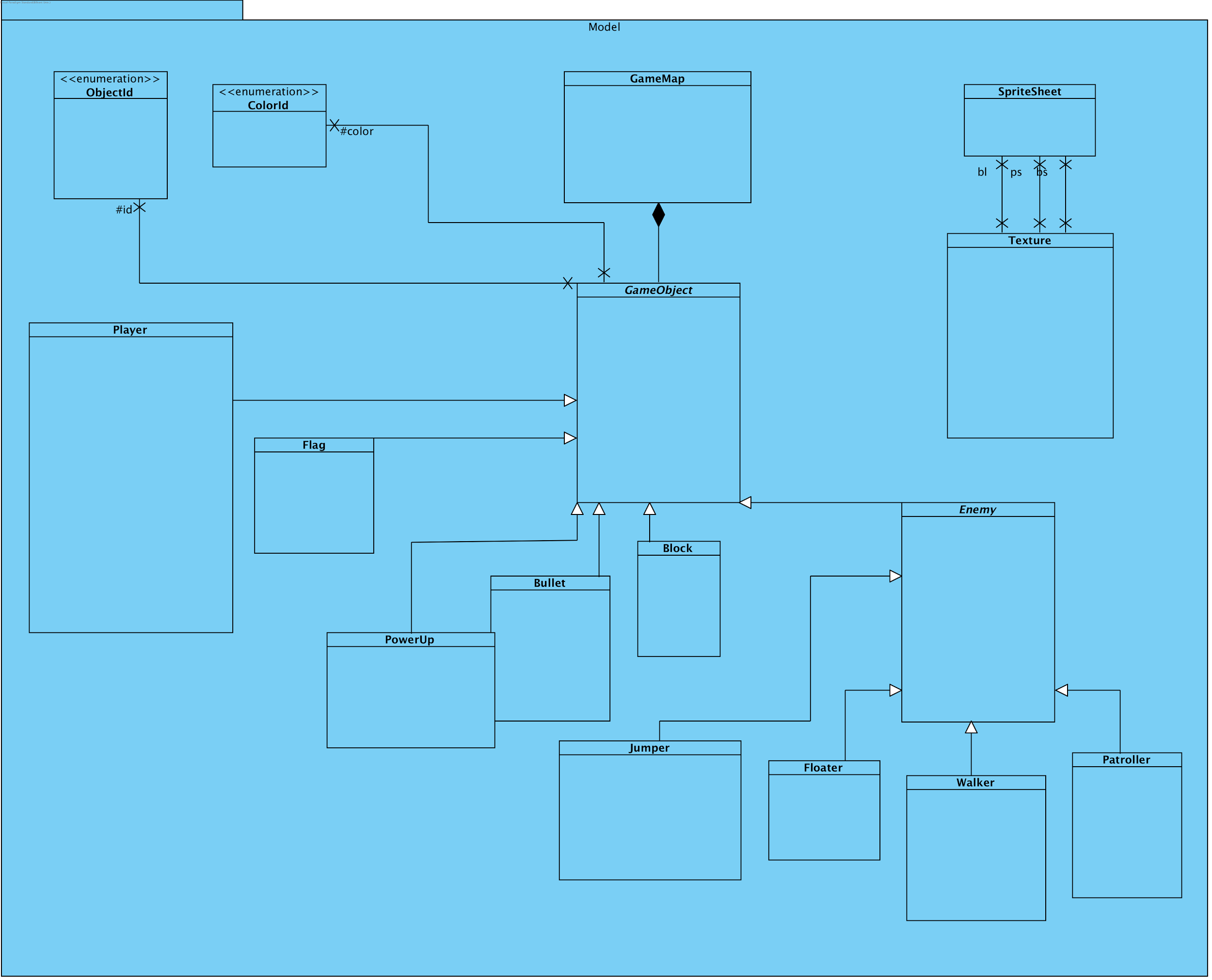
**2.1. *Overview***

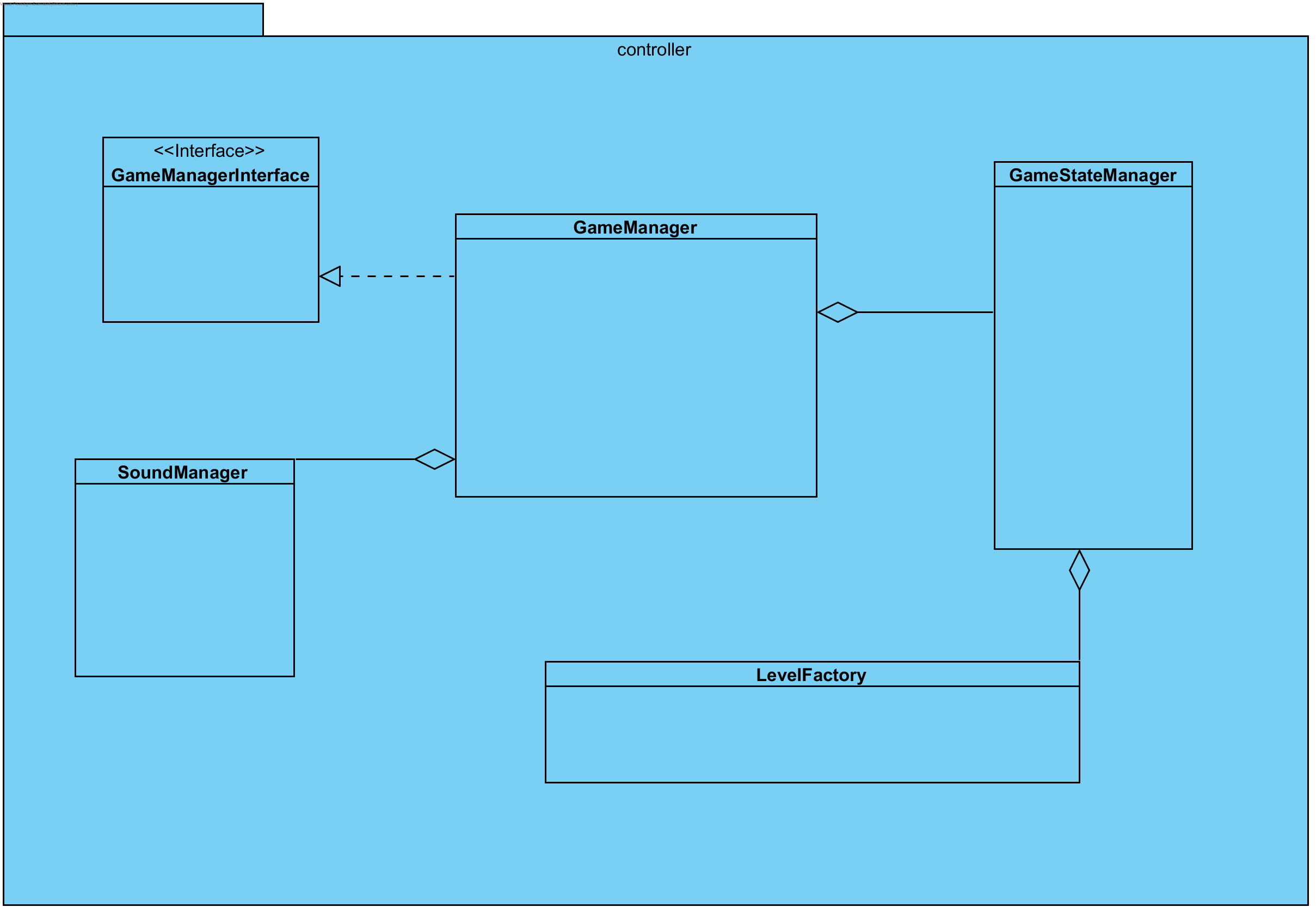
In order to overcome the complexity, the system is decomposed into manageable parts taken into consideration the functionalities defined in the requirements elicitation and design report. While decomposing the system, our main aim was to design subsystems which minimizes coupling and maximizes cohesion. By using assembly connectors(ball and socket connectors) the dependencies between subsystems is minimized. Additionally we will use MVC as the architecture style in this product.

**2.2. *Subsystem Decomposition***

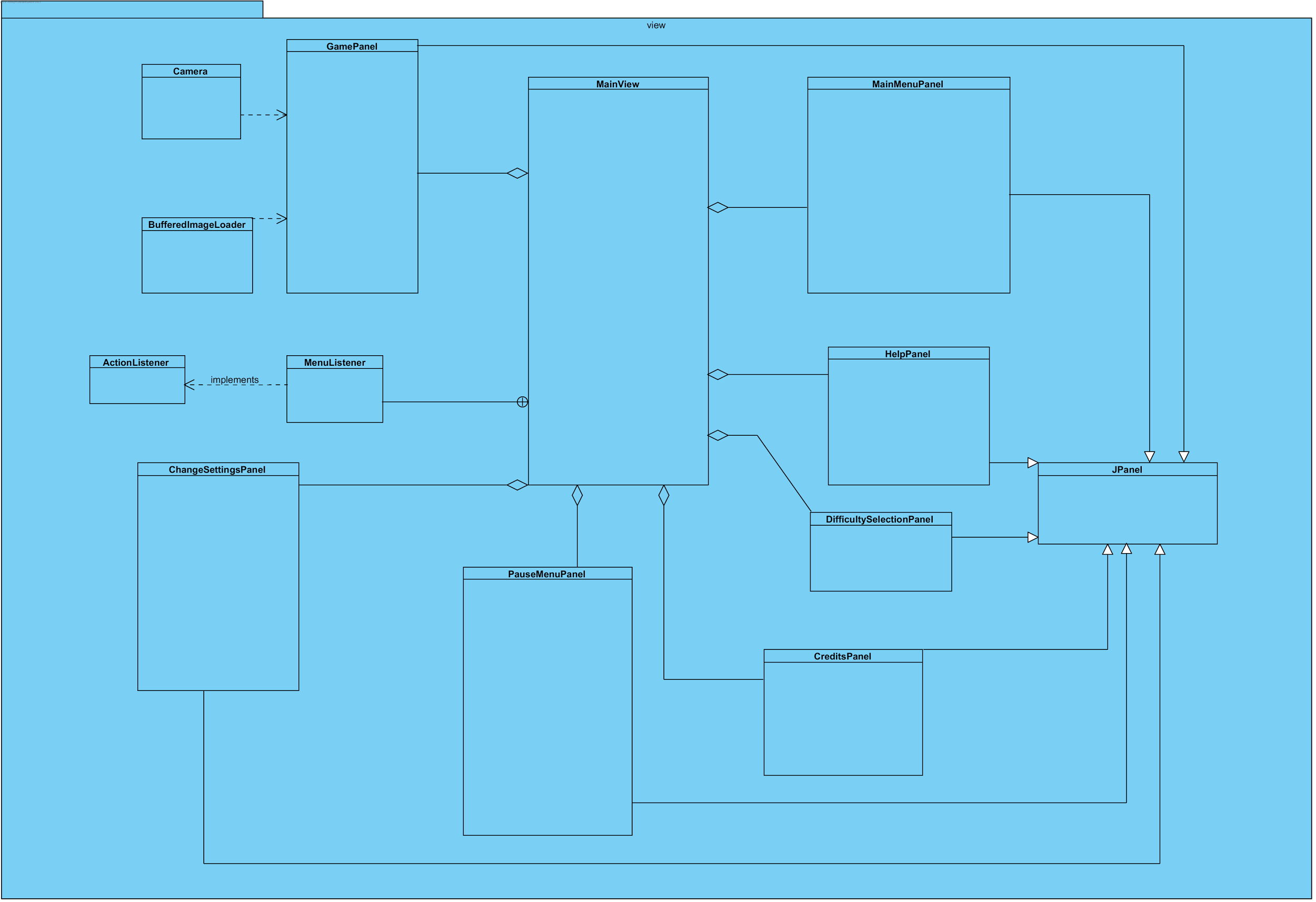
In this part, in order to overcome system complexity, our design is decomposed into several manageable parts, called subsystems. The responsibilities of each subsystem and the dependencies among them are considered in this process. Since, performance of the system, extendibility and usability of the project, which are gathered as design goals in section 1, are affected by design, we give priority to subsystem decomposition. We hope to achieve our goal by decrementing dependencies between subsystems and by gathering similar classes in the same subsystem.

