CS 319 - Object-Oriented Software Engineering

System Design Report

Battle City

Group 1-B

Kaan Sancak

Mahin Khankishizade

Ozan Kerem Devamlı

Teymur Bakhishli

Supervisor: Ugur Dogrusoz

Table of Contents

1. Introduction

1.1 Purpose of the system

Battle City is a 2-D tank destruction and attack game. Its design is implemented in such ways that, the users could get the possible maximum satisfaction from the game. Its difference from the original game is, the level ends when all the other enemy tanks are destroyed. Battle City is planned to be a portable, user-friendly and challenging game, which aims to entertain the users by involving them into the survivor game. The player's goal is to destroy all the enemy tanks in each level in order to pass to the next level. It aims to be enough reflexive, fast, user-friendly and with a high performance engine.

1.2 Design goals

Design is another important step for creating the system. It helps to identify the design goals for the system that we should focus on. As it was mentioned in our analysis report, there are many non-functional requirements of the system that needs to be better clarified in the design part. In other words, they are the object of the focus in this paper. Following sections are the descriptions of the important design goals.

1.2.1 Criteria

End User Criteria

Usability:

Battle City is expected to be an entertaining game for the users, which will provide them with a user-friendly interface, so that it was easy for the players to use

it. For example, the simple and understandable menu and interface of the game will help the player to focus on the game itself, rather than understanding the game functions. The system will get the keyboard inputs from the users, which is the easy use for the players.

Ease of Learning: It is obvious that, most of the players skip the 'how to play' part of the game, almost every time. However, since our system is created in such a way that, it was easy for the players to understand it even without the help of how to play screen. From the player's point of view, it will be very easy to identify and understand how the level is ended, when the tank fires, attacks, how it moves, how the power-up life bonuses effect the tank etc.

Performance:

Performance is one of the important design goals, which is needed for the games in general. We are using GUI library for the better enhancement of the performance.

Maintenance Criteria

Extendibility:

The design of the game lets us to modify and change its features and functionalities in the future depending on the user feedback or other reasons. For example, we can extend its challenges by adding the time span for each level, or the strength differing between enemies and the player's tanks.

Modifiability:

Battle City game is designed as multilayered. With this design structure, it is easy to modify the system. Since the subsystems are connected weakly, the modification in one subsystem does not affect the other one. Thus, in our system, it is easy to modify the existing functionalities.

Reusability:

There are some subsystems (i.e. GUI) which are possible to be used in other games, without any change. For their reusability, they were designed independent from the system.

Portability:

Portability is an important point for a software in general, because it enlarges the range of the game's usability for users. In other words, it helps the game to be played in various devices. Thus, we decided to implement in Java, since its JVM lets the platform to be independent and the system to be portable.

Performance Criteria

Response Time: It is important for the game to have the immediate reflex and reaction for the users' requests. Battle City game is designed so that its reactions were almost immediate, during the game itself, in display of the animations and effects.

1.3 Definitions

& Acronyms & Abbreviations

Java Virtual Machine (JVM) – an abstract computing machine for running a Java program.

Model View Controller (MVC) – the design pattern that we are going to use.

Graphic User Interface (GUI) – the library that we are going to code for the game.

2. System Architecture

2.1. Subsystem Decomposition

In this section, we will decompose our system into subsystems. During this decomposition, our purpose is to reduce the coupling between different subsystems of the main system and increasing coherence of the components. By decomposing the game system as described we can easily modify the game or extend it when it is needed.

During the decomposition of the system, we have decided that MVC(Model-View-Controller) architectural pattern is great fit to apply on our system. We divided our system based on MVC principles. Meaning that, we decided that our menu classes will be our view since they provide interface for users and our management classes will be our controllers since they control and maintain the game. Moreover, our object classes and map class will be representing our Model. We will go into more detail in continues parts.

As mentioned before, we divided our system into three subsystems which are represented in Figure-1. These three subsystems are User Interface subsystem, Game Management subsystem and Game Objects subsystem. We tried to decompose these systems according to their different functionalities. On the other hand, each subsystem has a responsibility to invoke other systems, so that game can stay maintainable during runtime. Meaning that, User Interface subsystem can only request a functional action to Gama Management subsystem by requesting menu class to invoke GameManager class. Therefore any interaction between User Interface subsystem and Gama Management subsystem must happen through GameManager class. This is known as Façade design pattern. It makes very easy to solve any errors that may occur also it makes the game more stabilized, modifiable and extendable.

Moreover, Game Management subsystem acts like a controller of the game. It has collision manager class which controls whether there is collision in the game and reports it to the GameManager class, input manager class which handles the user input of the game and MapManager class which updates the game according to the data coming from other

classes. As in the first case, the interaction between Game Management subsystem and Game Objects Subsystem occurs through Map class and Map Manager Class by using Façade's principles.

As a result, our system decomposition will provide us high cohesion and low coupling. Having a system as described will make "Battle City" more flexible and extendable game.

