

Bilkent University

Department of Computer Engineering

**CS 319 - Object-Oriented Software**

**Engineering Project Report**

*Color Shooter: The Spectrum Adventurer*

System Design Report Draft

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…………………………………………………………………………….. 4**

**1.1. Purpose of the system………………………………………………………... 4**

**1.2. Design Goals.…………………………………………………………………. 4**

**1.2.1. Performance Criteria…………………………………………….... 4**

**1.2.2. Maintenance Criteria……………………………………………… 5**

**1.2.3. End User Criteria …………………………………….………….... 5**

**1.2.4. Trade Offs..…………………………………………………………. 5**

**2. Proposed Software Architecture.…………………………………………………..….. 6**

**2.1. Overview..…………………………………………………………………….. 6**

**2.2. Subsystem Decomposition.…………………………………………………... 7**

**2.3. Architectural Styles.………………………………………………………….. 9**

**2.3.1 Layers.……………………………………………………………….. 9**

**2.3.2 Model View Controller.……………………………………………... 9**

**2.4. Hardware / Software Mapping.…………………………………………….. 10**

**2.5. Persistent Data Management……………………………………………….. 10**

**2.6. Access Control and Security.………………………………………………... 11**

**2.7. Boundary Conditions.……………………………………………………….. 11**

**3. Subsystem Services.…………………………………………………………………... 12**

**3.1. Design Pattern : Facade Design…...…………………………………....…... 12**

**3.2. View Subsystem Interface……………..………………………………..….. 13**

**MainMenuView Class.……………………………………..………….…. 14**

**PauseMenuView Class.…………………...……………………………… 16**

**ScreenManeger Class.……………………………………….………….... 17**

**MyActionListener Class.……………………………………………….… 19**

**3.3 Controller Subsystem……………………....……………………………….... 19**

**GameManager Class.……………………………………...……………... 20**

**SoundManager Class.………………………………………………...…... 22**

**CollisionManager Class.……………,,,,…………………………...…….. 23**

**GameStateManager Class.……………………………………...……….. 23**

**InputManager Class.…………………………………………………….. 24**

**GameInformation Class.…………………………………….…………... 25**

**3.4 Model Subsystem……………………...…………………………………….. 26**

**GameMap Class.………………………………………………………… 27**

**TileMap Class.……………………………………………………..…,…. 28**

**GameObject Class.……………………………………………………..... 28**

**Sprite Class.…………………………………………………….………... 29**

**Player Class.…………………………………………………….………... 29**

**Enemy Class.…………………………………………………….……….. 30**

**Walker Class.…………………………………………………….………. 31**

**Jumper Class.…………………………………………………….……… 31**

**Floater Class.…………………………………………………….………. 31**

**Weapon Class.…………………………………………………….……... 32**

**Powerup Class.…………………………………………………….…….. 32**

**Bullet Class.…………………………………………………….………... 33**

**Tiles Class.…………………………………………………….……...….. 33**

**EnumType Class.……………………………………………….………... 34**

**EnumColor Class.………………………………………………………... 35**

**1. *Introduction***

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

Color Shooter: The Spectrum Adventurer is a 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***