Figure 1,2 and 3 illustrates the general design and the decomposition of the system. There are mainly three subsystems, Model, View and Controller. The relations/dependencies between the subsystems can be seen explained in the following. MainView class provides access to the View subsystem. GameManager class will perform several operations according to the user requests on the menu, while MenuView provides user choices and provides frame to the GameManager. GameMap class in the Model subsystem provides the graphics of each entity to the Controller subsystem. While Controller subsystem organizes each graphics according to the user input and updates and organizes the view through the GameManager class. Hence by this design coupling between subsystems is decremented and the cohesion in a subsystem is increased.

*(Figure-1: View Subsystem*)



*(Figure-2: Controller Subsystem)*



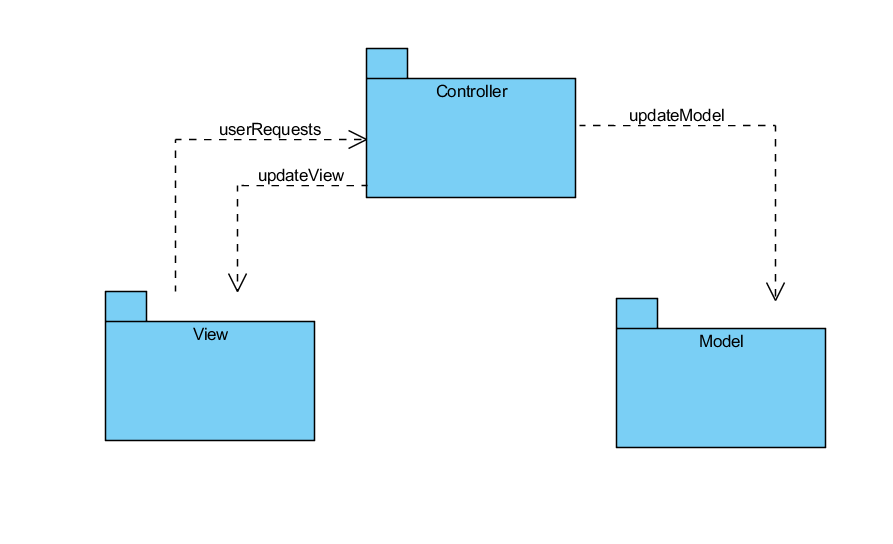
*(Figure-3: Model Subsystem)*

**2.3. *Architectural Styles***

**2.3.1 *Layers and Partitioning***

The system is decomposed hierarchically, it creates mainly two layers, on the top there is Controller and View and at the bottom there is Model. Each layer offers a special service. View provides the graphical user interface, while controller interacts with both View and Model, and manages the game dynamic. Lastly model holds the game data. The subsystem decomposition is based on closed architecture in other words opaque layering is used. Hence each layer can access only the layer below it and there is partitioning between Controller and View as each has information about each other.

**2.3.2. *Model View Controller***

The subsystems are organized according to the MVC architectural style. Model subsystem stores the data, while controller manages the interactions with the user and communicates with view and model to make necessary changes. Hence View displays user interfaces according to the changes in the Model and it receives the necessary information through the Controller. Controller initiates and reports the user action to the Model and Model makes necessary changes on the data. According to the modified data, View is updated. Each subsystem interacts mostly with one class from another class. This is designated for a particular purpose, in order to decrease coupling as stated before.

*(Figure-4: Layers )*

**2.4. *Hardware/ Software Mapping***

As Java is suitable for object oriented design and it provides portability, Color Shooter: The Spectrum Adventurer will be implemented on Java’s latest version. The hardware requirements for the game are kept as minimal. A computer with its operating system and Java installed, keyboard and mouse will be enough to play the game. Java is needed to run and compile the game, .java file. Keyboard is used to get player commands, left and right arrows for movement, up arrow for jump. Z, X and C will be used for color shifting, and spacebar will be used for shooting and etc. Mouse provides the player inputs in the menu views. Additionally, to store map and to detect in which level the player has left the game we will use txt files. Hence there should be a readable and writable text editor on the computer. As we did not provide a multiplayer mode in the game, the game does not require an internet connection.

**2.5. *Persistent data management***

The persistent data in the game will be stored in a low level storage by using flat files, Since the game system does not require a huge bunch of data, we did not prefer a database system. The maps of the game will be created and stored beforehand in .png file, and while the execution of the game, the level design will be structured according the contents of this file. Hence they are one of the important persistent data in the game. Secondly, images and sounds will be used as well, their format will be kept simple to minimize the performance issues in the game like .gif and .wav and they will be stored in the hard disk. Additionally, game allows player to continue the game. It is kept simple in the game as well, player will continue to the game from the level they left, however they will start from the beginning of the level with full health. Therefore, we will basically store the last unlocked level number in a .txt file.

Low level data management enables speed optimization, however if the system crashes the files might be lost. Besides the design will consider the concurrent accesses as well and will try to prevent these cases as well.

**2.6. *Access control and security***

Color Shooter will not store any player information and it will not require a network connection. Everyone can play the game in the same way with same functionalities and options, hence the system does not require any access control mechanism. Since the game is played individually through a single computer, it will not create any network connection and security issues. As stated in section 2.3 there won’t be any database system, hence through storage, we won’t deal with any security issues as well.

**2.7. *Boundary Conditions***

***Initialize the system***

Game will be an executable file, hence when the player opens it, it will automatically start the system. The Game Manager in the controller runs the main program and it will initiate the view accordingly and display a menu on the screen.

***Shut down the system***

Game can be terminated by two events. Firstly, from the main menu user can terminate the game.Secondly,as the view will fit into the screen, there will be a “x” button on the right corner of the view and player can exit the game anytime when they click to “x”. If player wants to quit during the gameplay, he/she has two options to quit. They can go back to the main menu from pause menu or they can click to “x” button on the right corner. Besides, unlocked level will be saved automatically in both cases.

***Exceptions***

If the system could not load the resources specified in section 2.4, it will continue to execute the program without them and display a meaningful error message.

If hardware failures occur, such as hard disk crashes, the saved data will be lost. Hence player should start the game by clicking new game in this case.

**3. *Subsystem Services***

Subsystem services ensured by each layer will be explained in detail. First the design pattern used in the system will be explained and then each class in the system, their attributes and methods will be stated.

**3.1. *Design Pattern: Facade Design***

Facade Design pattern allows programmers to design a system with low coupling. It is one of the most popular design patterns to solve complexity of the system by diminishing dependencies between systems. This pattern aims to hide the complexity of each subsystem. Therefore a class is defined to access them with a more simpler way. These classes provide an interface to access the data that is hidden. Hence, with this approach the complexity is alleviated as the communication is supplied by a single class.

In our system, there are three classes in each subsystem which can be declared as Facade classes. MainView class and GameManager classes corresponds to the Facade classes of the View and Controller. GameManager runs the main game loop and it provides the information about game dynamics and performs and informs other classes according to the User inputs.

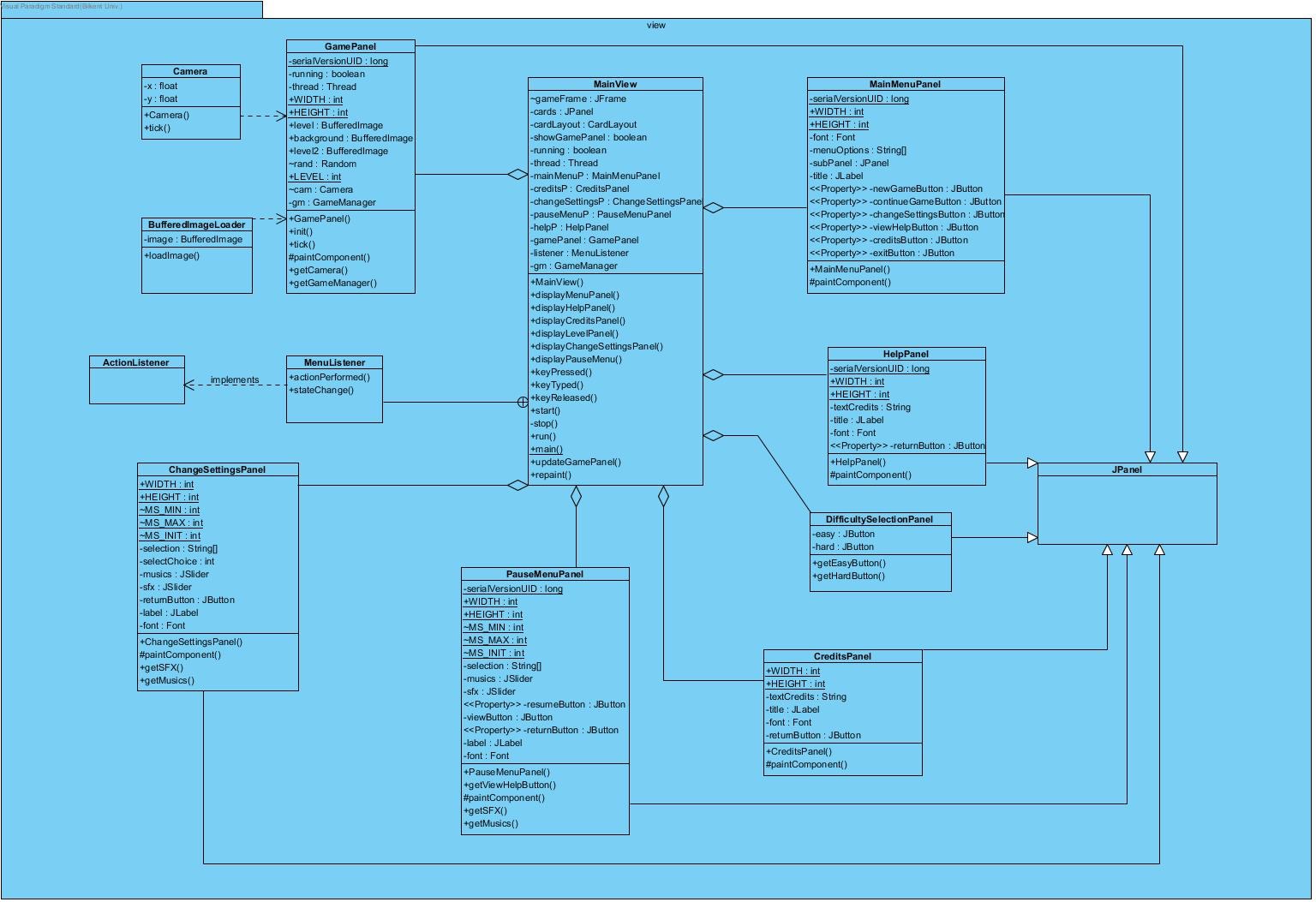
***3.1.1.Strategy Pattern***

For view the controller provides the strategy. Controller takes the user actions from the View and decides what to do next, and might update the view. View, in this project mainly MainViewModel delegates to the controller GameManager about decision making strategies. By this pattern view is decoupled from the model as well, as model get updates from only the controller. Therefore, GameManagerInterface allows us to separate view from controller. By using the same view subsystem, if controller implements the GameManagerInterface, same view can be used again.