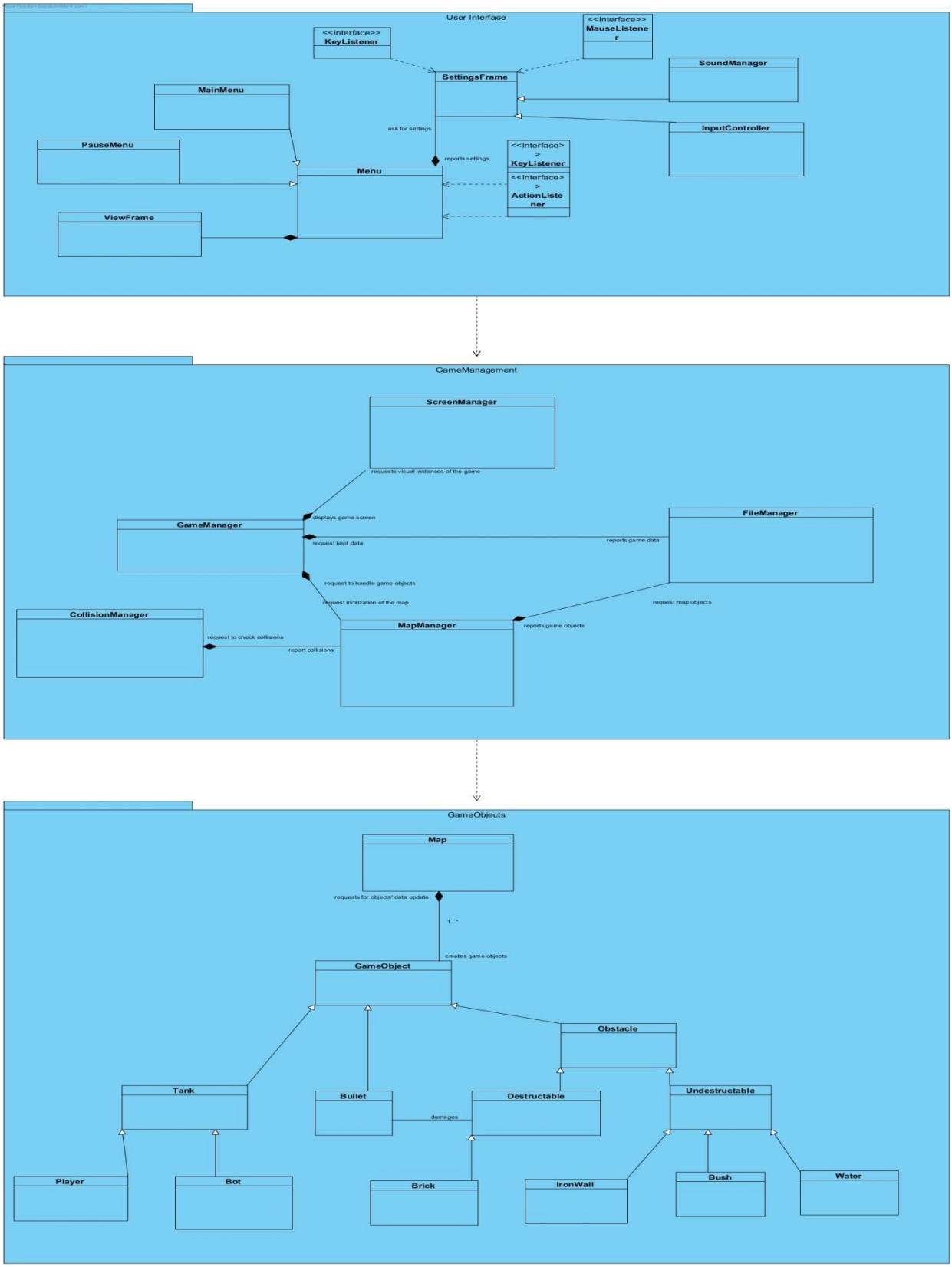

Figure 1- Basic Subsytem Decomposition

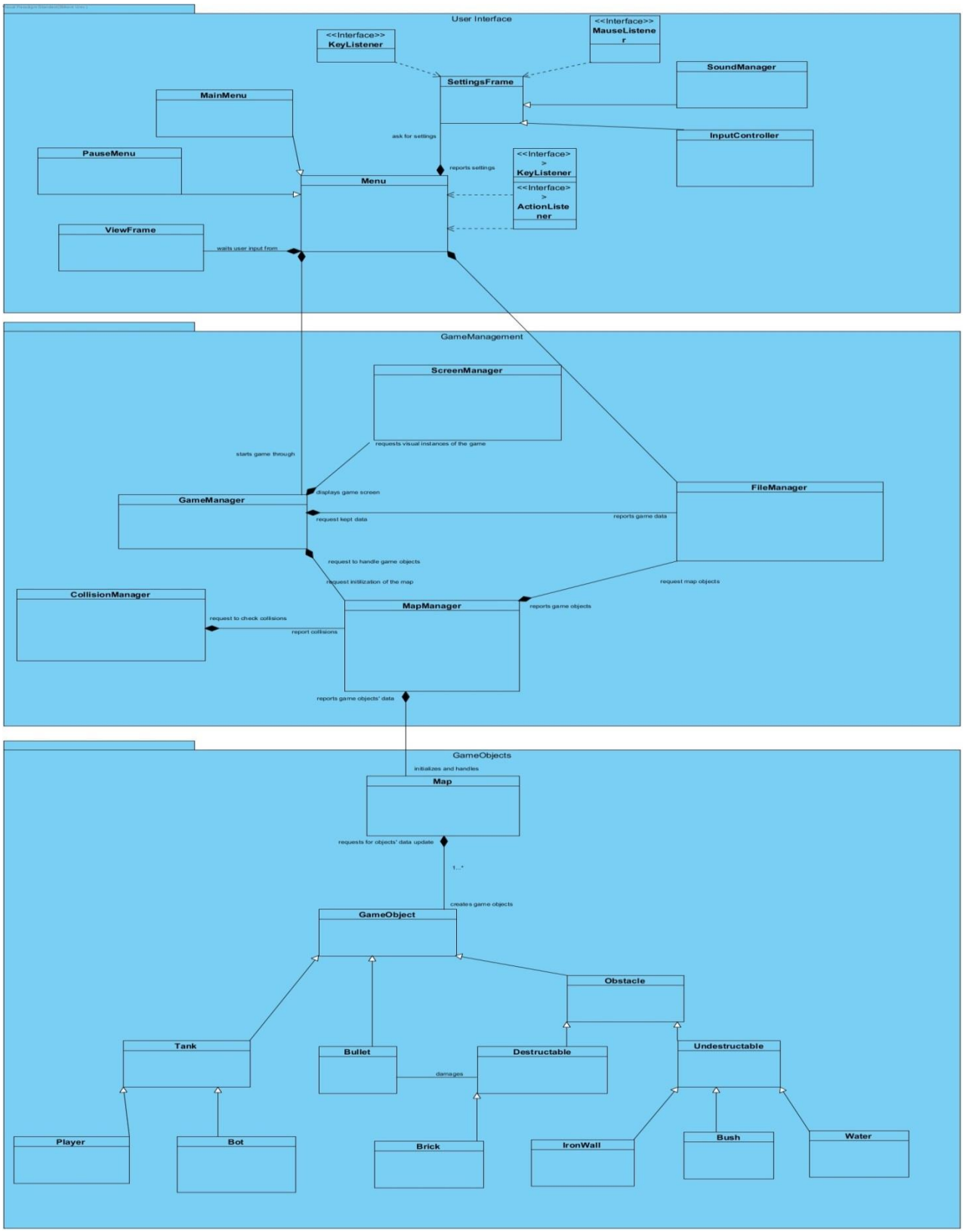

Figure 2-Detailed Subsytem Decomposition

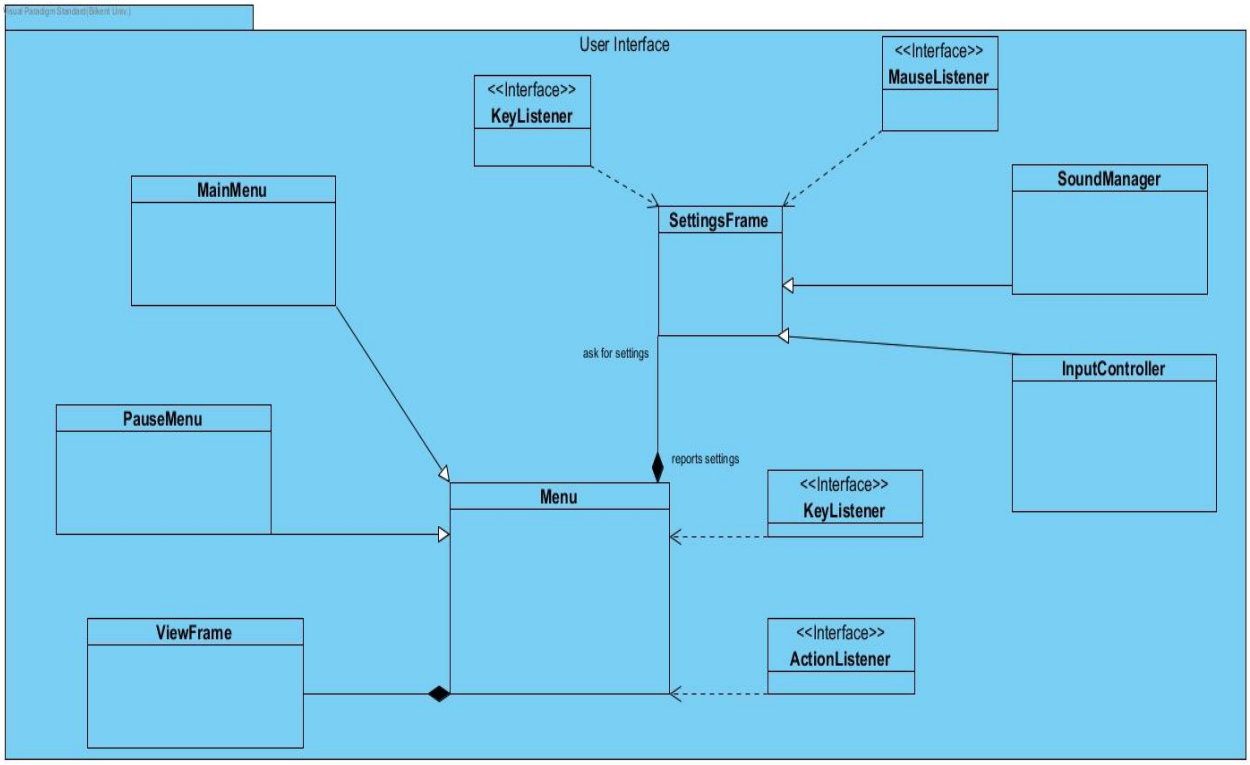

Figure 3 - User Interface Subsystem

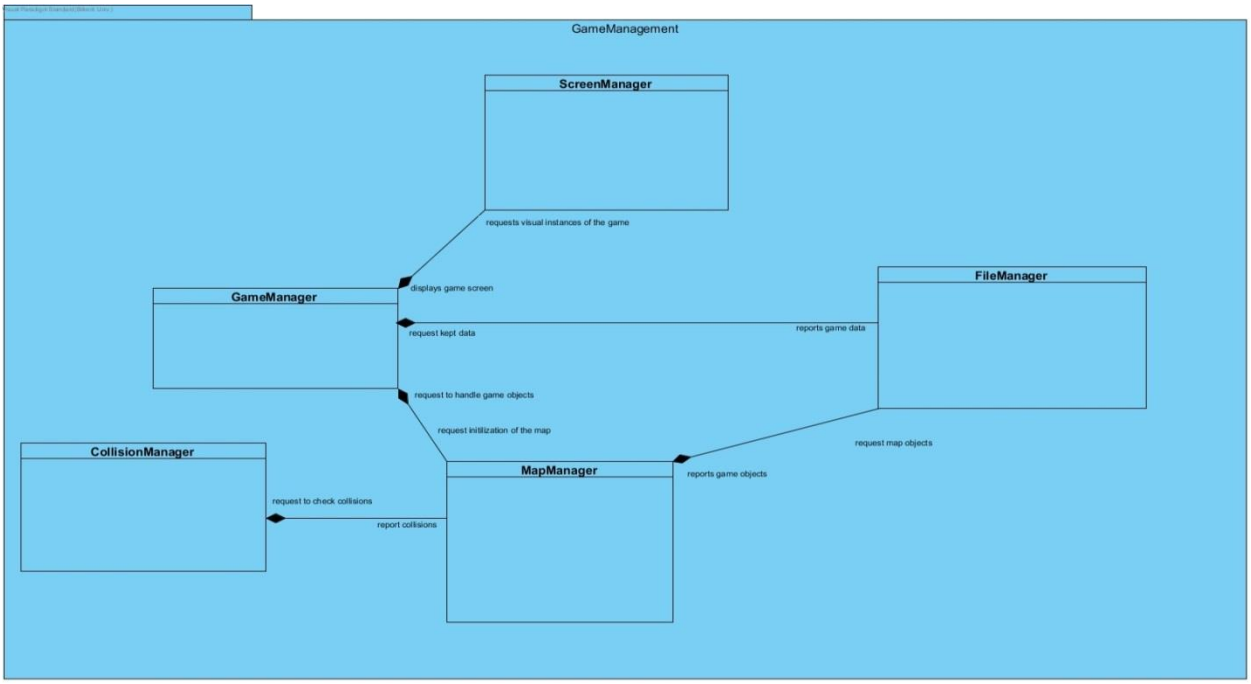

Figure 4- Game Management Subsystem

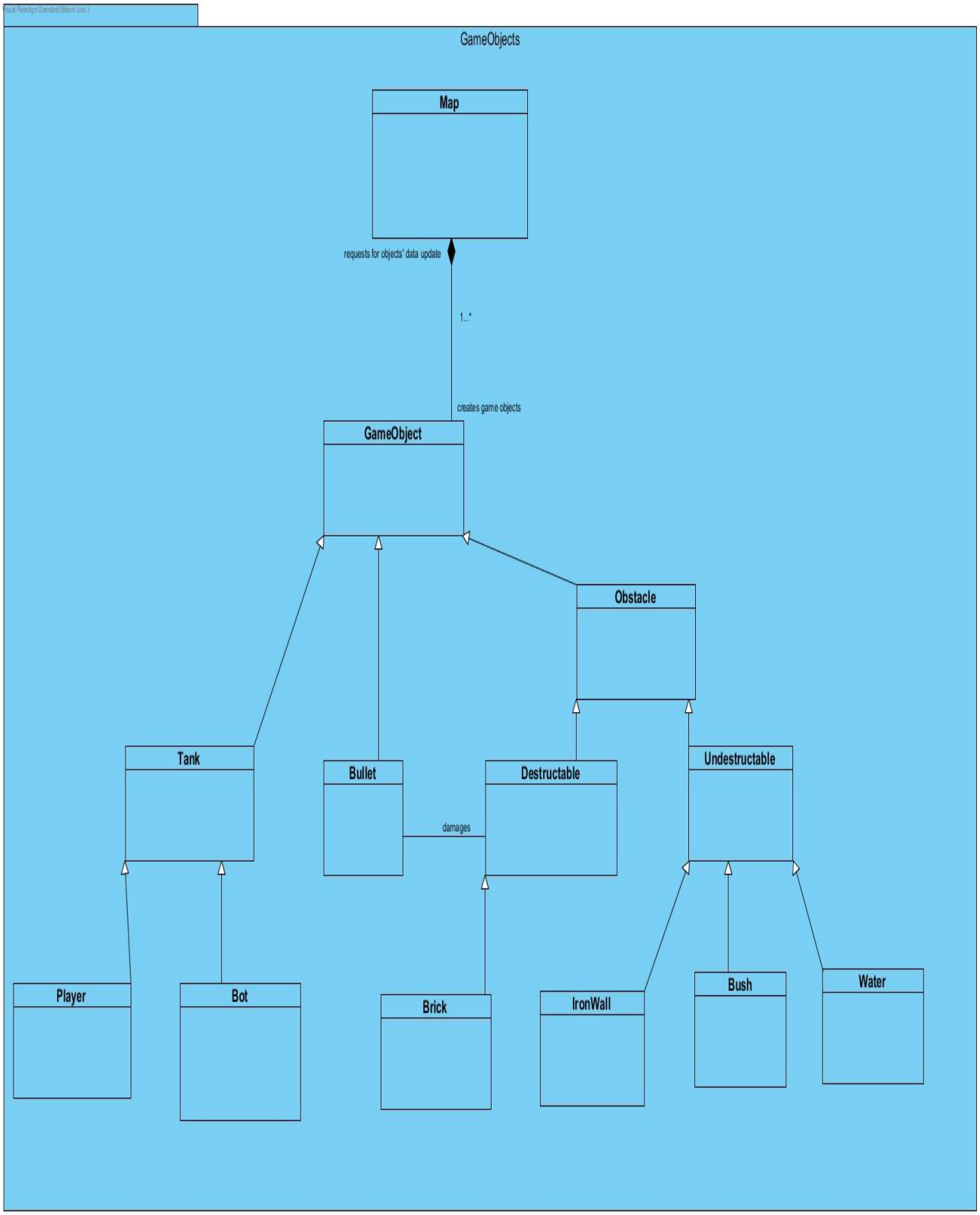

Figure 5 - GameObjects Subsytem

2.2. Hardware/Software Mapping

Battle City game will be implemented in Java programming language; more specifically we will use JAVAFX libraries. [1] That's why, as software requirement "Battle City" will require Java Runtime Environment in order to be executed by other users and also "Battle City" will require at least the Java Development Kit 8 or a newer version because these versions include JAVAFX libraries.

Moreover, "Battle City" will require a keyboard as a hardware requirement. Keyboard will be used for I/O tool of the game. Users will control the menu selections, tank movements and also fire interactions with keyboard inputs. As a results of these two requirements, our game's system requirements will be minimal, just a computer with necessary software installed will be enough to compile the game and run it.

For storage of the game, meaning storage of the maps, highest scores, user setting preferences, we will use text files. Therefore, "Battle City" will not require any internet connection or database to operate.

2.3. Persistent Data Management

Battle City does not require any complex data storage system or database. Battle City will store the game instances, like objects, maps etc. in hard drive of the client. Meaning that, we will keep the game data in text files. Some of the text files like map designs will be instantiated during the implementation stage and these files cannot be modified. On the other hand, some