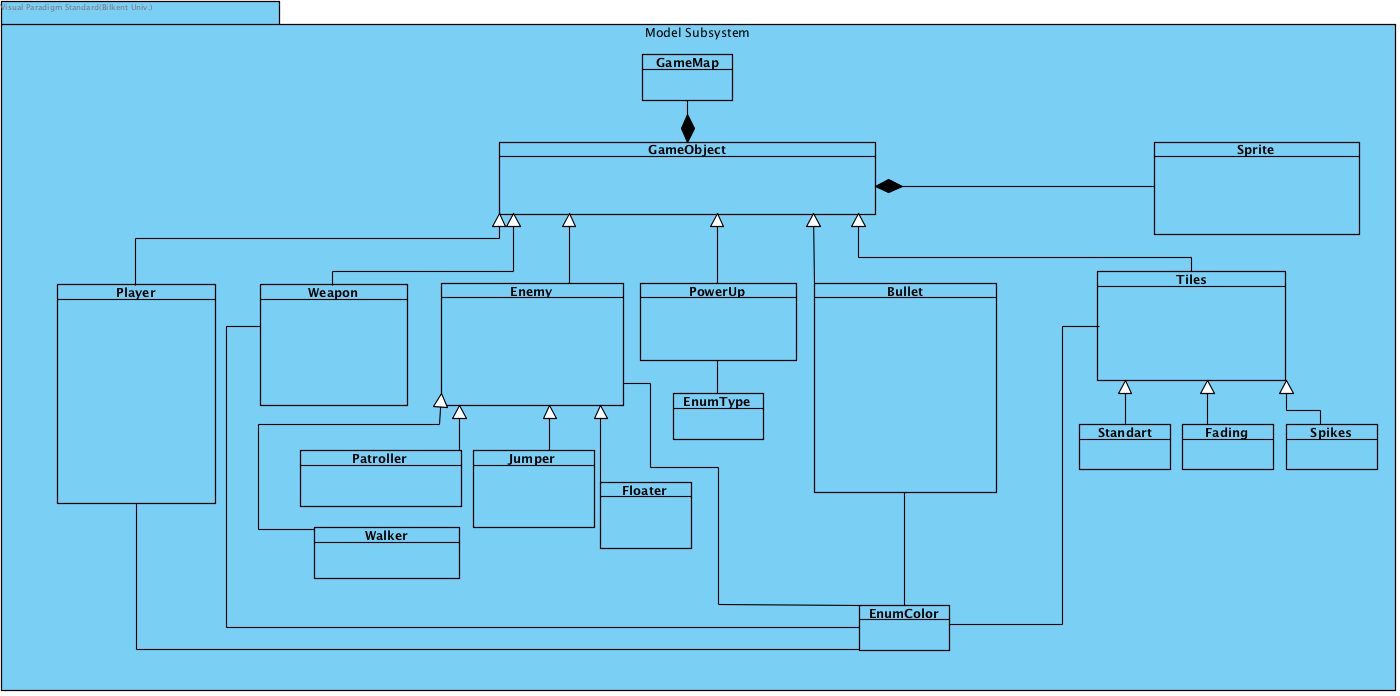
**2.1. *Overview***

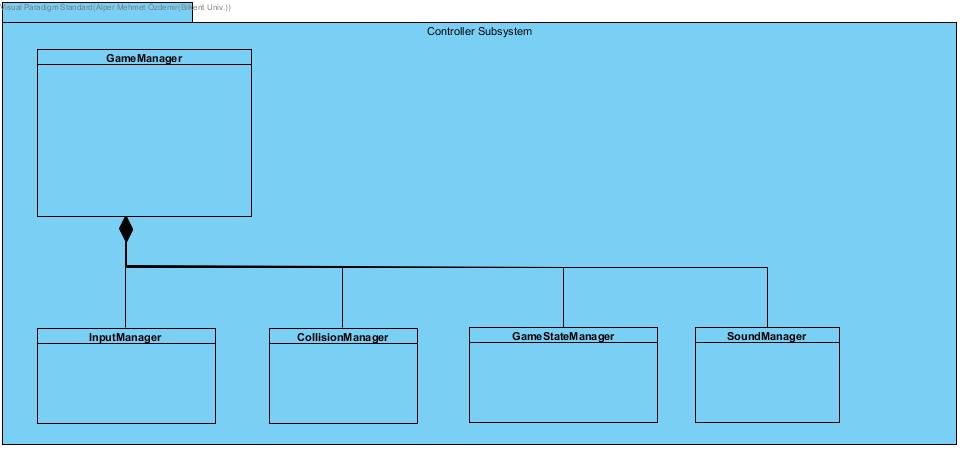
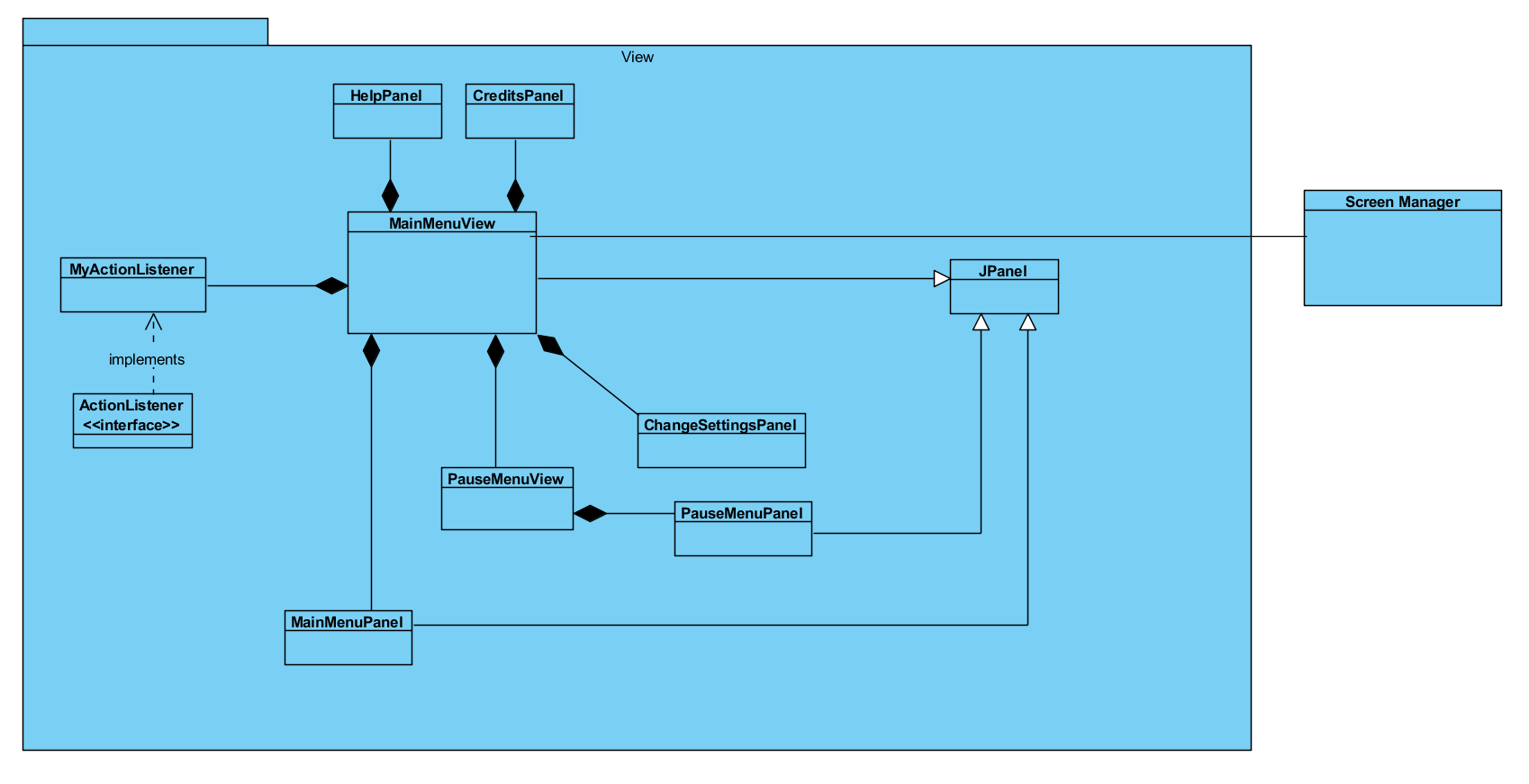
In order to overcome the complexity, the system is decomposed into manageable parts taken into consideration the functionalities defined in the requirements elicitation and design report. While decomposing the system, our main aim was to design subsystems which minimizes coupling and maximizes cohesion. 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. ScreenManager class provides access to the View subsystem. GameManager class will perform several operations according to the user requests on the menu, while ScreenManager provides user choices and provides frame to the GameManager. GameMap class in the View 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: Model Subsystem)*