***3.1.2.Factory Pattern***

Factory pattern is used in LevelFactory class. It creates level objects for current level from the image files. It provides extendibility in project, for instance if there are several views it can both use the LevelFactory class. Hence, encapsulating the creation helps to reuse it for future improvements and modifications.

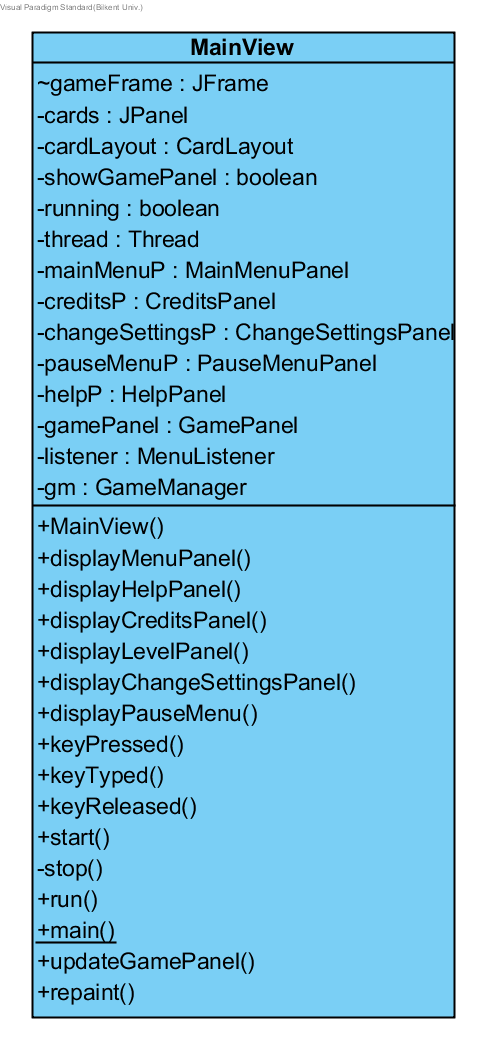
***3.2. View Subsystem***

******

*(Figure- 5: View Subsystem)*

The graphics of the game is separated from the game dynamics and logic through view subsystem. According to Facade design, MainView allows to access the view through controller. Frame and panels of the game is instantiated in this subsystem and the menu panels are totally separated from the controller and model.

**MainView class:**



**Attributes:**

* **private MenuListener listener:** ActionListener is used in order to get player inputs and to perform necessary changes, such as changing the panel.
* **private JFrame frame:** It is the main frame of the game.It holds the cards all panels inside.
* **private GameManager gm:** GameManager is instantiated in order to send user interactions on panels to the game manager. Hence, it performs necessary actions acccording to the calls made by MainMenuView.
* **private HelpPanel helpPanel:** It is used in order to display the help panel on the screen to the player. It is a JPanel and it includes labels and a back button.
* **private CreditsPanel creditsPanel:** CreditsPanel is a JPanel. It is used to view user interface of credits. There is a single label which describes the project and the developers and a back button to return to the main menu.
* **private ChangeSettingsPanel changeSettingsPanel:** ChangeSettingsPanel is a JPanel. It allows users to change music and sound effects. There is an additional return button which returns to the main menu view.
* **private PauseMenuPanel pauseMenuPanel:** pauseManuPanel is an instance of PauseMenuPanel, it is a JPanel. It will have two Jsliders for music and sound, additionally it contains resume and back buttons and returnToMainMenu button.
* **private MainMenuPanel mainMenuPanel:** MainMenuPanel is a JPanel. It has a background image, and six buttons with labels on it.
* **private GamePanel gamePanel:** GamePanel is a JPanel and it contains the gameplay view.
* **private DifficultySelectionPanel difficultySelectionPanel:** DifficultySelectionPanel is a JPanel. It has two buttons easy and hard.
* **private JPanel cards:** It is a JPanel which holds all other JPanels, which are gamePanel, mainMenuPanel, pauseMenuPanel, changeSettingsPanel, helpPanel, creditsPanel, difficultyPanel.
* **private CardLayout layout:** It is a layout, which allows to show selected panel. It is added on JPanel named cards.
* **private boolean showGamePanel:** To activate keyListener methods, a boolean is hold, When it is true, the methodcalls inside of the keyReleased() and keyPressed() methods can be called,
* **private boolean running:** It is used for the game loop, while it is true, thread starts.
* **private Thread thread:** Thread is used for the game threads.

**Constructor:**

* **public MainView():** Initializes gamePanel with new statement and calls the init method of gamePanel. Each panels on the attributes are initialized with new statement and for each view keyListener is added. From each view getButtons() methods are called and added MenuListener on these. For sfx and music sliders in ChangeSettingsPanel and PauseMenuPanel, added ChangeListener by passing MenuListener listener inside of the addChangeListener(listener) method. GameFrame is initialized, pack(), setDefaultCloseOperation(), setResizable() with false parameter, setFocusable(true), setLocationRelativeTo(null), setVisible(true) methods of JFrame are called . ShowGamePanel boolean is set to false. Running boolean is set to false.

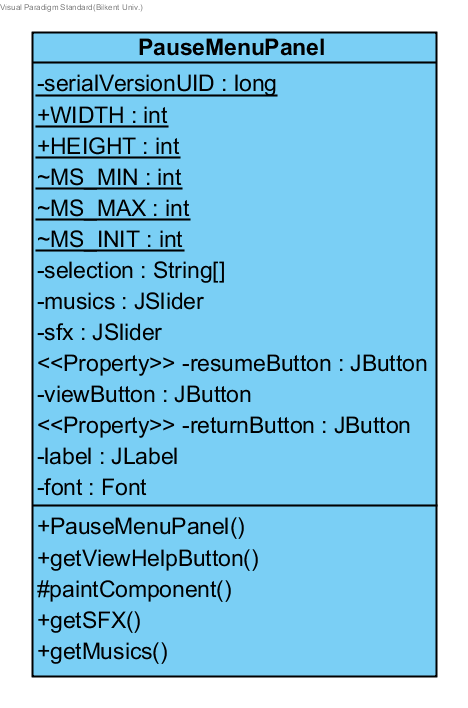
**Methods:**

* **public void displayMenuPanel():** It calls cardLayout.show(cards, "1") which displays MenuPanel and showGamePanel is set to false.
* **public void displayHelpPanel() :** It calls cardLayout.show(cards, "5") which displays HelpPanel and showGamePanel is set to false.
* **public void displayCreditsPanel():** It calls cardLayout.show(cards, "2") which displays CreditsPanel and showGamePanel is set to false.
* **public void displayLevelPanel() :** It calls cardLayout.show(cards, "6") which displays gamePanel and showGamePanel is set to true.
* **public void displayChangeSettingsPanel() :**It calls cardLayout.show(cards, "3") which displays changeSettingsPanel and showGamePanel is set to false.
* **public void displayPauseMenu() :** It calls cardLayout.show(cards, "4") which displays pauseMenu and showGamePanel is set to false.
* **public void displayDifficultySelectionPanel():** It calls cardLayout.show(cards, "7") which displays gamePanel and showGamePanel is set to false.
* **public void keyTyped(KeyEvent e) :** This method is empty, it is written in order to satisfy the conditions for KeyListener.
* **public void keyReleased(KeyEvent e) :** It gets keyCode in a integer and in if statements checks if it is right,left,up arrow keys and according to the key code calls stopPlayer(int) of gameManager, with appropriate integer values.

(integer values 0 for Left, 1 for Right, 2 for Up Keys)

* **public void keyPressed(KeyEvent e) :** It gets keyCode in a integer and in if statements checks if it is space, z, x, c, p or right,left,up arrow keys and according to the key code calls gm.updateLevelView(int) method of gameManager, with appropriate integer values. If the keys are z,x,c then calls the changeColor() method of gameManager gm.changeColor(int i) 0 for z, 1 for x, 2 for c. Otherwise calls gm.updateLevelView(int). (integer values for updateLevelView 0 for Left, 1 for Right, 2 for Up Keys, 3 for Bullets)
* **public synchronized void start() :** It makes running boolean true and creates a new Thread with new statement and calls thread.start() method.
* **public void run() :** It calculates the time and according to that computes fps and number of ticks in the game in a while loop which loops when running is true ( when game is running).
* **public static void main(String[args]) :** It creates a MainView object with new statement and calls view.start() method.

**PauseMenuPanel class:**

****

**Attributes:**

* **private static final long serialVersionUID**
* **public static int WIDTH, HEIGHT :** width and height of the panel.
* **private static final int MS\_MIN :** minimum number on JSlider.
* **private static final int MS\_MAX :**maximum number on JSlider.
* **private static final int MS\_INIT :** initiale value of JSlider.
* **private String[] selection:** It contains the strings for button and two texts, which are return to main menu, sfx and music.
* **private JSlider musics**
* **private JSlider sfx**
* **private JButton resumeButton**
* **private JButton viewButton**
* **private JButton returnButton**
* **private Font font**

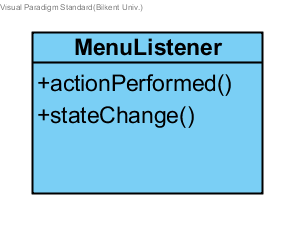
**Constructor:**

* **public PauseMenuPanel():** Firstly initializes the musics JSlider with JSlider.HORIZONTAL, MS\_MIN, MS\_MAX, MS\_INIT values. Then calls setMajorTickSpacing(5), setPaintTicks(true), setBackground(Color.BLACK) of JSlider. It is repeates for sfx JSlider as well. Return button is instantiated and its background color is set to Color.Black.Then these are added to the panel by calling add(JComponent) method of JPanel. Lastly, JPanels setPrefferedSize(WIDTH,HEIGHT), setFocuable(true), requestFocus() methods are called.

**Methods:**

* **public JButton getReturnButton()**
* **public JButton getResumeButton()**
* **public JSlider getsfx()**
* **public JSlider getMusics()**
* **public JButton getViewHelpButton()**
* **protected void paintComponent(Graphics g)**

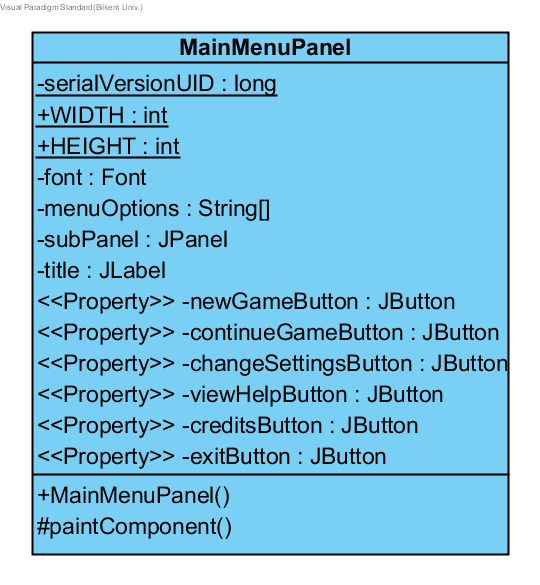
**MenuListener Class:**

****

It is a private class in MainView class and it implements ActionListener and ChangeListener interfaces.