data members like user setting preferences, sound settings, and highest score will be kept in a modifiable text files, therefore user can change or update the values of these files during game time. Moreover, to store the sound data we will use .waw format and to store the images we will use .gif format.

2.4. Access Control and Security

As mentioned at section 2.2 and 2.3, Battle City will not use any internet/network connection or any database. Basically, after loading the game and starting Battle City,

anybody can play the game. Therefore, there will not be any access control and security measurement in order to prevent data leaks or any malicious actions. The highest scores will be unique for each computer; meaning that highest scores will not kept the highest score made in the all of the computers but it will be kept unique for each computer.

2.5. Boundary Conditions

Battle City will not require any installation; it will not have an executable .exe extension. Battle City game will have an executable jar and the will be executed from .jar. This will also bring portability to the game. The game can be copied from a computer to another computer easy and in a status that ready to play.

Battle City game can be terminated by clicking “Exit” button in the main menu. In another scenario where player wants to quit during game time, user should pause the game by pressing “p” from keyboard or clicking “pause” button from game screen. After that, user can terminate the game by clicking “Exit” button in the main menu.

If there will be error during the file read, the game may start without sound and images. This problem can be solved by validating data members of the game and fixing the corrupted text file data.

If game collapses during game time because of an performance or design issue, the data(settings/score) will be lost.