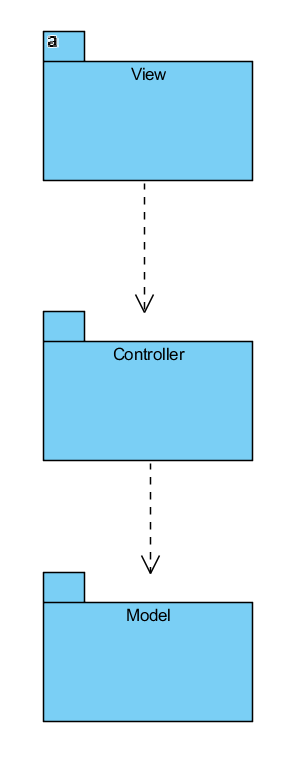
*(Figure-2: View Subsystem)*

*(Figure-3: Controller Subsystem)*

**2.3. *Architectural Styles***

**2.3.1 *Layers***

The system is decomposed hierarchically hence it creates mainly three layers, which are View, Controller and 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. The following Figure- 4 illustrates the layering in the system.



*(Figure-4: Layers )*

**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 has a class which interacts with other subsystems. This is designated for a particular purpose, in order to decrease coupling as stated before.

**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 .txt 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***

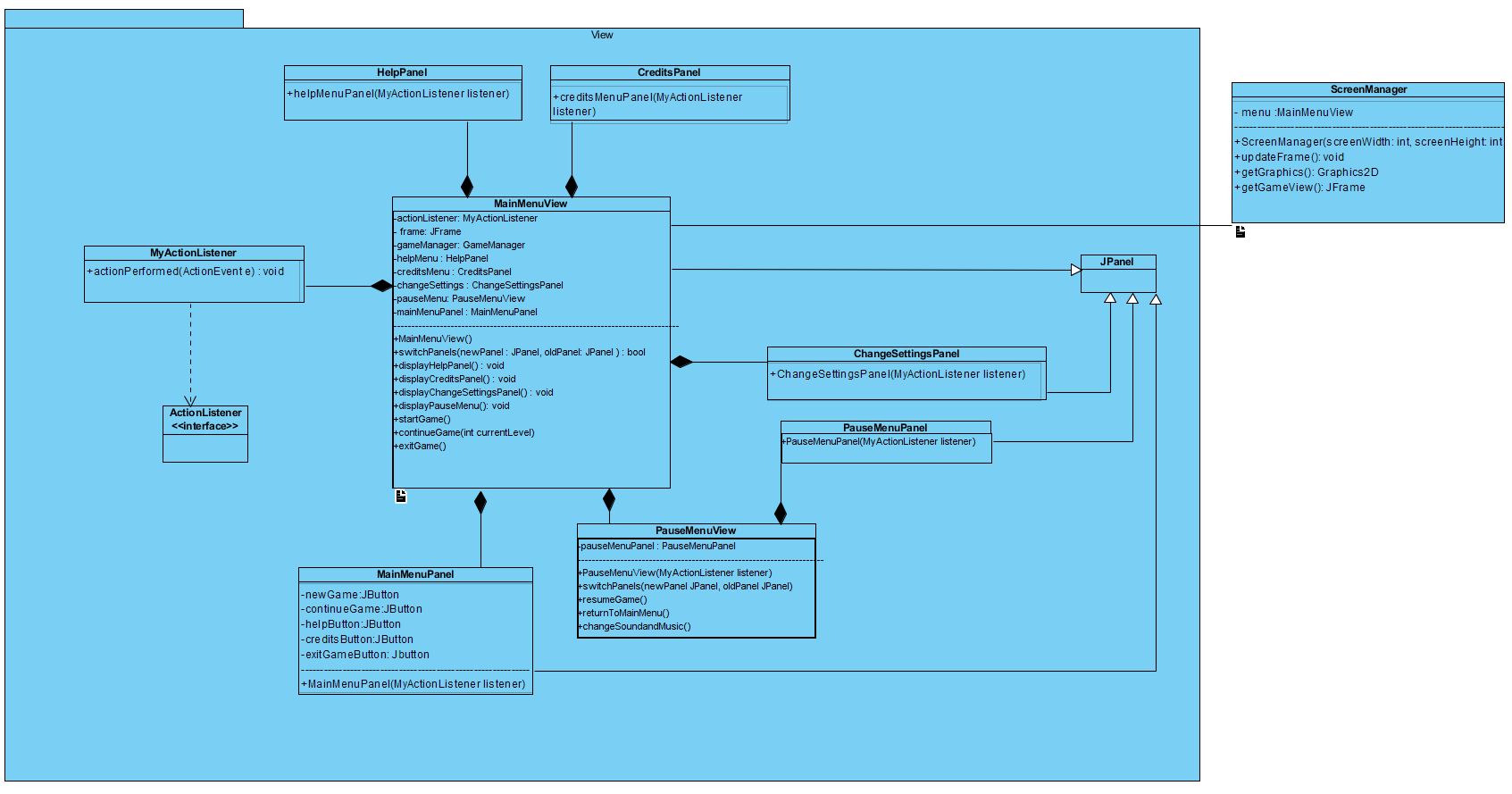
Subsystem services ensured by each layer will be explained in detail. First the design pattern used in the system will be explained and then each class in the system, their attributes and methods will be stated.