* **public void actionPerformed(ActionEvent e):** It implements the actionPerformed method. This method handles the actions performed when button is clicked. There are several if statements and in each one checks which button is clicked. For instance, if mainMenuPanels new game button is clicked it calls the gm.changeView(0). Hence for each button it identifies it and calls GameManagers changeView(int) method. If exit button on mainMenuPanel is clicked it calls System.exit(0).
* **public void stateChanged(ChangeEvent e) :** It is implemented for ChangeListener which is required for the JSliders on the ChangeSettingsPanel and PauseMenuPanel for sfx and music. It gets source by calling Jslider e.getSource than checks in if statements which source it is (sfx or music). Then calls gm’s changeSound(int i) or changeMusic(int i) where i is the volume rate, values between 0-10.

**MainMenuPanel Class:**

****

**Attributes:**

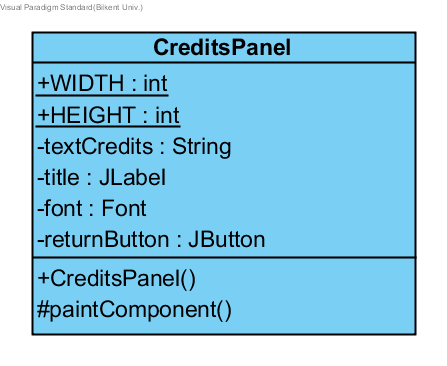
* **private static final long serialVersionUID**
* **public static int WIDTH ,HEIGHT:** width and height of the panel.
* **private Font font**
* **private String[] menuOptions:** It contains the strings on each button, these are NewGame, ContinueGame, ChangeSettings, View Help, Credits, Exit.
* **private JButton newGameButton**
* **private JButton continueGameButton**
* **private JButton changeSettingsButton**
* **private JButton viewHelpButton**
* **private JButton creditsButton**
* **private JButton exitButton**

**Constructor:** Creates each button with new statement, calls setBackground(Color) methods for each of the buttons and sets their opaqueness with setOpaque(true). Then adds these on the panel and calls standard methods of JPanel setPreferredSize(WIDTH, HEIGHT), setBackground(Color.WHITE), setFocusable(true) and lastly requestFocus().

**Methods:**

* **public JButton getNewGameButton()**
* **public JButton getContinueGameButton()**
* **public JButton getViewHelpButton()**
* **public JButton getChangeSettingsButton()**
* **public JButton getCreditsButton()**
* **public JButton getExitButton()**
* **protected void paintComponent(Graphics g):** JPanels method, calls super.paintComponent(g) and makes each button visible and sets its location on the panel calling setLocation(x,y). and in a try catch close loads an ImageIO. from resources and draws it as background Image. By calling g.drawImage(image,0,0,null).

**CreditsPanel Class:**

****

**Attributes:**

* **private static final long serialVersionUID**
* **private JButton returnButton**

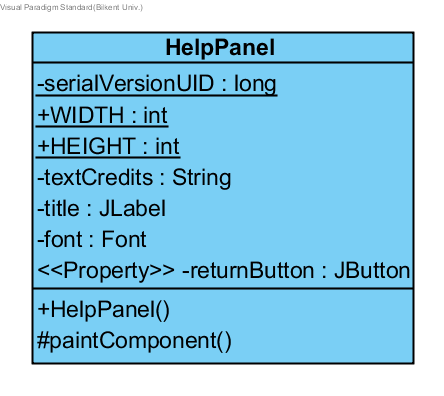
**Constructor:**

* **public CreditsPanel():** Instantiates returnButton.Then calls setBackground(Color) methods for the button and sets its opaqueness with setOpaque(true). Then adds iton the panel and calls standard methods of JPanel setPreferredSize(WIDTH, HEIGHT), setBackground(Color.WHITE), setFocusable(true) and lastly requestFocus().

**Methods:**

* **public JButton getReturnButton()**
* **protected void paintComponent(Graphics g):** It sets returnButtons visibility and location on the panel. Then loads background image in the same way with MainMenuPanel then l drawsStrings by calling g.drawString(String).

**HelpPanel Class:**

****

**Attributes:**

* **private static final long serialVersionUID**
* **public static int WIDTH , HEIGHT**
* **private JButton returnButton**

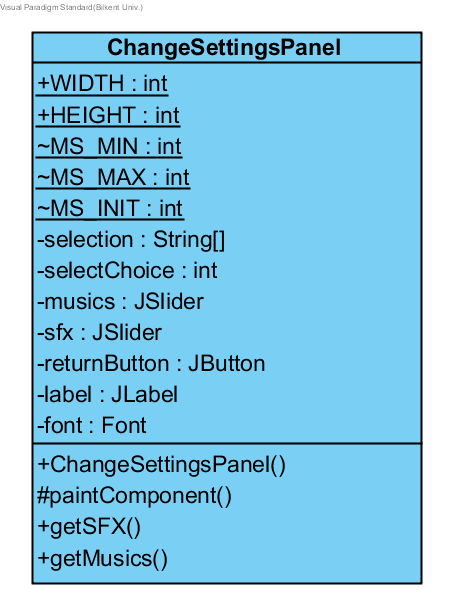
**Constructor:**

* **public HelpPanel():** Instantiates returnButton.Then calls setBackground(Color) methods for the button and sets its opaqueness with setOpaque(true). Then adds iton the panel and calls standard methods of JPanel setPreferredSize(WIDTH, HEIGHT), setBackground(Color.WHITE), setFocusable(true) and lastly requestFocus().

**Methods:**

* **protected void paintComponent(Graphics g):** It sets returnButtons visibility and location on the panel. Then loads background image in the same way with MainMenuPanel then draws three rectangles on the panel by calling g.drawRectangle(x,y,width,height) and each panel drawsStrings by calling g.drawString(String). Then at the rightmost rectangle draws 5 images for each power up. These are in a try catch close as it loads images from resources.

**ChangeSettingsPanel Class:**

****

**Attibutes:**

* **private static final long serialVersionUID**
* **public static int WIDTH , HEIGHT:** width and height of the panel.
* **static final int MS\_MIN**
* **static final int MS\_MAX**
* **static final int MS\_INIT**
* **private String[] selection** It contains the strings for one button, Return to Main Menu and two texts Music, SFX.
* **private JSlider musics;**
* **private JSlider sfx;**
* **private JButton returnButton**

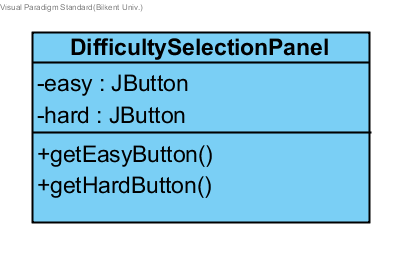
**Constructor:**

* **public ChangeSettingsPanel():** Firstly initializes the musics JSlider with JSlider.HORIZONTAL, MS\_MIN, MS\_MAX, MS\_INIT values. Then calls setMajorTickSpacing(5), setPaintTicks(true), setBackground(Color.BLACK) of JSlider. It is repeated for sfx JSlider as well. Return button is instantiated and its background color is set to Color.Black.Then these are added to the panel by calling add(JComponent) method of JPanel. Lastly, JPanels setPrefferedSize(WIDTH,HEIGHT), setFocuable(true), requestFocus() methods are called.

**Methods:**

* **public JButton getReturnButton()**
* **public JSlider getSFX()**
* **public JSlider getMusics()**
* **protected void paintComponent(Graphics g)**

**DifficultySelectionPanel Class:**

****

**Attibutes:**

* **private static final long serialVersionUID**
* **private JButton easy:**
* **private JButton hard:**

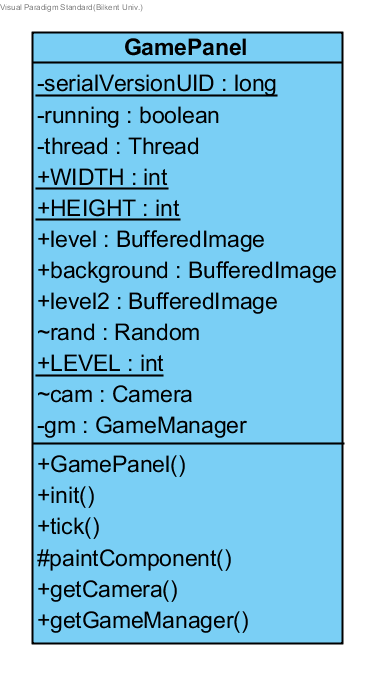
**Constructor:**

* **public DifficultySelectionPanel():** Instantiates easy and hard buttons.Then calls setBackground(Color) methods for each of the buttons and sets their opaqueness with setOpaque(true). Then adds these on the panel and calls standard methods of JPanel setPreferredSize(WIDTH, HEIGHT), setBackground(Color.WHITE), setFocusable(true) and lastly requestFocus().

**Methods:**

* **public void getEasyButton():**
* **public void getHardButton():**
* **protected void paintComponent(Graphics g)**

**GamePanel Class:**

****

**Constructor:**

* **public GamePanel(MainMenuView view):** Calls standard methods of JPanel setPreferredSize(WIDTH, HEIGHT), setBackground(Color.WHITE), setFocusable(true) and lastly requestFocus().

**Attributes:**

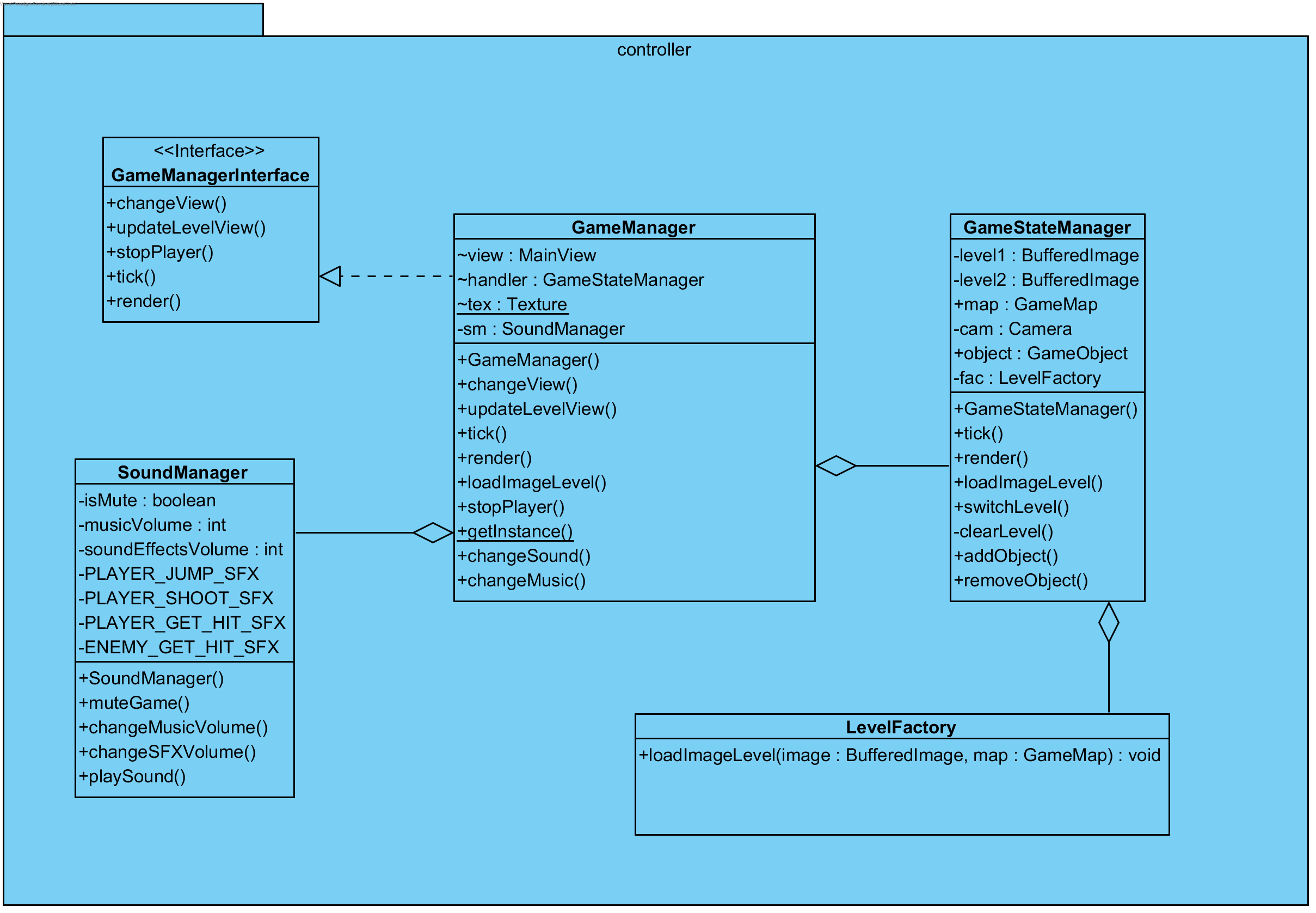
* **private static final long serialVersionUID**
* **private GameManager gm**
* **static Texture tex**
* **public static int WIDTH ,HEIGHT:** width and height of the panel.