3. Subsystem Services

3.1. User Interface Subsystem

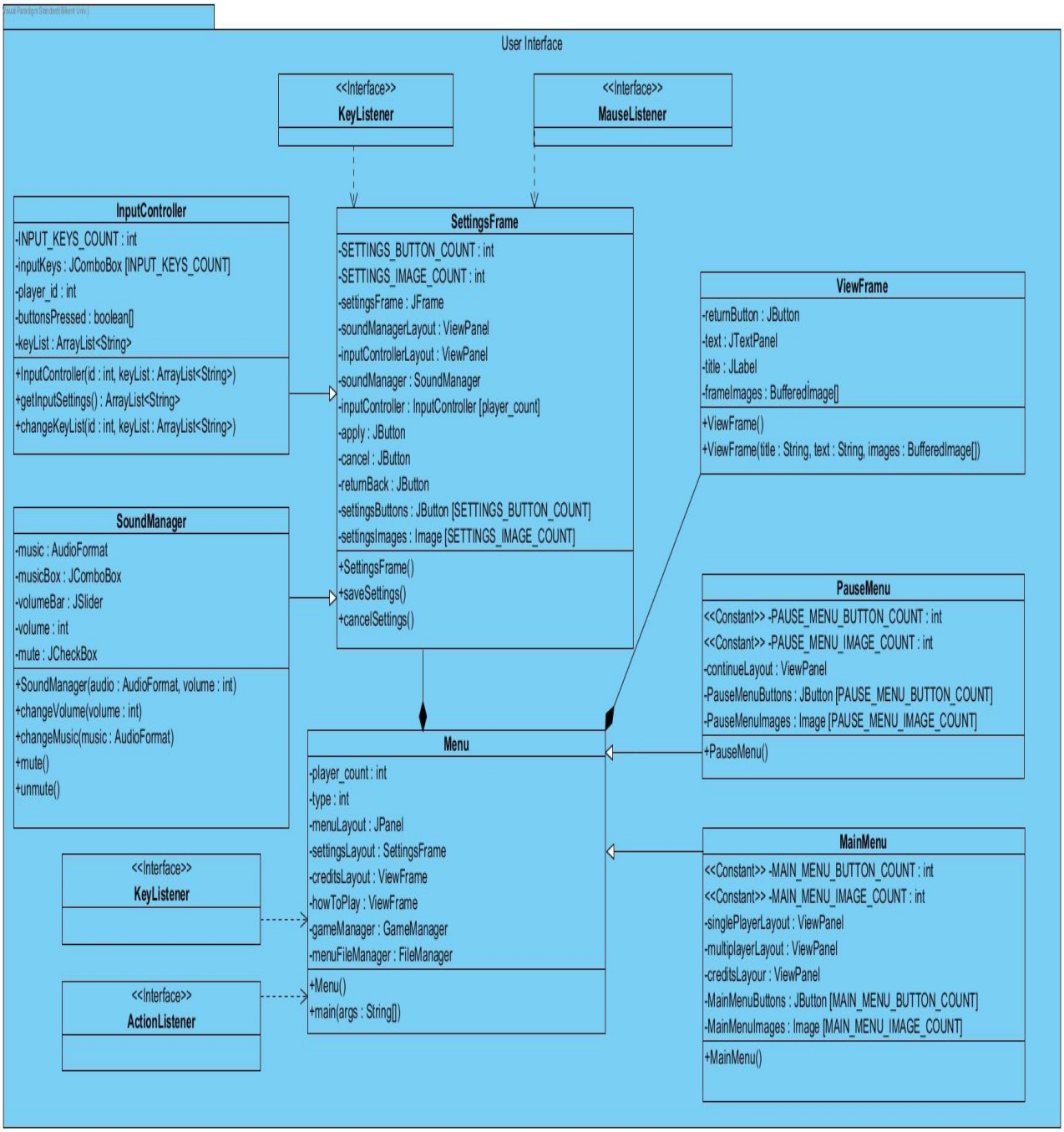

Figure 6-Detailed User Interface Subsystem

User Interface Subsystem consists of seven classes and is responsible for the establishment of the interface between the system and the user. “Menu” class is the Façade class of the User Interface Subsystem. Actually it is not totally Façade design pattern however as a functionality relation the only change in Game Management subsystem should have happen through Menu class, FileManager actually does not have an effect on functionality.

InputController and SoundManager classes can be achieved from SettingsFrame class, since their common methods and attributes are inside the SettingsFrame class. Similarly, PauseMenu and MainMenu classes can be achieved from the Menu class. ViewFrame class helps to display the classes mentioned above. Menu class is a façade class, because all the classes are linked to it and the user can access them from this class. The user can also access “How to Play”, “Credits” and “Quit” screens from here.

The user will see 6 buttons on the Menu display; single player, multiplayer, how to play, settings, credits and quit. Each screen will be initialized with the help of the instances of the classes mentioned before. For instance, by entering the Settings screen, the user will be able to adjust his/her settings with the help of changeKeyList(id : int, keyList : ArrayList<String>) method of the InputController, which gets the id from the Menu class.

3.1.1. Menu Class

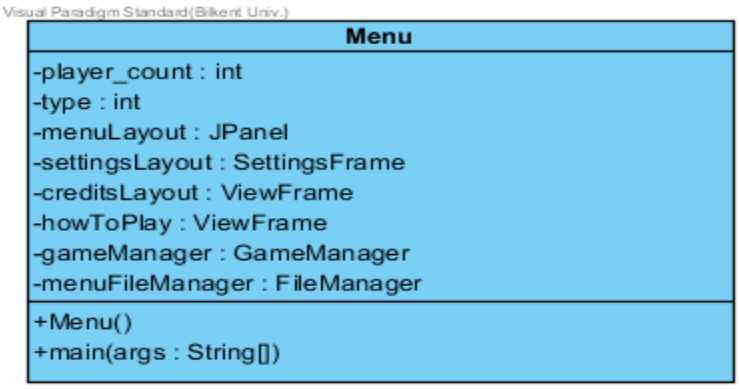

Figure 7-Menu Class

- **private int player_count:** This attribute will be used in the number of players.
- **private int type:** This attribute will be used in order to determine the type of menu(Single or Multi Player menu type)
- **private JPanel menuLayout:** This attribute will provide the Menu object layout in the Menu, via JPanel
- **private SettingsFrame settingsLayout:** This attribute will be used to create the frame for the Settings screen
- **private ViewFrame creditsLayout:** This attribute will be used to create the frame for the Credits screen
- **private ViewFrame howToPlay:** This attribute will be used to create the frame for the How to play screen
- **private GameManager gameManager:** This attribute will be used in order to call the gameManager object from GameManager class
- **private FileManager menuFileManager:** This attribute will be used in order to call the menuFileManager object from FileManager class

Constructors:

- **public Menu():** Default constructor for the Menu class.

Methods:

- **public void main(args: String[]):** main method of the Menu class to call the other functions and etc.

3.1.2. MainMenu Class

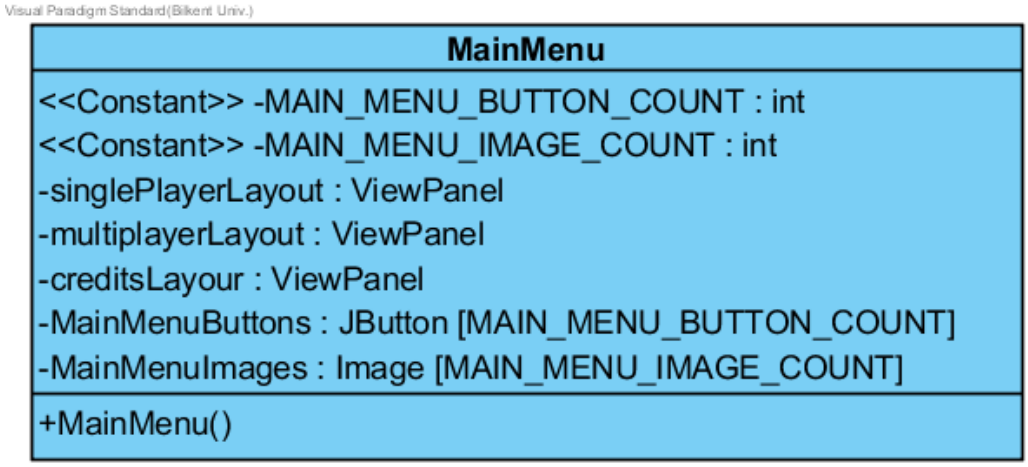

Figure 8-Main Menu Class

Attributes:

- **private final int MAIN_MENU_BUTTON_COUNT:** This attribute will be used in the creation of the JButton array to determine the count of the buttons the Settings screen should display
- **private final int MAIN_MENU_IMAGE_COUNT:** This attribute will be used in the creation of the Image array to determine the count of the images the Main menu screen should
- **private ViewPanel singlePlayerLayout:** This attribute will provide the SinglePlayer object layout in the MainMenu, via ViewPanel
- **private ViewPanel multiPlayerLayout:** This attribute will provide the MultiPlayer object layout in the MainMenu, via ViewPanel
- **private ViewPanel creditsLayout:** This attribute will provide the Credits object layout in the MainMenu, via ViewPanel

- **private JButton[] MainMenuButtons:** This attribute is the JButton array for the creation of the several buttons at the same time, which will be displayed in the MainMenu screen

- **private Image[] MainMenuImages:** This attribute is the Image array for the creation of the several images at the same time, which will be displayed in the MainMenu screen

Constructors:

- **public MainMenu():** Default constructor for the MainMenu class.