**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.

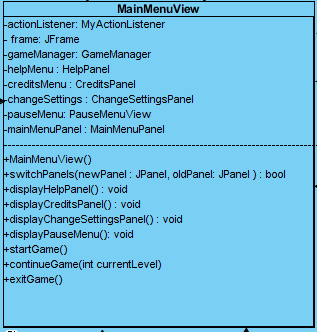
In our system, there are three classes in each subsystem which can be declared as Facade classes. ScreenManager class, GameManager class and GameMap classes corresponds to the Facade classes of the View, Controller and Model respectively. GameMap class provides the information to the View subsystem about game objects/entities. 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. ScreenManager class provides the user interface frame and the graphics.

***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, ScreenManager class is the Facade class which enables other subsystems to get frame and access the menu panels. Frame and panels of the game is instantiated in this subsystem and the menu panels are totally separated from the controller and model.

**MainMenuView class:**



**Attributes:**

* **private MyActionListener 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.
* **private GameManager gameManager:** GameManager is instantiated in order to perform startGame() and continueGame() methods. Additionally, it will be used for supplying necessary updates for the game when pause menu is displayed.
* **private HelpPanel helpMenu:** It is used in order to display the user interface on the screen to the player. It is a JPanel and it will include labels and a back button.
* **private CreditsPanel creditsMenu:** CreditsPanel is a JPanel. It is used to view user interface of credits. There will be a single label which describes the project and the developers and e back button to return to the main menu.
* **private ChangeSettingsPanel changeSettings:** ChangeSettingsPanel is a JPanel. It allows users to change music and sound effects. There will be a additional return button which returns to the main menu view.
* **private PauseMenuView pauseMenu:** Pause menu view has an instance of PauseMenuPanel. Additionally, it contains additional methods which can be done in this panel.
* **private MainMenuPanel mainMenuPanel:** MainMenuPanel is a JPanel. It will have a background image, and six buttons with labels on it.

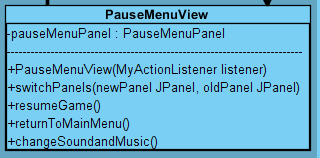
**Constructor:**

* **public MainMenuView():** Constructor initializes Game manager, MyActionListener, each panels and views in the attributes of the class.

**Methods:**

* **public boolean switchPanels (JPanel newPanel, JPanel oldPanel):** According to the player selection on the main menu panel, this methods allows programmer to change the current panel on the screen.
* **public void displayHelpPanel():** It calls the switchPanel() with appropriate parameters and displays the user interface of Help menu.
* **public void displayCreditsPanel():** It calls the switchPanel() with appropriate parameters and displays the user interface of Credits.
* **public void displayChangeSettingsPanel():** It calls the switchPanel() with appropriate parameters and displays the user interface of Change Settings.
* **public void displayPauseMenu():** It calls the switchPanel() with appropriate parameters and displays the user interface of Pause.
* **public void startGame():** It initializes the game by using the instance of GameManager object. GameManager will handle the game dynamics.
* **public void continueGame(int currentLevel):** It will be invoked when user clicks to the continueGame button in the Main Menu. It requires the level which player has left the game in the parameters. It will use the gameManager object to run the game.
* **public void exitGame():** It will end the game immediately, however beforehand it will save the current level.

**PauseMenuView class:**

****

**Attributes:**

* **private PauseMenuPanel pauseMenuPanel:** It is a JPanel object and used for the user interface of PauseMenu.

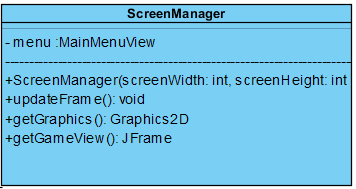
**Constructor:**

* **public PauseMenuView(MyActionListener listener):** It initializes the the pauseMenuPanel.

**Methods:**

* **public void switchPanels( JPanel newPanel, JPanel oldPanel):** According to the player selection on the main menu panel, this methods allows programmer to change the current panel on the screen.
* **public void resumeGame():** It removes the panel on top which is the Pause menu and continues the game.
* **public void returnToMainMenu():** It calls the switchPanels() method with appropriate parameters and displays the MainMenu by removing pauseMenu on the screen.
* **public void changeSoundandMusic():** According to the player input it changes the the Sound and Music though GameManager object.

**ScreenManager class:**

****

**Attributes:**

* **private MainMenuView menu:** ScreenManager class can acces the JFrame through MainMenuView object.

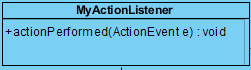
**Constructor:**

* **ScreenManager(int ScreenWidth, int screenHeight):** It gets height and width of the screen and initializes the MainMenuView object.

**Methods:**

* **public void updateFrame():** It updates the frame according to GameManager object, it repaints the components. It access the GameManager through the MainMenuView object.
* **public Graphics2D getGraphics():** It returns the graphics on the screen.
* **public JFrame getGameView():** It returns the frame, a JFrame object.