**Methods:**

* **public void init():** Loads the level and background image and create a Camera instance.
* **public void tick():** Updates all objects.
* **protected void paintComponent(Graphics g):**  First draws the background image and then invokes ‘render()’ method of GameManager.

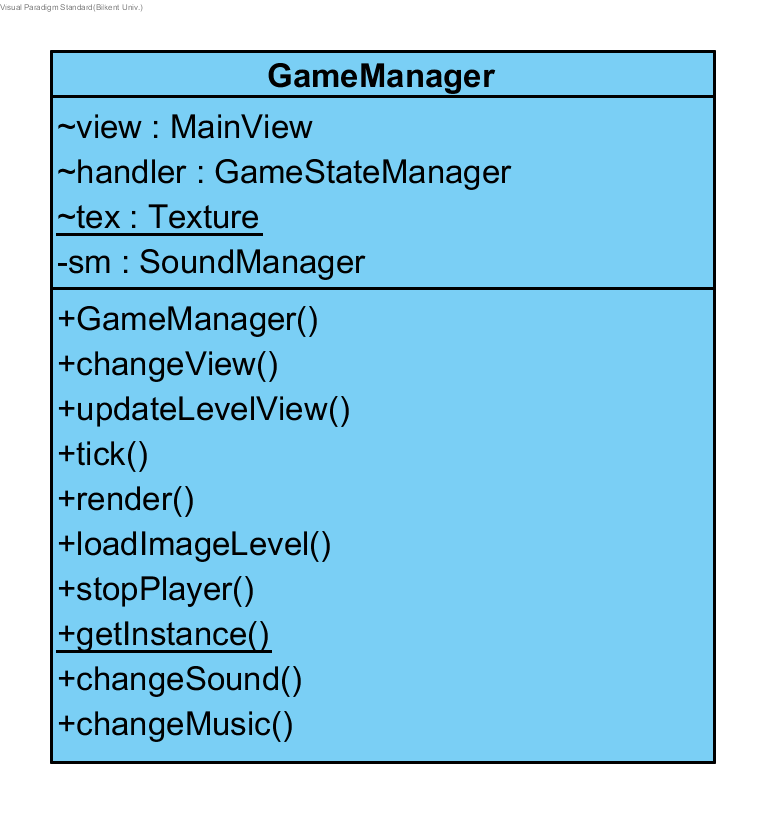
***3.3. Controller Subsystem***

In this subsystem our controller classes are are grouped together to manage the game dynamics and game logic. We have 5 components in this subsystem. Four of them are controller classes and one of them is an interface. As illustrated in Figure-5, we have the GameManagerInterface interface, GameManager, SoundManager, GameStateManager classes and LevelFactory class. These classes will be further explained below.



*(Figure-6: Controller Subsystem)*

**GameManager Class:**

****

**Attributes:**

* **private boolean difficultyIsHard:** Holds a boolean value that is true if the game difficulty is set to hard and false if the game difficulty is set to normal.
* **private boolean paused:** Holds a boolean value that is true if the game is paused, false if otherwise
* **private GameStateManager gsm:** Holds a GameStateManager object to deal with the current game state.
* **private SoundManager soundmng:** Holds a SoundManager object to deal with sound.

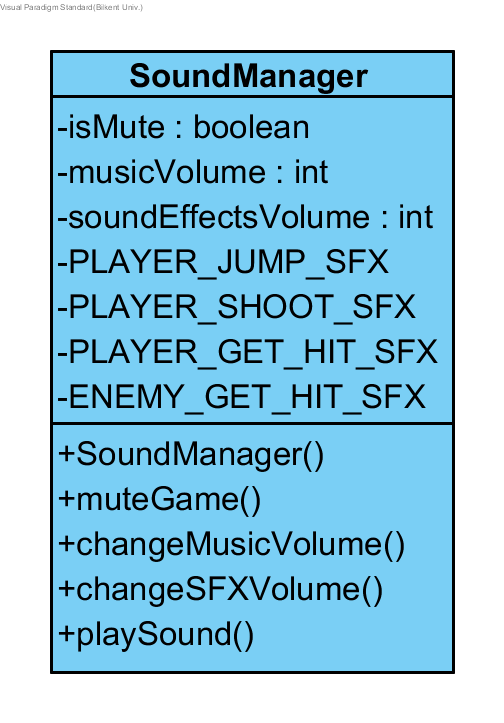
**Constructor:**

* **public GameManager(MainView view, Camera cam):** Constructer of gameManager class, it takes MainView and Camera classes’ instances and implements ithe interface GameManagerInterface.

**Methods:**

* **public void changeView( int i):** Checks the parameter and invokes the corresponding method of the MainView class, such as “displayPauseMenu”.
* **public void updateLevelView(int c):** Checks the Player object’s state in the level and change direction and direction of shooting accordingly.
* **public void tick():** Invokes the GameStateManager class’s method ‘tick()’, to be used in MainView.
* **public void render():** Invokes the GameStateManager class’s method ‘render()’, in order to be called in MainView.
* **public void loadImageLevel ( BufferedImage image):** Invokes the GameStateManager class’s method ‘loadImageLevel()’, in order to be called in MainView.
* **public void stopPlayer(int c)**: Checks for Player’s states that corresponds to stop, and sets velocity property of the Player object zero, eventually invokes MainView method to update view.
* **public void changeSound(int i):** Calls the soundManager’s changeSFXVolume() method with i as its parameter.
* **public void changeMusic(int i):** Calls the soundManager’s changeMusicVolume() method with i as its parameter.

**SoundManager Class:**

****

**Attributes:**

* **private boolean isMute:** Is a flag to determine whether the sound is enabled or disabled in the program.
* **private int musicVolume:** Holds the level of music volume as an integer.
* **private int soundEffectsVolume:** Holds the level of sound effects volume as an integer.
* **private static final AudioClip PLAYER\_JUMP\_SFX:** Holds audio clip to be played when the player jumps.
* **private static final AudioClip PLAYER\_SHOOT\_SFX:**
* **private static final AudioClip PLAYER\_GET\_HIT\_SFX:**
* **private static final AudioClip ENEMY\_GET\_HIT\_SFX:**

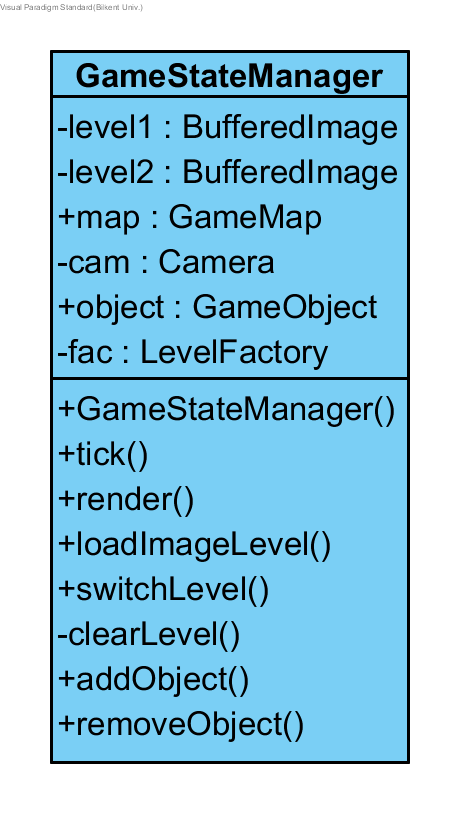
**Constructor:**

* **public SoundManager():** Initializes the object of this class, isMute is set false as default.

**Methods:**

* **public void muteGame(boolean mute):** Changes the isMute attribute to the parameter.
* **public void changeMusicVolume(int newMusic):** Changes the music volume level according to the parameter.
* **public void changeSFXVolume(int newVol):** Changes the music volume level according
* **public synchronized void playSound(int soundID):** This method is şnvoked by the GameManager class when needed. This method plays a sound sampling corresponding to the soundID.

**GameStateManager Class:**

****

**Attributes:**

* **private Camera cam:** Refers to the Camera object which is passed by parameter to the constructor. It is needed when level switches.
* **public GameMap map:** In order to update the whole objects, The GameStateManager class instantiates the GameMap object.
* **private BufferedImage level1, level2:** Holds the level images as buffered images.
* **public LinkedList<GameObject> object:** Holds the GameMap object’s LinkedList.

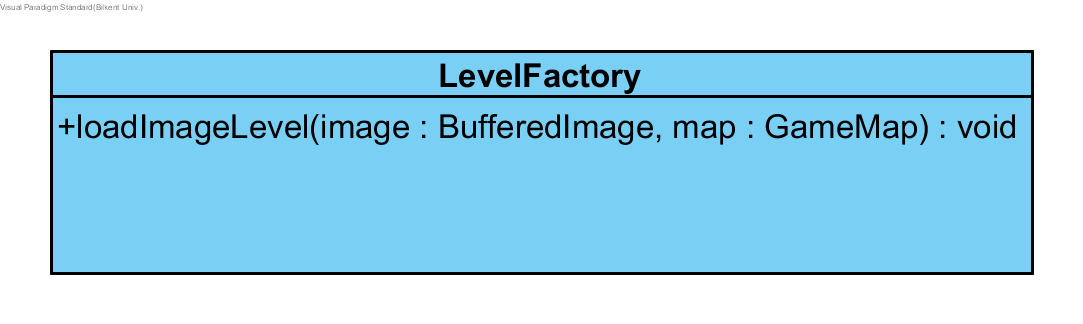
**Constructor:**

* **public GameStateManager(Camera cam):** Instantiates the GameStateManager class, with parameter Camera object. In the constructor GameMap and LinkedList get declared.