3.1.3. PauseMenu Class

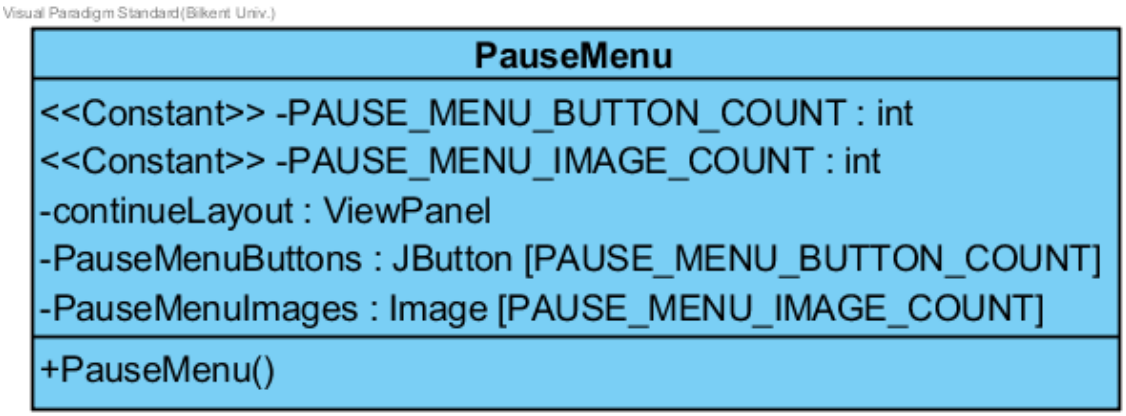

Figure 9-Pause Menu Class

Attributes:

- **private final int PAUSE_MENU_BUTTON_COUNT:** This attribute will be used in the creation of the JButton array to determine the count of the buttons the PauseMenu screen should display

- **private final int PAUSE_MENU_IMAGE_COUNT:** This attribute will be used in the creation of the Image array to determine the count of the images the PauseMenu screen should

- **private ViewPanel continueLayout:** This attribute will provide the Continue object layout in the PauseMenu, via ViewPanel
- **private JButton[] PauseMenuButtons:** This attribute is the JButton array for the creation of the several buttons at the same time, which will be displayed in the PauseMenu screen
- **private Image[] PauseMenuImages:** This attribute is the Image array for the creation of the several images at the same time, which will be displayed in the PauseMenu screen

Constructors:

- **public PauseMenu():** Default constructor for the PauseMenu class.

3.1.4. ViewFrame Class

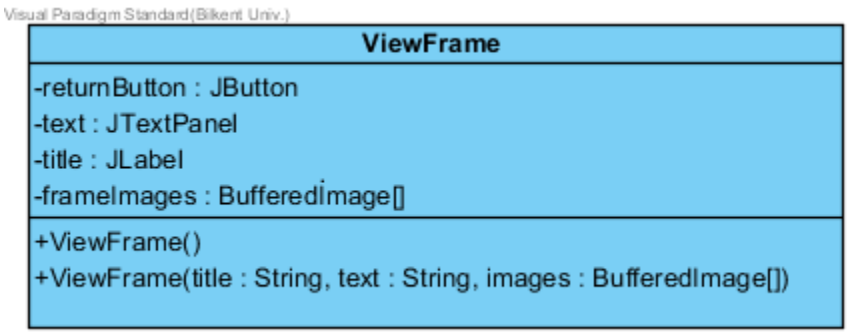

Figure 10- View Frame Class

Attributes:

- **private JButton returnButton:** This attribute provides a button in the frame which has a function to return the screen to previous frame. (E.g. In the how to play screen you can return the main menu by pressing the returnButton)
- **private JTextPanel text:** This attribute proves text field for the frame. The necessary texts will be taken by fileManager class and integrated into necessary frames.

- **private JLabel title:** This attribute provides title for the frame.(E.g. “How to Play”, “Credits” …)
- **private BufferedImage[] frameImages:** This array attribute keeps the images in the array so that when the frame is constructed. It takes them initializes on the screen.

Constructors:

- **public ViewFrame():** Default constructor for ViewFrame class. Basically initializes an empty Frame to be filled in.
- **public ViewFrame(title : String, text : String, images : BufferedİmageArray[]):** This constructor takes string title, string text and buffered image array images of a Frame and initializes a ViewFrame.

3.1.5. SettingsFrame Class

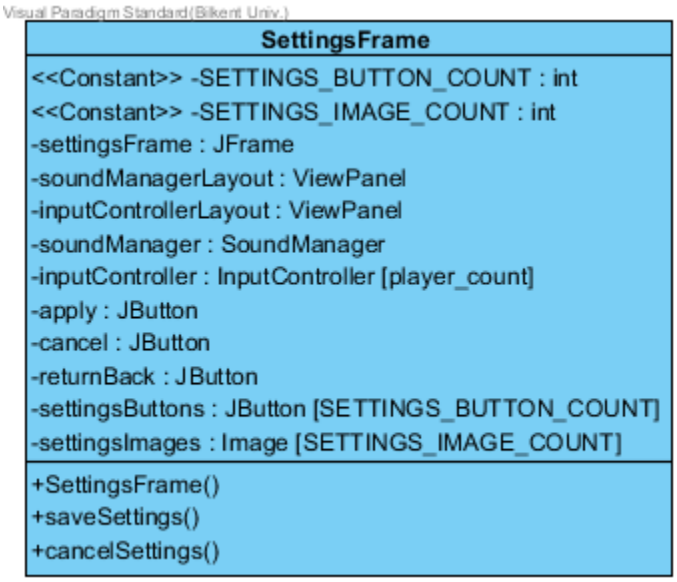

Figure 11-Settings Frame Class

Attributes:

- **private final int SETTINGS_BUTTON_COUNT:** This attribute will be used in the creation of the JButton array to determine the count of the buttons the Settings screen should display
- **private final int SETTINGS_IMAGE_COUNT:** This attribute will be used in the creation of the Image array to determine the count of the images the Settings screen should display
- **private JFrame settingsFrame:** This attribute will be used to create the frame for the Settings screen
- **private ViewPanel soundManagerLayout:** This attribute will provide the SoundManager object layout in the SettingsFrame, via ViewPanel
- **private ViewPanel inputManagerLayout:** This attribute will provide the InputController object layout in the SettingsFrame, via ViewPanel
- **private SoundManager soundManager:** This attribute is the instance of the SoundManager class in the SettingsFrame. It will provide the system to use SoundManager class' methods in the Settings screen
- **private InputController[] inputController:** This attribute is the instance of the InputController class in the SettingsFrame. It will provide the system to use InputController class' methods in the Settings screen. player_count determines the mode of the game (i.e. single or multi player – 1 or 2)
- **private JButton apply:** This attribute provides the user with the 'application' button, which aims to save the changes made in the Settings screen by the user
- **private JButton cancel:** This attribute provides the user with the 'cancellation' button, which aims to cancel and return to the default state of the settings by the user
- **private JButton returnBack:** This attribute provides the user to return to the Main screen from the Settings screen
- **private JButton[] settingsButtons:** This attribute is the JButton array for the creation of the several buttons at the same time, which will be displayed in the Settings screen
- **private Image[] settingsImages:** This attribute is the Image array for the creation of the several images at the same time, which will be displayed in the Settings screen

Constructors:

- **public SettingsFrame():** Default constructor for the SettingsFrame class, to fill the empty frame

Methods:

- **public void saveSettings():** This method saves the changed settings after apply button was pressed
- **public void cancelSettings():** This method cancels the changed settings and returns to the default state of the game after cancel button was pressed