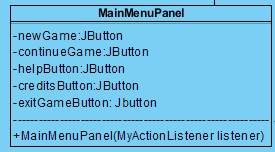
**MyActionListener class:**

****

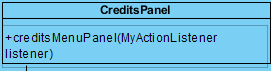
* **public void actionPerformed(ActionEvent e):** It implements the actionPerformed method. This method handles the actions performed when button is clicked.

The following classes are Panels for the view and their constructors require a MyActionListener object as there will be buttons on the screen on each panel. These classes are instantiated in the MainMenuView class.

**MainMenuPanel:**

****

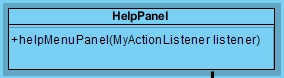
**CreditsPanel:**



**PauseMenuPanel:**

**pausemenupanel.PNG**

**HelpPanel:**

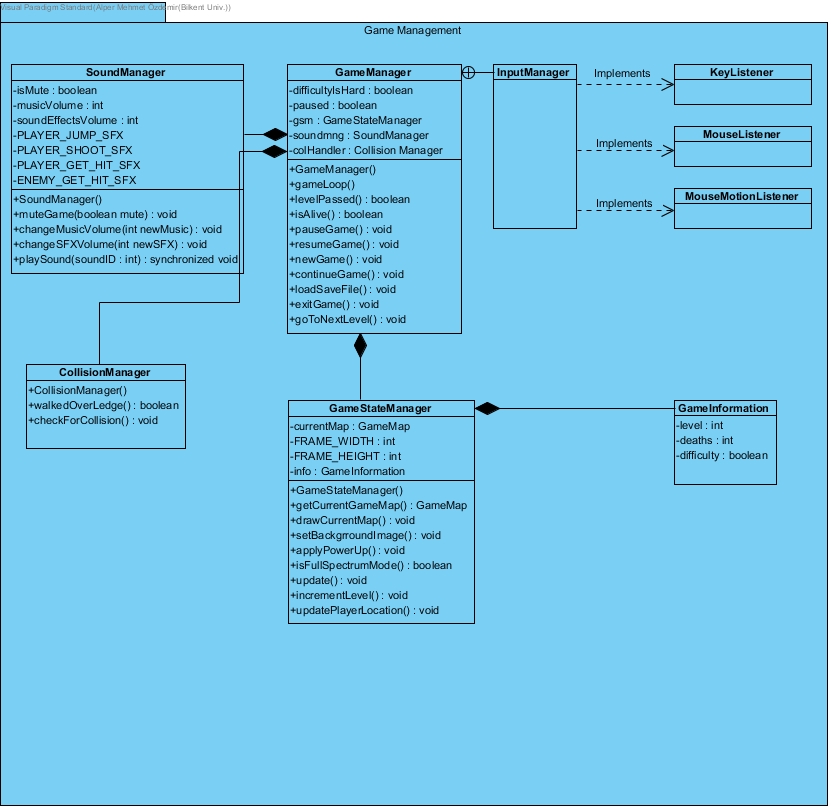
****

**ChangeSettingsPanel:**

***changesettingspanel.PNG***

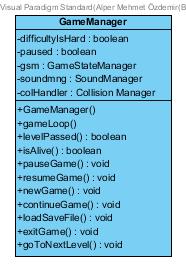
***3.3. Control Subsystem***

In this subsystem our controller classes are are grouped together to manage the game dynamics and game logic. We have 6 components in this subsystem. Five of them are controller classes and one of them is a property class. As illustrated in Figure-5, we have the GameManager, SoundManager, Collision Manager, GameStateManager classes and an InputManager class which is a subclass of the GameManager class and will be used for handling user input. We also have a GameInformation class as a property class of the GameStateManager. These classes will be further explained below.



*(Figure-6: Control 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 falue 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.
* **private CollisionManager colHandler:** Holds a CollisionManager object to handle collisions between game objects.

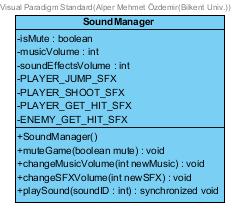
**Constructor:**

* **public GameManager(SoundManager soundmng, CollisionManager mng, GameStateManager gsm):** Creates a game GameManager object holding the other controller objects of the control subsystem

**Methods:**

* **public void gameLoop():** Initiates a loop in which the system in constantly updated until the player quits the game
* **public boolean levelPassed():** Checks if the player has finished the current level.
* **public boolean isAlive():** Returns true if the players current health is bigger than zero.
* **public void pauseGame():** When invoked stops the game loop from iterating by setting the paused attribute to true.
* **public void resumeGame():** When invoked allows the game loop to continue iterating by setting the paused attribute to false.
* **public void newGame()**: Starts the game from level 1, resetting the data on the save file(.text file) to initial values.
* **public void continueGame():** Continues game based on the data on the save file. Gets the save file from the loadSaveFile() method.
* **public void loadSaveFile():** Loads data from a save file(.txt file).
* **public void exitGame():** Updates the save file and Terminates the program.
* **public void goToNextLevel():** Checks the value from the levelPassed method and updates the game to go to the next level if true.

**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:**

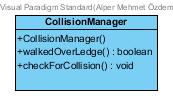
**Constructor:**

* **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.

**CollisionManager Class:**

****

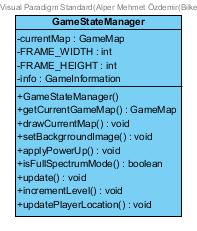
**Constructor:**

* **public CollisionManager():** Instantiates the collision manager which handles all collision events between the player, tiles, enemies and bullets.

**Methods:**

* **public boolean walkedOverLedge():** Returns true if the player has walked over a ledge. Will only be used for hard difficulty.
* **public void checkForCollision():** Handles all collisions.