**Methods:**

* **public void tick():** Calls the ‘tick()’ method of GameMap class and checks whether level is changed or not. If corresponding boolean of the GameMap class is true, then here the level gets switched by ‘switchLevel()’ method.
* **public void render():** Invokes GameMap class’s render method.
* **public void switchLevel():** Invokes clearLevel() method and sets Camera to zero position, then calls loadImageLevel method to load the next level.
* **public void clearLevel():** Makes LinkedList of game objects clear.
* **public void addObject(GameObject object):** Calls the addObject() method of GameMap class.
* **public void removeObject(GameObject object):** Calls the removeObject() method of GameMap class.

**LevelFactory Class:**

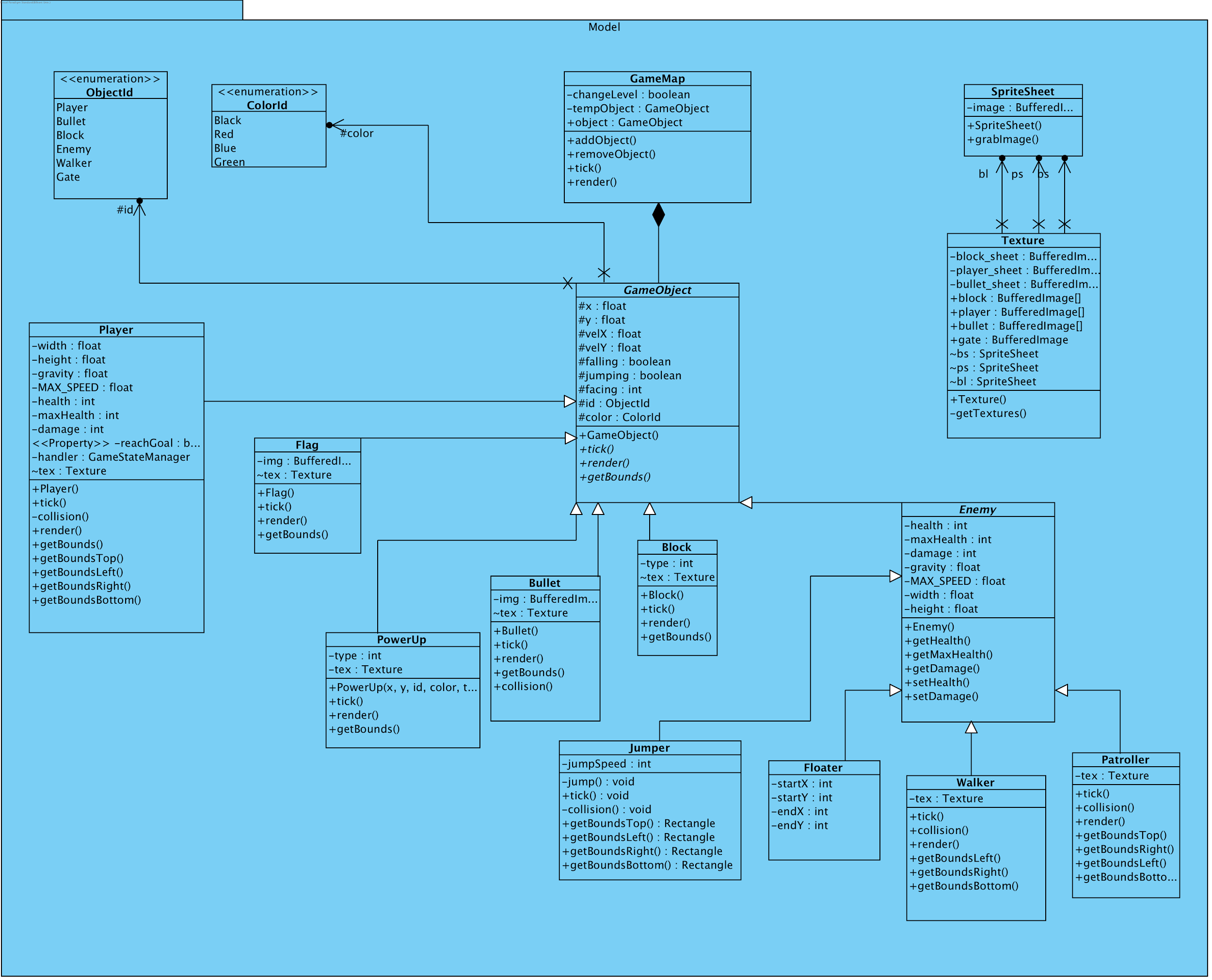
****

**Methods:**

* **public void loadImageLevel(BufferedImage image, GameMap map):** This method creates and adds game objects to the level.

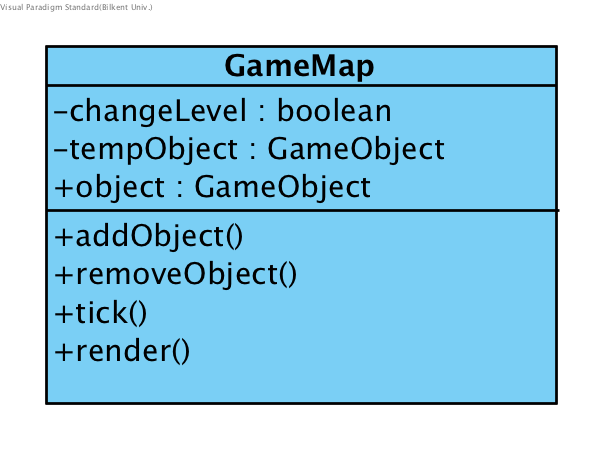
***3.4. Model Subsystem***

The model subsystem contains basically the functionality and data of the game itself, while View and Controller subsystems are mostly related with UI and the management of the game. The subsystem includes following classes: GameMap, GameObject, Player, Enemy, Bullet, Block, Flag, PowerUp and related enumeration classes, as well as Texture and SpriteSheet classes. We didn’t express all getter setter methods one by one.



*(Figure-7: Model Subsystem)*

**GameMap class:**

****

**Attributes:**

* **private LinkedList <GameObject> object:** It holds list of all objects for game entities.
* **private GameObject tempObject:** This attribute is to have an instance of Player class in GameMap class, to get and change the data with respect to interactions with Controller subsystem.
* **private booleaon changeLevel**

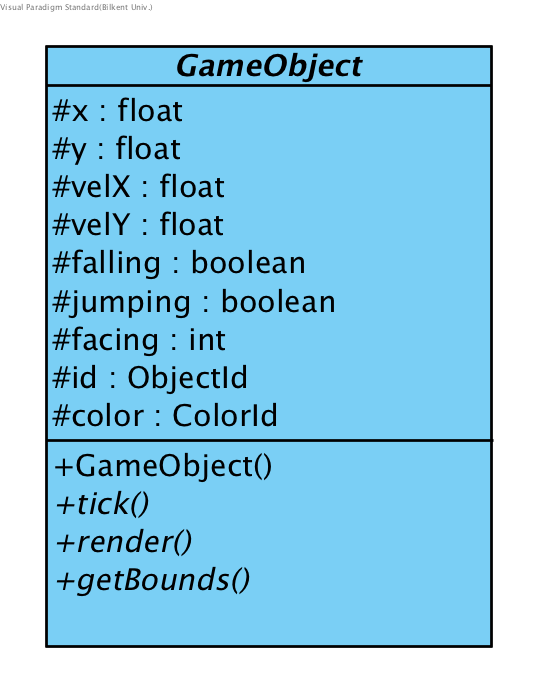
**Methods:**

* **public void tick():** Is used to update and check all objects by calling ‘tick()’ method of all objects.
* **private public void render(Graphics g):** Invokes render methods of all game objects.
* **public void LinkedList<GameObject> getObject()**
* **public void addObject(GameObject object)**
* **public void removeObject(GameObject object)**

**Getters and Setters:**

* **public boolean isChangeLevel()**
* **public void setChangeLevel(boolean changeLevel)**

**GameObject class:**

****

This class has common attributes for all game objects such as a x and y components, ID number and color type.

**Constructor:**

* **public GameObject(float x, float y, ObjectId id, ColorId color):** Constructs a GameObject.

**Attributes:**

* **protected float x:** This refers to x location of an object. All objects naturally has a x component.
* **protected float y:** This refers to y location of an object. All objects naturally has a y component.
* **protected ObjectId id:** This attribute is for the id number of game objects, which is provided for identifying each object.
* **protected ColorId color:** Color attribute is typical among game objects and it refers to the specific color of the player in this case.
* **protected int facing:** It determines whether the object faces right or left. If the object faces right, it equals “1”. Otherwise it equals “-1”.
* **protected float velX**
* **protected float velY**
* **protected boolean jumping**
* **protected boolean falling**

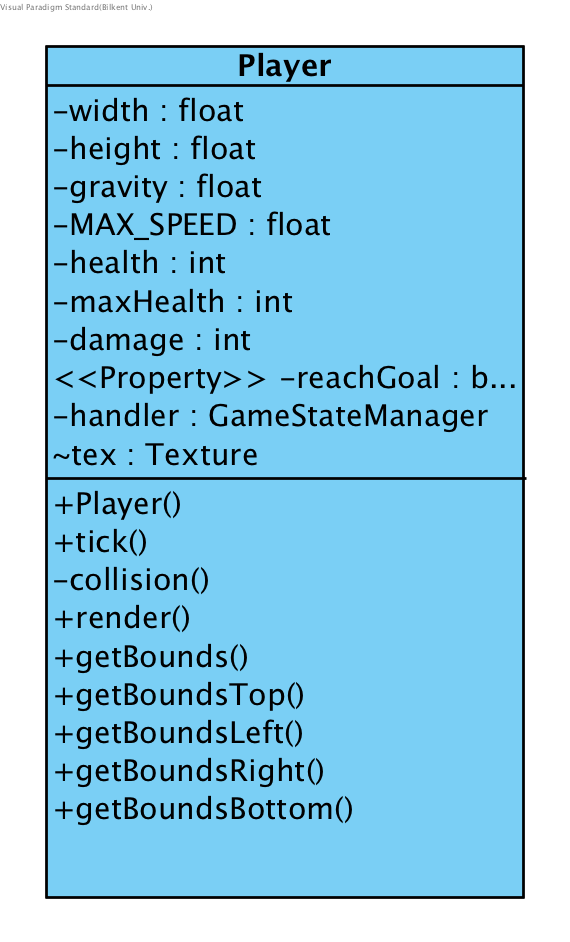
**Methods:**

* **public abstract void tick(LinkedList<GameObject> object):** An abstract method which will be used by child classes.
* **public abstract void render(Graphics g):** An abstract method which will be used by child classes for drawing purposes.

**Getters and Setters:**

* **public float getX()**
* **public float getY()**
* **public void setX(float x)**
* **public void setY(float y)**
* **public float getVelX()**
* **public float getVelY()**
* **public void setVelX(float velX)**
* **public void setVelY(float velY)**
* **public ObjectId getId()**
* **public int getFacing()**
* **public void setFacing(int facing)**
* **public ColorId getColor()**
* **public void setColor(ColorId color)**
* **public boolean isFalling()**
* **public void setFalling(boolean falling)**
* **public boolean isJumping()**
* **public void setJumping(boolean jumping)**

**Player class:**

****

Extends to the GameObject class.

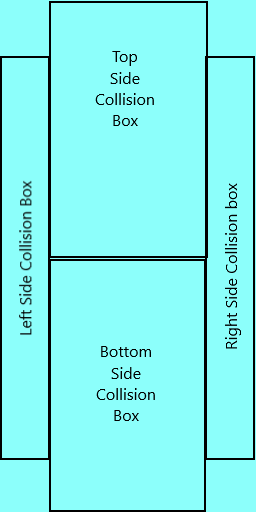
**Constructor:**

* **public Player(float x, float y, ObjectId id, ColorId color):** Constructs a Player which is also a GameObject.