3.1.6. SoundManager Class

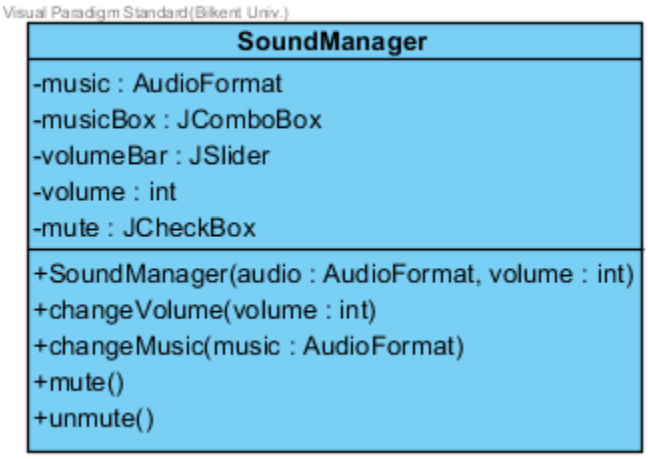

Figure 12 - Sound Manager Class

Attributes:

- **private AudioFormat music:** This attribute provides the user with the songs, which they can later choose from
- **private JComboBox musicBox:** This attribute provides the user with the list of the songs, and will be displayed on the screen, so that the users could choose music
- **private JSlider volumeBar:** This attribute provides the user with the slider, by which they can adjust the volume of the background music
- **private int volume:** This attribute keeps the current state of the volume inside it

- **private JCheckBox mute:** This attribute provides the user to mute or unmute the background music completely

Constructors:

- **SoundManager(audio: AudioFormat, volume: int):** Constructor with the parameters, which creates the SoundManager frame, chooses default music and current volume

Methods:

- **public void changeVolume(volume: int):** This method changes the volume when the user adjusts the volumeBar
- **public void changeMusic(music: AudioFormat):** This method changes the music when the user chooses new song from the musicBox
- **public void mute():** This method mutes the song when the user checks the mute check box
- **public void unmute():** This method unmutes the song when the user unchecks the mute check box

3.1.7. InputController Class

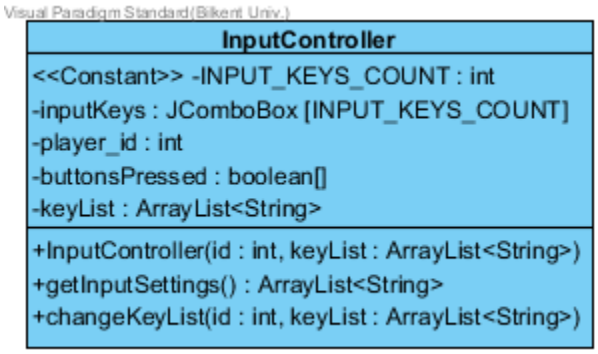

Figure 13 - Input Controller Class

Attributes:

- **private final int INPUT_KEYS_COUNT:** This attribute will be used in the creation of the JComboBox to determine the count of the keys which is possible to choose from
- **private JComboBox[] inputKeys:** This attribute is the JComboBox array which will keep the possible keys inside, and display them to the user in order to choose from them
- **private int player_id:** This attribute will keep the id of the player inside, so that when the user changes the keys of the particular player, it would not mess with the other player's keys
- **private boolean[] buttonsPressed:** This attribute is the boolean array, which keeps the track of the buttons whether they are pressed or not
- **private ArrayList<String> keyList:** This attribute will provide the user with the list of the keys

Constructors:

- **InputController(id: int, keyList: ArrayList<String>):** Constructor with the parameters, which creates the InputController frame, determines the player id and initializes the key list for the further implementation

Methods:

- **public void getInputSettings():** This method returns the ArrayList<String> and provides the class with the changed and new inputs for the keys
- **public void changeKeyList(id: int, keyList: ArrayList<String>):** This method changes the keys for the particular player, after user input the new keys

3.2. Game Management Subsystem

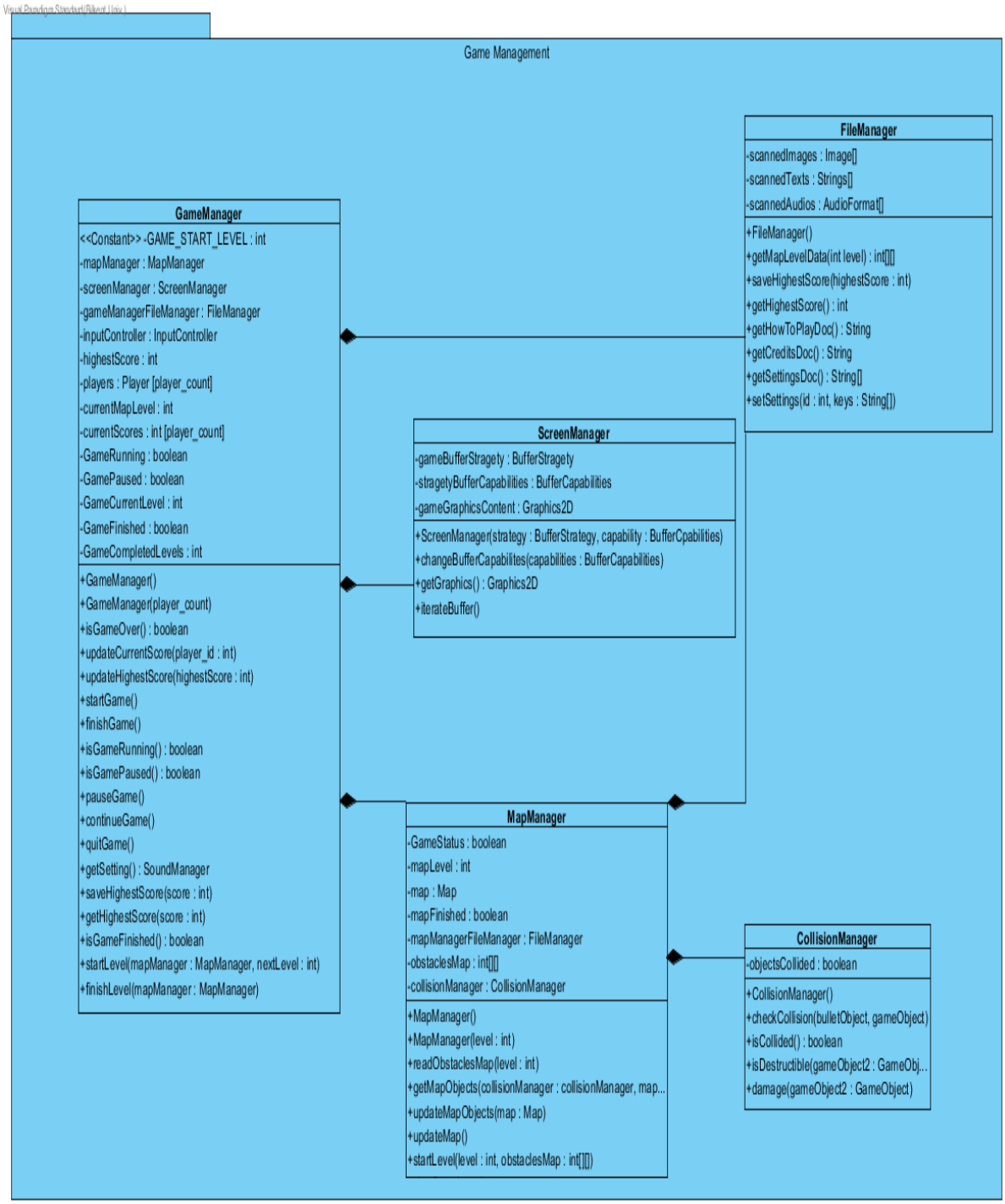

Figure 14-Game Management Subsystem

GMS consists of five classes and is responsible for the game logic and handling the game mechanisms such as collisions and display.

In this subsystem, GameManager is the main class which holds the game logic in itself. So, it holds the instances of ScreenManager, FileManager and MapManager classes in it. GameManager class reads the files via FileManager and determines the level, the map of the level, the objects on the map and passes it to MapManager class. Whereas via ScreenManager it displays the initialized game.