**GameStateManager Class:**

****

**Attributes:**

* **private GameMap currentMap:** Holds ths GameMap object for the current level.
* **private GameInformation info:** Holds the game information for the current level.
* **private static final int FRAME\_WIDTH:** Holds the game frame’s width value.
* **private static final int FRAME\_HEIGHT:** Holds hte game frame’s height value.

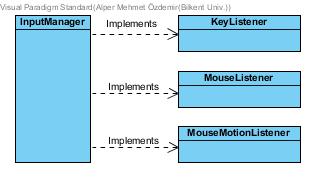
**Constructor:**

* **public GameStateManager():** Initializes a GameStateManager object with default attribute values.

**Methods:**

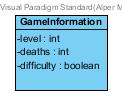
* **public GameMap getCurrentGameMap():**  Returns the currently used GameMap object for the current level.
* **public void drawCurrentMap():** Contains the algorithm to draw game objects base on the currentMap attribute.
* **public void setBackgroundImage(BufferedImage newBackGround):** Sets the parameter as the background image.
* **public void applyPowerUp(PowerUp pwrUp):** Method containing how the powerup in the parameter effects the game. Will be used together with the CollisionManager object.
* **public boolean isFullSpectrumMode():** Reutrns true if the player is effected by the full spectrum mode power-up. Will be used together with the collision handler object.
* **public void update():** Updates all data regarding the GameStateManager object.
* **public void incrementLevel():** Advances to the next level.
* **public void updatePlayerLocation()**  Updates the player objects location based on input recieved by the InputManager object.

**InputManagerClass:**

****

* This class is designed to detect user actions performed by the mouse(for menu item selection) and keyboard (gameplay). In this context, this class implements necesar

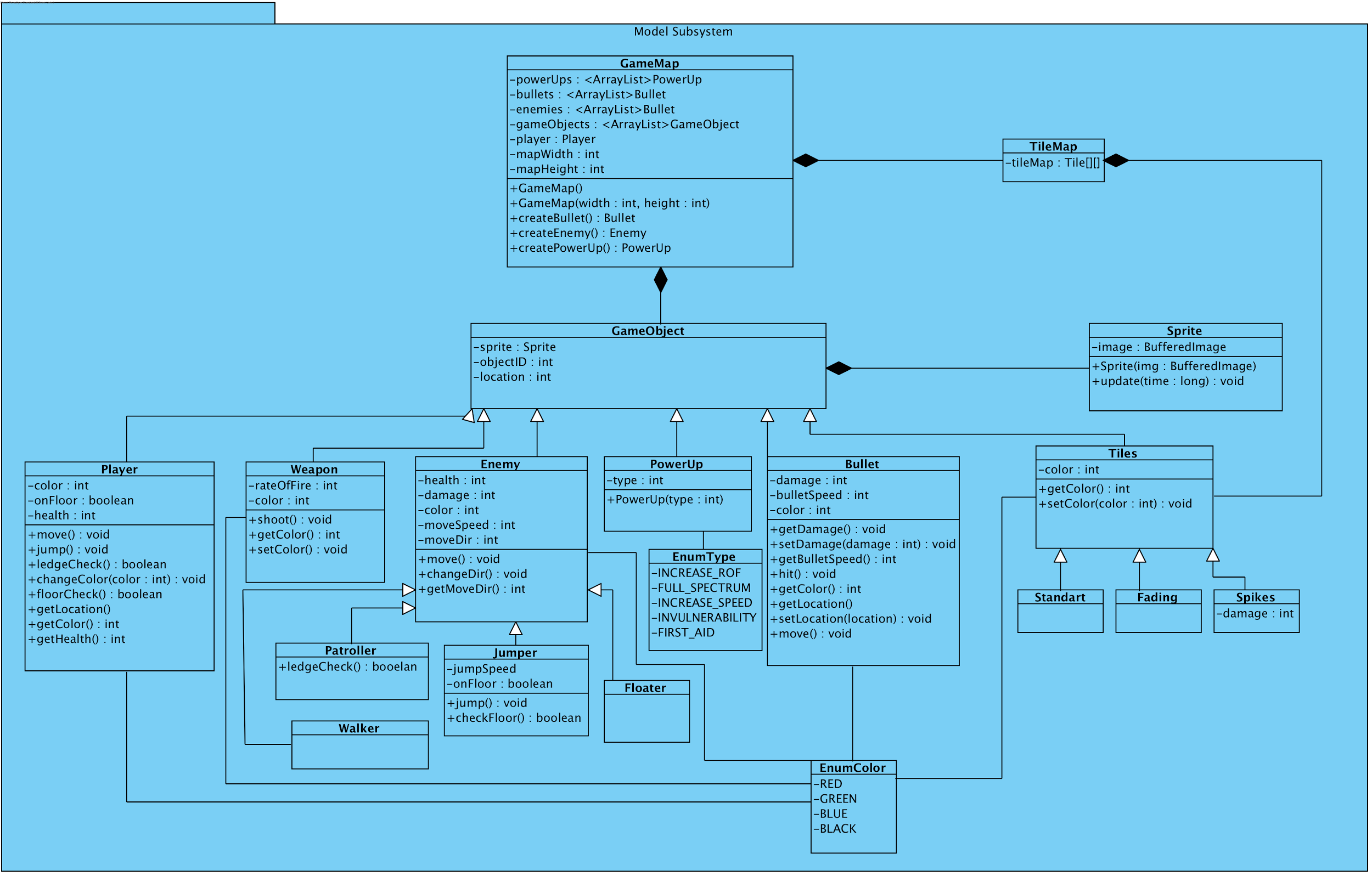
**GameInformationClass:**

****

* This class is a simple property class, which holds the information about the current state of the game. It is used by GameStateManager class.