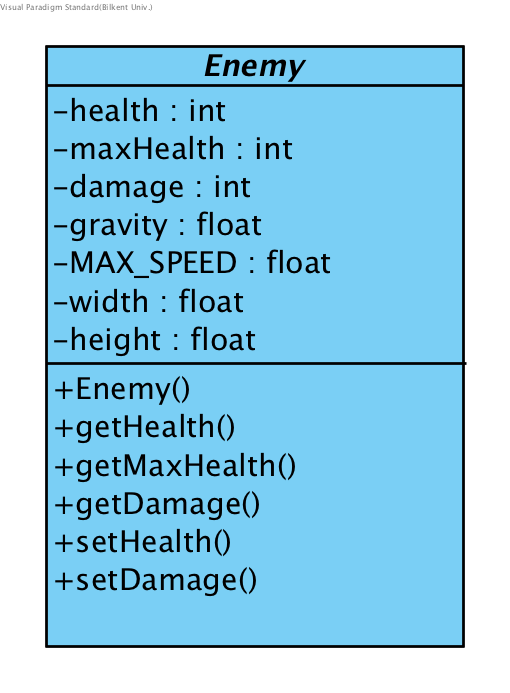
**Attributes:**

* **private final float MAX\_SPEED:** Refers to the maximum fall speed and the player can reach.
* **private int health:** Refers to how much health our player has.
* **private int maxHealth:** Refers to maximum health that the player has.
* **private float gravity**
* **private int damage**
* **private boolean reachGoal:**  Determines whether the player can progress the next level or not.

**Methods:**

* **public void tick(<LinkedList<GameObject> object):** Is used to update the player. It does this by incrementing the x position of the player by velX and the y position of the player by velY. It determines which way the player is facing by checking the sign of its velX. It also increases velY by gravity if the player is falling. The vertical speed gained by gravity cannot exceed the MAX\_SPEED. It also checks for collisions.
* **private void collision(LinkedList<GameObject> object):** Checks if the rectangle returned from the getBounds() method intersects with any other GameObjects’ getBounds() method.
* **public void render(Graphics g):** Invokes the drawImage function on g to draw the image of the block on the GamePanel. The image drawn depends on the color of the player (image) and x, y coordinates (location) .
* **public boolean ledgeCheck():** Checks if the player has moved over a ledge.
* **public Rectangle getBoundsTop():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking top collision.
* **public Rectangle getBoundsBottom():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking bottom collision.
* **public Rectangle getBoundsLeft():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking left side collision.
* **public Rectangle getBoundsRight():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking right side collision.
* **public void render(Graphics g):** Invokes the drawImage function on g to draw the image of the block on the gamepanel. The image drawn depends on the walkers color (image) and x, y coordinates (location) .
* **public boolean isReachGoal():** If it is true, player can progress the next level.
* **public void setReachGoal(boolean reachGoal)**

**Enemy Abstract Class:**

****

Extends to the GameObject class.

**Constructor:**

* **public Enemy(float x, float y, int maxHealth, ObjectId id, ColorId color ):** In this constructor x, y, id and color are used to call the constructor in the parent abstract class. The variable maxHealth is assigned to the maxHealth and health attributes.

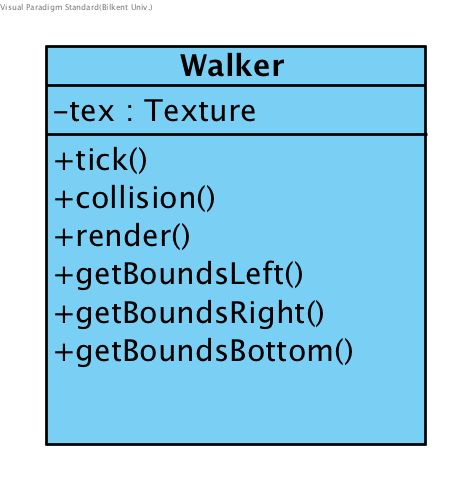
**Attributes:**

* **private int health:** Refers to how much health an enemy currently has.
* **private int damage:** Refers to how much damage collision with an enemy will do to the player.
* **private int maxHealth:** Refers to the maximum health an enemy has.
* **protected float width:** Refers to the width of the enemy object. Is a standard 64 for all enemy objects.
* **protected float height:** Refers to the height of the enemy object. Is a standard 64 for all enemy objects.
* **protected float gravity:** Refers to how much an enemy will acclerate int the +y direction (downawrds) when the enemy isn’t on top of a block.. Is a standard 0.3 for all enemy objects
* **protected final float MAX\_SPEED:** Refers to the maximum fall speed and enemy can reach. Is a standard 10 for all enemy objects.

**Methods:**

* **public int getHealth():** Returns health.
* **public int getMaxHealth():** Returns maxHealth.
* **public int getDamage():** Returns the amount of damage collision with the enmey does.
* **public void setHealth(int health):** Sets the enemy’s health to the health parameter.
* **public void setDamage(int damage):** Sets the enemy’s damage to the damage parameter.

**Walker class:**

****

Extends the Enemy abstract class.

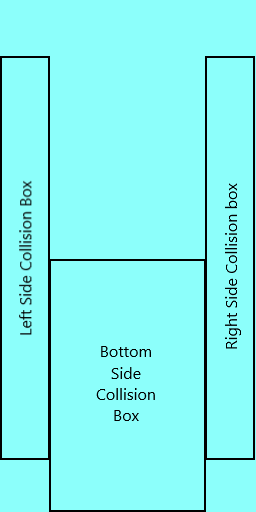
**Constructor:**

* **public Walker(float x, float y, int maxHealth, ObjectId id, ColorId color):** In this constructor x, y, max health, id and color are used to call the constructor in the parent abstract class. The attribute velX of the parent class is set to a standard 2 in the constructor.

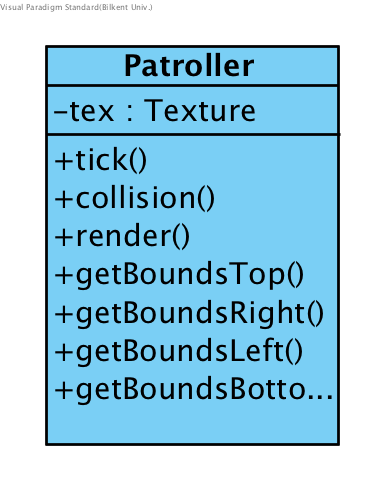
**Attributes:**

* **Texture tex:** References to the static Texture tex initialized in the GamePanel object. It is used to access the images corresponding to walkers.

**Methods:**

* **public void tick(LinkedList<GameObject> object):** Is used to update the enemy. It does this by incrementing the x position of the enemy by velX and the y position of the enemy by velY. It determines which way the walker is facing by checking the sign of its velX. It also increases velY by gravity if the walker is falling. The vertical speed gained by gravity cannot exceed the MAX\_SPEED attribute of the parent class. It also checks for collisions.
* **private void collision(LinkedList<GameObject> object):** Checks if the rectangle returned from the getBounds() method intersects with any other GameObjects’ getBounds() method. If it intersects with blocks the player’s velX is reversed. It also checks if the player is colliding with a block beneath it to determine whether the walker is falling or not.
* **public Rectangle getBoundsBottom():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking bottom collision.
* **public Rectangle getBoundsLeft():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking left side collision.
* **public Rectangle getBoundsRight():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking right side collision.
* **public void render(Graphics g):** Invokes the drawImage function on g to draw the image of the walker on the GamePanel. The image drawn depends on the walkers’ color (image) and x, y coordinates (location) .

**Patroller class:**

****

Extends the Enemy abstract class. Enemy that moves left and right . It changes its direction if it collides with a block or comes to a ledge.

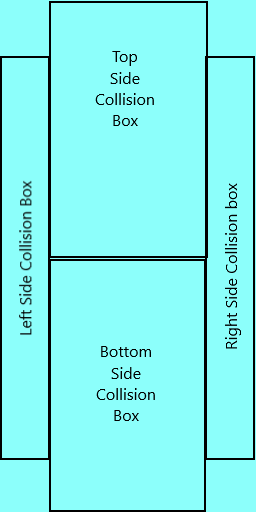
**Constructor:**

* **public Patroller(float x, float y, int maxHealth, ObjectId id, ColorId color):** In this constructor x, y, max health, id and color are used to call the constructor in the parent abstract class. the gsm parameter is assigned to the gsm attribute. The attribute velX of the parent class is set to a standard 2 in the constructor.

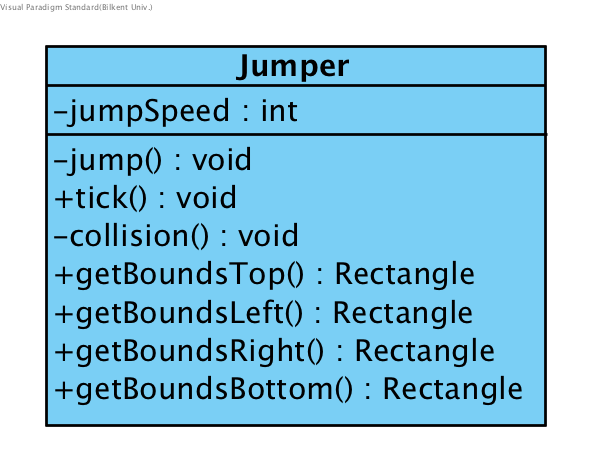
**Attributes:**

* **Texture tex:** References to the static Texture tex initialized in the GamePanel object. It is used to access the images corresponding to walkers.

**Methods:**

* **public void tick(LinkedList<GameObject> object):** Is used to update the patroller. It does this by incrementing the x position of the patroller by velX and the y position of the patroller by velY. It determines which way the patroller is facing by checking the sign of its velX. It also increases velY by gravity if the patroller is falling. The vertical speed gained by gravity cannot exceed the MAX\_SPEED attribute of the parent class. It also checks for collisions and handles collisions of the patroller object and also calls the turnAtLedge method.
* **private void collision(LinkedList<GameObject> object):**Checks if the rectangle returned from the getBounds() method intersects with any other GameObjects’ getBounds() method. If it intersects with blocks the enemy’s velX is reversed. It also checks if the patrolleris colliding with a block beneath it to determine whether the patrolleris falling or not.
* **public Rectangle getBoundsBottom():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking bottom collision.
* **public Rectangle getBoundsTop():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking top collision.
* **public Rectangle getBoundsLeft():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking left side collision.
* **public Rectangle getBoundsRight():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking right side collision.
* **public void render(Graphics g):** Invokes the drawImage function on g to draw the image of the patroller on the game panel. The image drawn depends on the patroller’s color (image) and x, y coordinates (location) .
* **private void turnAtLedge():** If the patroller is at a ledge it reverses the patroller’s speed. Is invoked in the tick(LinkedList<GameObject> object) method of the patroller class.

**Jumper class:**

****

Enemy that constantly jumps whenever it touches the floor.Extends the Enemy abstract class.

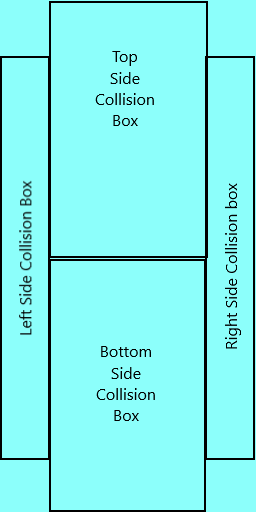
**Constructor:**

* **public Jumper(float x, float y, int maxHealth, ObjectId id, ColorId color, int jumpSpeed):** In this constructor x, y, max health, id and color are used to call the constructor in the parent abstract class. The parameter jumpSpeed is assigned to the jumpSpeed attribute. The attribute velX of the parent class is set to a standard 3 in the constructor.

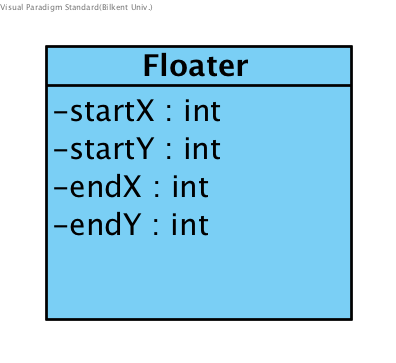
**Attributes:**

* **private int jumpSpeed:** Variable that determines how high the jumper object will jump.

**Methods:**

* **private void jump():** Function that makes the jumper object “jump” if the player is on the floor.It decides whether to jump or not based on the velY value of the object
* **public void tick(LinkedList<GameObject> object):** Is used to update the jumper. It does this by incrementing the x position of the jumper by velX and the y position of the jumper by velY. It determines which way the jumper is facing by checking the sign of its velX. It also increases velY by gravity if the walker is falling. The vertical speed gained by gravity cannot exceed the MAX\_SPEED attribute of the parent class. It also checks for collisions and handles collisions of the jumper object and also calls the jump() method.
* **private void collision(LinkedList<GameObject> object):**Checks if the rectangle returned from the getBoundsLeft() or getBoundsRight methods intersects with any other GameObjects’ getBounds() method. If it intersects with blocks the enemy’s velX is reversed. It also checks if the jumper is colliding with a block beneath it to determine whether the walker is falling or not.
* **public Rectangle getBoundsBottom():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking bottom collision.
* **public Rectangle getBoundsTop():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking top collision.
* **public Rectangle getBoundsLeft():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking left side collision.
* **public Rectangle getBoundsRight():** Returns a rectangle which will be used with the intersects method of the java.awt.Rectangle class for checking right side collision.****
* **public void render(Graphics g):** Invokes the drawImage function on g to draw the image of the jumper on the game panel. The image drawn depends on the jumper’s color (image) and x, y coordinates (location) .

**Floater class:**



Enemy that goes back and forth between two (x,y) coordinates without being affected by gravity. Extends the enemy class.

**Constructor:**

* **public Patroller(float x, float y, int maxHealth, ObjectId id, ColorId color):** In this constructor x, y, max health, id and color are used to call the constructor in the parent abstract class. The variable gravity is set to 0 and velX is set to 2.

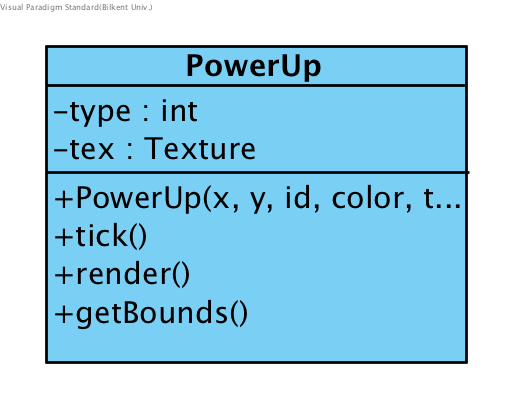
**Attributes:**

* **private int startX:** X coordinate of the starting position.
* **private int endX:** X coordinate of the ending position.

**Methods:**

* **public void tick(LinkedList<GameObject> object):** Is used to update the floater. It does this by incrementing the x position of the floaterby velX . It determines which way the walker is facing by checking the sign of its velX. It also checks if the floater has reached the startX or endX and reverses the velX if so.
* **public void render(Graphics g):** Invokes the drawImage function on g to draw the image of the floater on the game panel. The image drawn depends on the patrollers’ color (image) and x, y coordinates (location) .

**PowerUp class:**

****

Extends to the GameObject class.

**Constructor:**

* **public PowerUp(float x, float y, ObjectId id, ColorId color, int type):** In this constructor x, y, id and color are used to call the constructor in the parent abstract class. The type parameter will be assigned to the type attribute of the object.

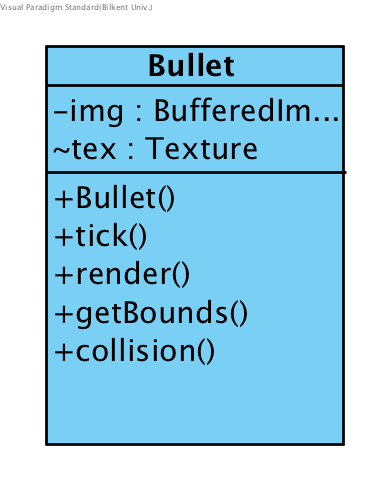
**Attributes:**

* **private int type:** This variable holds the type of power-up. It is of public enum PowerUpId class. The type variable will be used to determine what effects the power up will cause and which image will be selected for the powerUp object(s).
* **Texture tex:** References to the static Texture tex initialized in the GamePanel object. It is used to access the images corresponding to power-ups.

**Methods:**

* **public void tick(LinkedList<GameObject> object):** Does not do anyting. It was added to this class as a necessity due to inheritance from an abstract class containing this abstract method.
* **public void render(Graphics g):** Invokes the drawImage function on g to draw the image of the block on the gamepanel. The image drawn depends on the blocks type(image) and x, y coordinates (location).
* **public Rectangle getBounds():** Returns a 16x16 pixel rectangle (the standard size of blocks) with the same position as the power-up. This method is used to detect and handle collision between power-ups and the player.

**Bullet class:**



Extends to the GameObject class.

**Constructor:**

* **public Bullet(float x, float y, ObjectId id, ColorId color, int velX):** In this constructor x, y, id and color are used to call the constructor in the parent abstract class. The parameter velX is to determine the speed of the bullet in the x-axis. The resaon we have this parameter is so we can shoot the bullet in the direction the shooter (player, enemies) is facing.

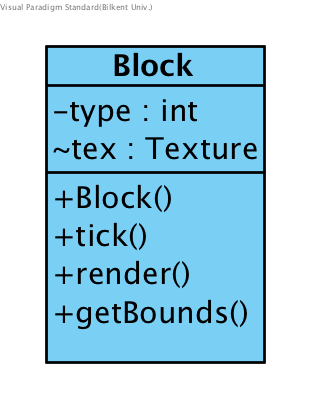
**Attribute**

* **Texture tex:** References to the static Texture tex initialized in the GamePanel object. It is used to access the images corresponding to bullets (red bullet, blue bullet, green bullet).

**Methods:**

* **private void collision(LinkedList<GameObject> object):** Checks if the rectangle returned from the getBounds() method intersects with any other GameObjects’ getBounds() method. If it intersects with the player the player will be damaged, if it intersects with an enemy the enemy will be damaged if the enemey is the same color as the enemy. In both cases the bullet will be destroyed by getting removed from the object LinkedList<> . The bullet will also be destroyed if it collides with a block.
* **public Rectangle getBounds():** Returns a 16x16 pixel rectangle (the standard size of blocks) with the same position as the bullet. This method is used to detect and handle collision between bullets and other GameObjects (player, enemies, blocks).
* **public void render(Graphics g):** Invokes the drawImage function on g to draw the image of the block on the gamepanel. The image drawn depends on the bullets’ type(image) and x, y coordinates (location) .
* **public void tick(LinkedList<GameObject> object):** Updates the bullets position by incrementing x by velX. It also checks for collision with every object in the object LinkedList<>.

**Block Class:**

****

Extends to the GameObject class.

**Constructor:**

* **public Block(float x, float y, int type, ObjectId id, ColorId color):** In this constructor x, y, id and color are used to call the constructor in the parent abstract class. The parameter type is assigned to the type attribute of the object.

**Attributes:**

* **Texture tex:** References to the static Texture tex initialized in the GamePanel object. It is used to access the images corresponding to blocks.
* **private int type:** Holds the type of the block. This value ranges between 0 - 11. With:
  + 0-3: Standart Blocks
  + 4-7: Fading Blocks
  + 8-11: Spikes

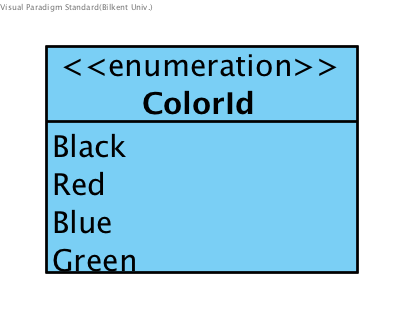
where:

* + 0, 4, 8: are black blocks,
  + 1, 5, 9: are red blocks,
  + 2, 6, 10: are blue blocks and
  + 3, 7, 11: are green blocks.

**Methods:**

* **public void tick(LinkedList<GameObject> object):** Does not do anything. It was added to this class as a necessity due to inheritance from an abstract class containing this abstract method.
* **private void render(Graphics g):** Invokes the drawImage function on g to draw the image of the block on the gamepanel. The image drawn depends on the blocks type(image) and x, y coordinates (location).
* **public Rectangle getBounds():** Returns a 32x32 pixel rectangle (the standard size of blocks) with the same position as the block. This method is used to detect and handle collision between blocks and moving GameObjects (player, enemies).

**ColorId Enumeration:**



There will be 4 types of ColorId’s in this game. This enumeration will be used for ColorId variables. The ColorId is an important varibale in determining how to handle collisions between different GameObjects.

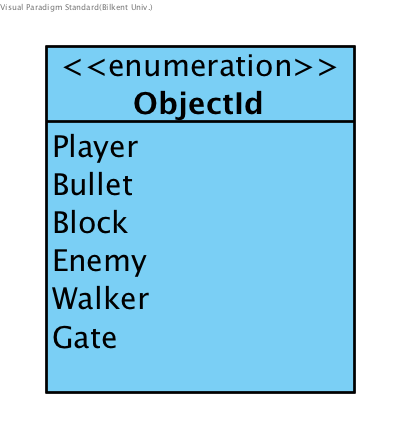
**-Red()**

**-Green()**

**-Blue()**

**-Black()**

**Object Id Enumeration:**

****

There will be 6 types of ObjectId’s in this game. This enumeration will be used for GameObjects. The ObjectId is an important variable in determining what type of the object is.

***4.Improvement Summary***

In the second iteration, the design goals of the system are reevaluated and according to these design goals, new design patterns are used, which are Factory Pattern and Strategy Pattern. They support the system extendibility and additionally support the reusability of the subsystems. The subsystem hierarchy is redesigned, where controller and view has partitioning on the top and controller accesses the model, which is at lower hierarchy above them. Mainly the classes which are declared before are stable. However some classes are removed for instance ScreenManager is removed, instead an additional class DifficultySelectionPanel is added, which was forgotten before. ChangeListener is implemented by MenuListener. BackgroundImageLoader and Camera classes are added to the view subsystem. In controller, CollisionManager is removed, now it is handled in methods.InputManager is removed as well, as all the user interaction is recognized by the view, by MenuView class. LevelFactory is added in order to encapsulate creations of the game objects. GameManagerInteface is defined, which supports the Strategy pattern.

Lastly, in model mainly the classes are same, some simplifications in the methods and properties are done. Texture and SpriteSheet classes are added. Texture is created by the GameManager. ObjectId and ColorId enumarations are added.