***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. GameMap class will be the façade class of the Model subsystem and be the one that communicates with other subsystems. The subsystem includes following classes: GameMap, GameObject, Sprite, Player, Enemy, Tiles, Weapon, Bullet and related enumeration classes. We didn’t express all getter setter methods one by one.



*(Figure-7: Model Subsystem)*



**GameMap class:**

**Constructor:**

* **public GameMap():** It is a constructor for GameMap with default values.
* **public GameMap(int width, int height):** This method constructs a game map with using width and height parameters.

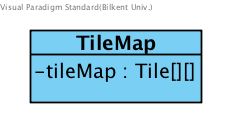
**Attributes:**

* **private Player player:** 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 ArrayList <GameObject> gameObjects:** It holds all arraylists of objects for game entities.
* **private ArrayList <Enemy> enemies:** It refers to all Enemy objects in game, and holding references with arraylist type will provide feasibility to deal with operations that interacts with objects.
* **private ArrayList <PowerUp> powerUps:** This list keeps all power up objects in it.

**Methods:**

* **private Bullet createBullet():** Creates a bullet with attributes relative to the weapon objects direction and location.
* **private Enemy createEnemy():**
* **private Enemy createPowerUp():**

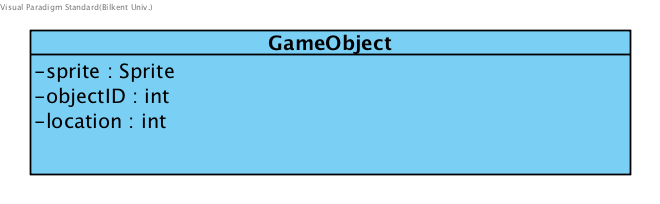
**TileMap class:**

****

**Attributes:**

* **private Tile[][] tileMap:** This 2-D array holds the references of all Tile objects that exist.

**GameObject class:**

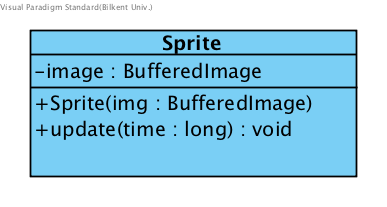
****

This class has common attributes for all game objects such as a sprite, location and an id number.

**Attributes:**

* **private Sprite sprite:** This attribute refers to sprite image of game objects which is hold in here since all objects naturally have an image.
* **private int objectID:** This refers to the id number of game objects, which is provided for identifying each object.
* **private int location:** This refers to location and it is common for all game objects as well.

**Sprite class:**

****

**Constructor:**

* **public Sprite(img: BufferedImage):** Constructs a Sprite image object.

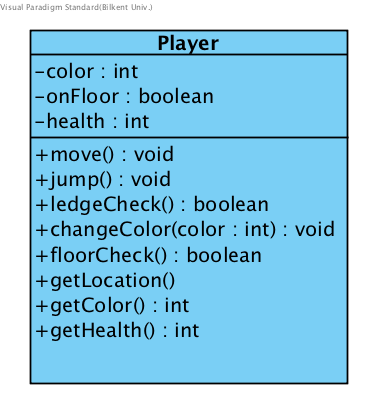
**Attributes:**

* **private BufferedImage image:** Refers an image corresponding to Sprite.

**Methods:**

* **public void update(time : long):** This method is to update the image while parameter time changes during the game play.

**Player class:**

****

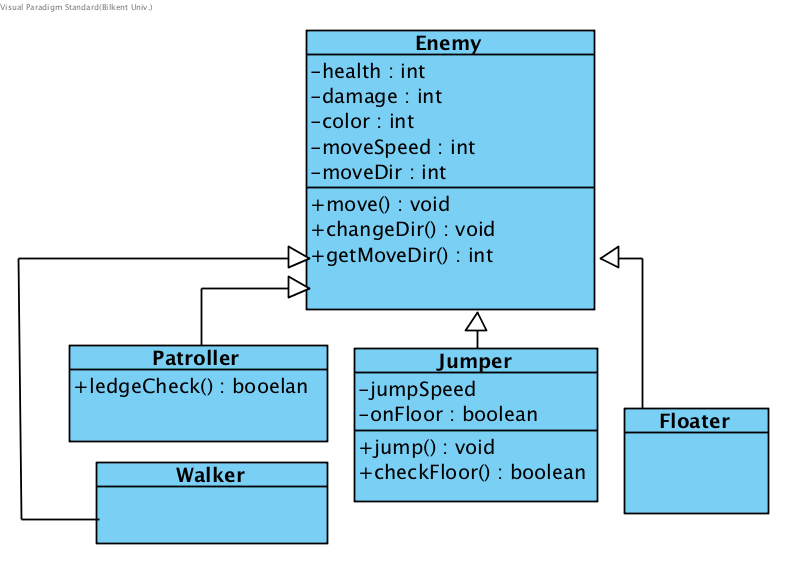
**Attributes:**

* **private int color:** Color attribute is typical among game objects except PowerUp objects and it refers to the specific color of the player in this case.
* **private boolean onFloor:** Checks if the player is on floor.
* **private int health:** Refers to how much health our player has.

**Methods:**

* **public void move():** Moves the player according to user input received by the input manager
* **public void jump():**  Makes the player jump based on user input (normal difficulty) or ledgeCheck (hard difficulty).
* **public boolean ledgeCheck():** Checks if the player has moved over a ledge.
* **public void changeColor(color : int):** Changes the color of player.
* **public boolean floorCheck():** Checks if the player is on floor or not.

**Enemy class:**

****

**Attributes:**

* **private int health:** Refers to how much health an enemy has.
* **private int damage:** Refers to how much damage collision with an enemy will do to the player.
* **private int color:** Refers to the color (red, green, blue) of the enemy.
* **private int moveSpeed:** Refers to how fast the enemy moves in horizontal directions.
* **private int moveDir:** Refers to which horizontal direction the enemy object is moving at the moment.

**Methods:**

* **public void move():** Changes the location attributes of the enemy object based on moveSpeed and moveDir
* **public void changeDir():** Changes the direction to the opposite direction
* **public int getMoveDir():** Returns the enemies moveDir.

**Patroller class:** Enemy that walks left and right on the same platform by changing its direction when it reaches a ledge.

**Methods:**

* **private boolean leadgeCheck():** Returns true if the enemy object is right next to a ledge. Used for calling changeDir when a patroller approaches a ledge.

**Walker class:** Simple enemy that walks in the same direction until it collides with a wall.

**Jumper class:** Enemy that constantly jumps whenever it touches the floor.

**Attributes:**

* **private int jumpSpeed:** Variable that determines how high the jumper object will jump.
* **private bool onFloor:** Variable that checks if the player is currently on a platform directly .

**Methods:**

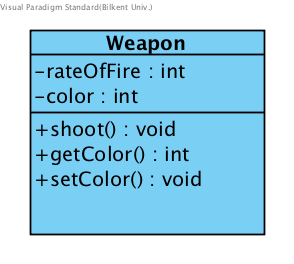
* **public bool checkFloor():** Function that checks whether the jumper object is directly on a tile.
* **public void jump():** Function that makes the jumper object “jump” if the player is directly on the floor.

**Floater class:** Enemy that goes back and forth between two (x,y) coordinates without being affected by gravity.

**Attributes:**

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

**Weapon class:**

****

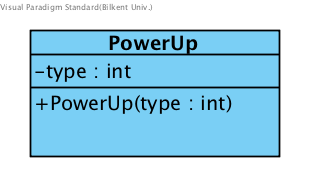
**Attributes:**

* **private int rateOfFire:** This attribute refers to the weapon’s rate of shooting.
* **private in color:** This attribute refers to the weapon’s color.

**Methods:**

* **public void shoot():** This method creates a bullet object in moving in the direction the weapon is facing.

**PowerUp class:**

****

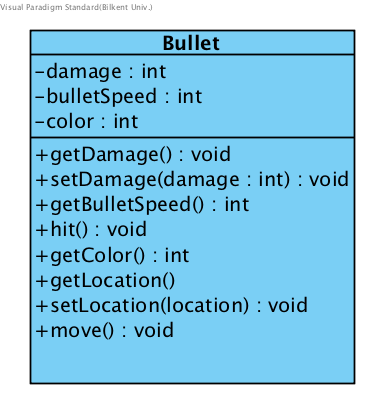
**Constructor:**

* **public PowerUp( type: int):** Constructs a power-up with type based on the parameter.

**Attributes:**

* **private int type:** Holds an integer that refers to the power-ups type.

**Bullet class:**

****

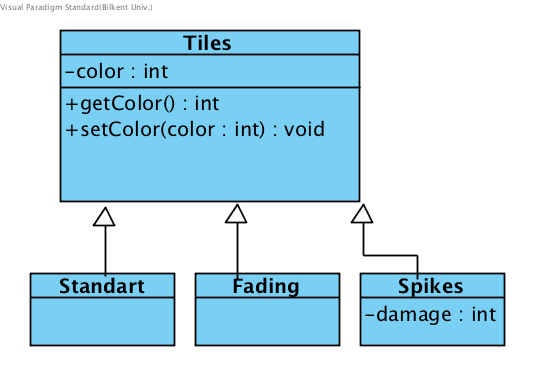
**Attributes:**

* **private int damage:** Refers to the amount of damage the bullet does on collision.
* **private int bulletSpeed:** Refers to the speed of which the bullet travels.
* **private int color:** Refers to the color of the bullet.

**Methods:**

* **private void hit():** It is called by the collision manager in the case of a collision between a bullet and the player/enemy. It is used for damage calculation and destruction of the bullet entity.

**Tiles class:**

****

**Attributes:**

* **private int color:** Refers to the color of tiles. Used for collision handling and sprites.

**Methods:** Only getters and setter methods.

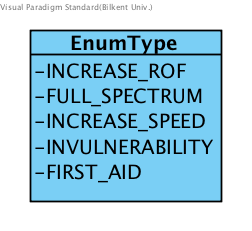
**StandardTile class:** Static tiles that the player can collide with.

**FadingTile class:** Tiles that will fade after the player has moved on them.

**Spikes class:**

* **private int damage:** This attribute is to calculate the damage done by the spikes to the player in the case of a collision.

**EnumType class:**

****

5 different types of powerups are provided to PowerUp class with enumeration (enum type).

**-INCREASE\_ROF**

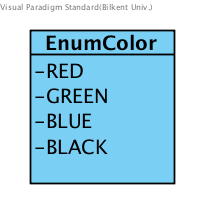
**-FULL\_SPECTRUM**

**-INCREASE\_SPEED**

**-INVULNERABILITY**

**-FIRST\_AID**

**EnumColor class:**



4 different types of powerups are provided to classes that use Color class with enumeration (enum type).

**-RED**

**-GREEN**

**-BLUE**

**-BLACK**