MapManager itself, manages the Map class, thus it is a façade class, which controls the class under it. MapManager holds the instance of CollisionManager in it, so that it could determine the collision and the type of it. For example, if the user destroys any brick or other destructible obstacle, the collision will be passed to MapManager via isCollided() method, and later MapManager will change the Map via updateMapObjects(map : Map) method.

3.2.1. GameManager Class

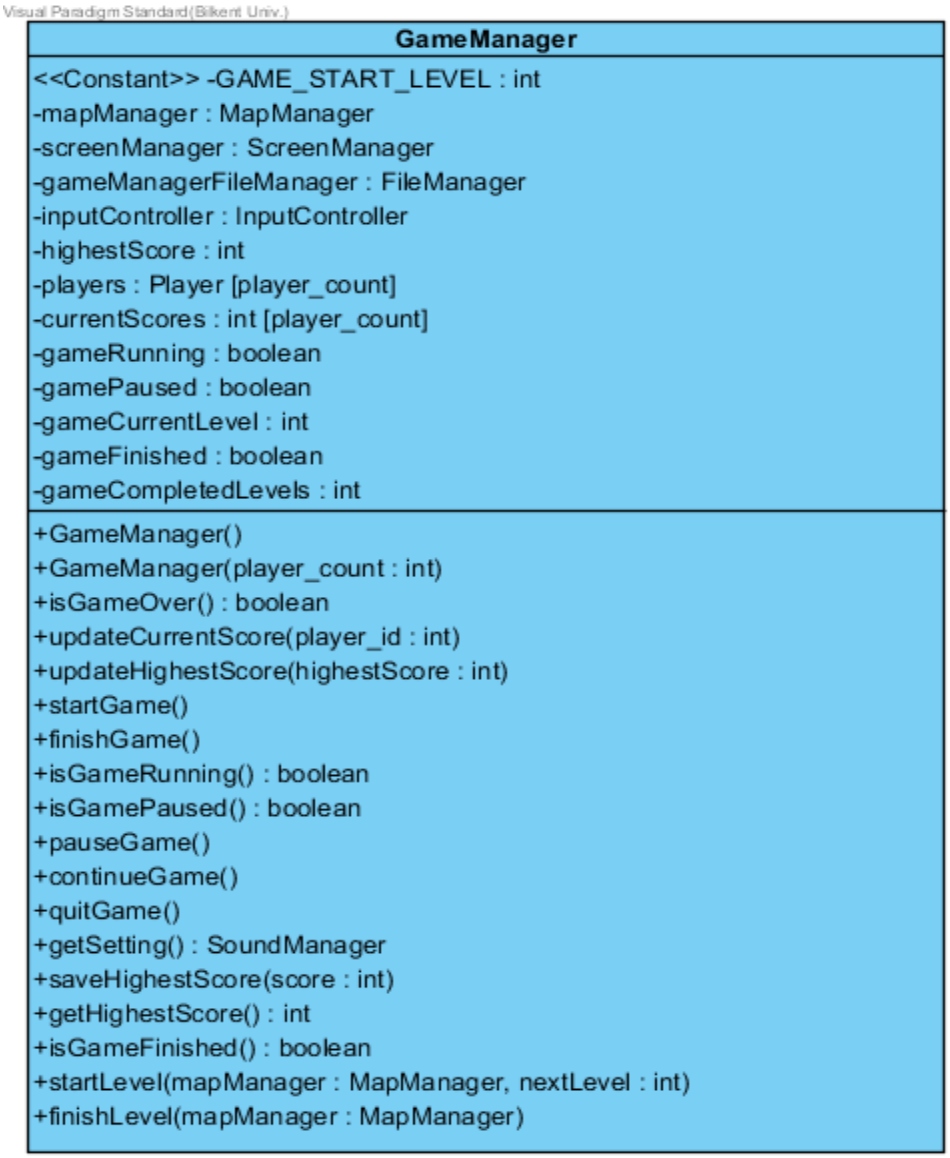

Figure 15 - Game Manager Class

Attributes:

- **private final int GAME_START_LEVEL:** This attribute provides default level count for the game, which is going to initialize as 1.
- **private MapManager mapManager:** This attribute is the instance of the MapManager class and provides the linkage between the logic of the game and map display

- **private ScreenManager screenManager:** This attribute is the instance of the ScreenManager class and provides the game display on the screen
- **private FileManager gameManagerFileManager**: This attribute is the instance of the FileManager class and provides the GameManager with the files from the FileManager
- **private InputController inputController**: This attribute is the instance of the InputController class and controls the user inputs during the game itself
- **private int highestScore**: This attribute keeps track of the highest score in the game
- **private Player[] players**: This attribute is an array of the Player class and aims to create the players with the identification number adjustment for each
- **private int[] currentScores**: This attribute is an integer array which keeps track of the scores for each player depending on their identification number
- **private boolean gameRunning**: This attribute keeps track of whether the game is running or not
- **private boolean gamePaused**: This attribute keeps track of whether the game is paused or not
- **private int currentGameLevel**: This attribute provides the game with the number of the current level the player is on, or passed to
- **private boolean gameFinished**: This attribute keeps track of whether the game is finished or not
- **private int gameCompletedLevels**: This attribute keeps the count of the finished and not lost levels for further calculation of the scores

Constructors:

- **GameManager():** Default constructor for the GameManager class, which simply initializes the class

- **GameManager(player_count : int):** This constructor determines whether the game is single or multiplayer game, then initializes the class

Methods:

- **private boolean isGameOver():** This method checks whether the game is over or not
- **private void updateCurrentScore(player_id : int):** This method updates the score of the particular player depending on the player_id
- **private void updateHighestScore(highestScore : int):** This method updates the highest score of the game
- **private void startGame():** This method simply starts the game
- **private void finishGame():** This method simply finishes the game
- **private boolean isGameRunning():** This method checks whether the game is currently running or not
- **private boolean isGamePaused():** This method checks whether the game is currently paused or not
- **private void pauseGame():** This method simply pauses the game
- **private void continueGame():** This method simply continues the game, if it was paused before
- **private void quitGame():** This method simply quits the game
- **private SoundManager getSetting():** This method gets the adjusted or default settings by returning the instance of the SoundManager class
- **private void saveHighestScore(score: int):** This method saves the highest score of the game

- **private int getHighestScore():** This methos provides the game with the highest score by returning it
- **private boolean isGameFinished():** This method checks whether the game is finished or not
- **private void startLevel(mapManager: MapManager, nextLevel: int):** This method starts the next level, if the user passed it or finished the level
- **private void finishLevel(mapManager: MapManager):** This method finishes the level.

3.2.2. MapManager Class

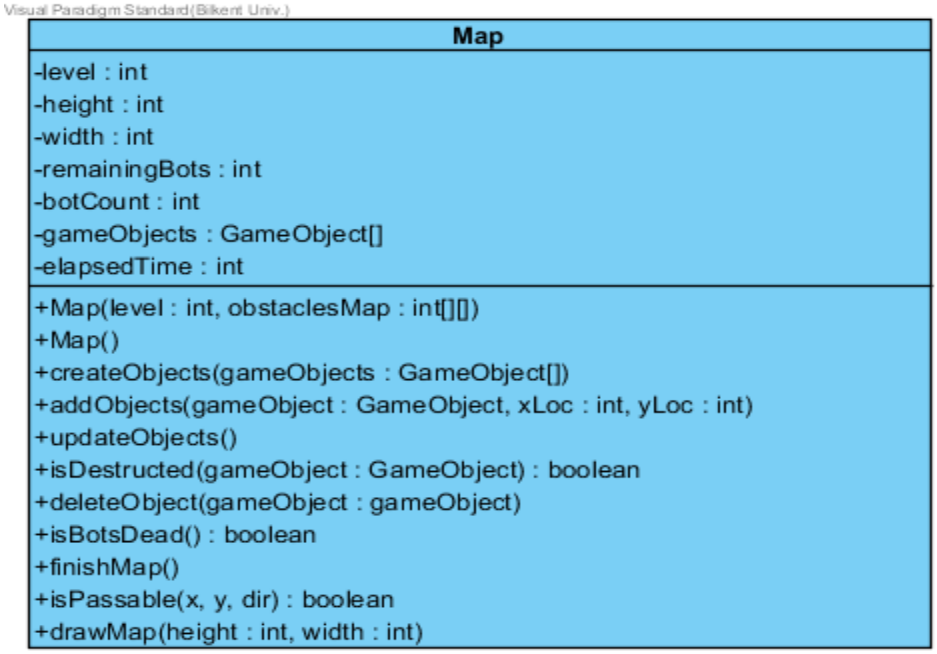

Figure 16 - Map Class

Attributes:

- **private boolean gameStatus**: This attribute saves the current status of the game (i.e running or paused etc.)
- **private int mapLevel**: This attribute keeps the count of the level the player is on
- **private Map map**: This attribute is the instance of the Map class

- **private boolean mapFinished**: This attribute keeps the track of whether the creation of the map is finished or not
- **private FileManager mapManagerFileManager**: This attribute is the instance of the FileManager class and provides the MapManager with the files from the FileManager
- **private int[][] obstaclesMap**: This attribute is the two dimensional array which keeps the count and type of the obstacles in the map
- **private CollisionManager collisionManager:** This attribute is the instance of the CollisionManager

Constructors:

- **MapManager():** Default constructor which initializes the class
- **MapManager(level : int):** This constructor initializes the map with the specific level count

Methods:

- **private void readObstaclesMap(level : int):** This method reads the count and types of the obstacles for the specific level
- **private Objects[] (collisionManager : CollisionManager, map : Map):** This method returns the instances of objects depending on the map
- **private void updateMapObjects(map : Map):** This method updates the map after each collision
- **private void updateMap():** This method updates the map for each level
- **private void startsLevel(level : int, obstaclesMap : int[][]):** This method starts the new level with the specific level count and the obstacles read before
- **private void stopGameLoop():** This method stops the game loop
- **private void gameLoop():** This method makes the game to continue by each movement
- **private boolean isMapFinished(completedLevels : int):** This method checks whether the map is finished initializing
- **private void finishLevel():** This method finishes the level

3.2.3. CollisionManager Class

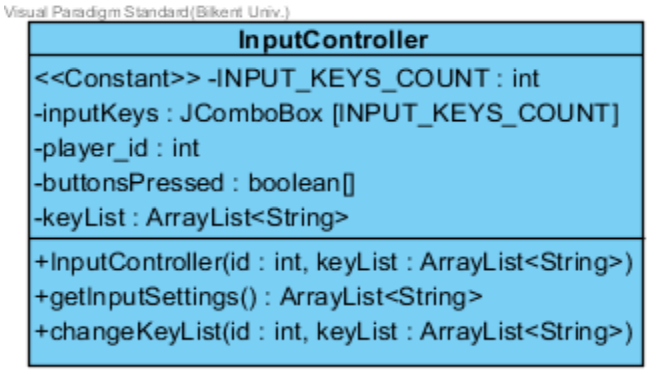

Figure 17-Input Controller Class

Attributes:

- **private boolean objectsCollided:** This attribute will be used to determine whether the collision is happened or not.

Constructors:

- **CollusionManager():** Default constructor of the CollusionManager class.

Methods:

- **public void checkCollision(bulletObject, GameObject):** This method will implement the process of collision.
- **public boolean isCollided():** This method will return true if there is a collision.
- **public boolean isDestructible(gameObject2: GameObject):** This method will return whether the collided object is destructable or not.
- **public void damage(gameObject2 : GameObject):** This method will damage the game object if there is a collision which any of objects might be damaged.

3.2.4. ScreenManager Class

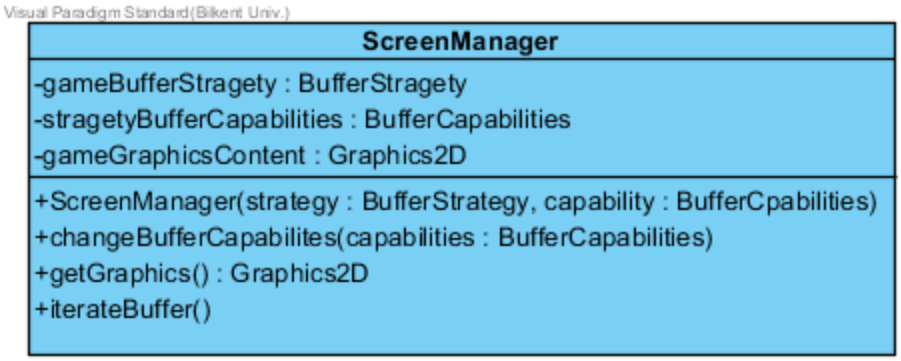

Figure 18-Screen Manager Class

Attributes:

- **private BufferStragety gameBufferStrategy:** This attribute keeps the buffer strategy of the screen manager which organizes complex buffers on the window/frame.
- **private BufferCapabilities stragetyBufferCapabilities:** This attribute keeps the capabilities of the buffer strategy which determines the data strategy in more detail.
- **private Graphics2D gameGraphicContent:** This attribute keeps the current graphics by using JAVA's 2D graphics library which uses GPU acceleration if needed.

Constructor:

- **public ScreenManager(strategy : BufferStrategy, capability : BufferCapabilities):** This constructor take a BufferStrategy and BufferCapabilities and initializes a Screen Manager which provides a user interface to users.

Methods:

- **public void changeBufferCapabilities(BufferCapabilities capabilities):** This method changes current buffer capabilities with new buffer capabilities in the case of any change. These changes might be caused by game progress.
- **public Graphics2D getGraphics():** This method return the current 2D Graphics.

- **public void iterateBuffer():** This method iterates the graphic buffers. Meaning that it initializes the next buffer when the old one is being displayed and iterates the new one.

3.2.5. FileManager Class

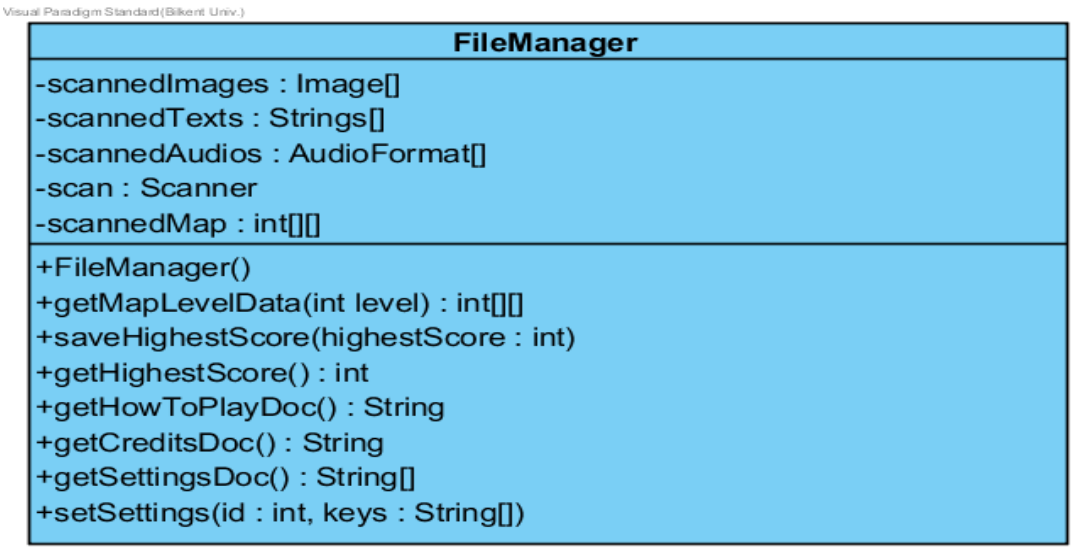

Figure 19-File Manager Class

Attributes:

- **privete Images[] scannedImages :** This attribute keeps the scanned images in an Image array to send the images to the draw methods of objects.
- **private String[] scannedTexts :** This attribute keeps the scanned texts in a String array, so that when it is needed texts won't be read from files second time but they will be reached from array.
- **private AudioFormat[] scannedAudios:** This attribute keeps the scanned audios for music options of the game.
- **private Scanner scan:** This attribute provider a scanner for FileManager class.

- **private int[][] scannedMap**: This attribute keeps the map and object orientation in its two dimensional array.

Constructor:

- **FileManager():** The constructor which creates a new FileManager object.

Methods:

- **public int[][] getMapLevelData(int level):** This method gets the map's level data from mapLevelData text file by taking map level as a parameter and assigns it to scannedMap attribute.

- **public void saveHighestScore(int highestScore):** This method gets the highest score from the GameManager and saves it to highest score text file.

- **public int getHighestScore():** This method gets the highest score from highest score text file.

- **public String getHowToPlayDoc():** This method gets the how to play documents from "How To Play" text file and assigns it to ViewFrame instance to represent "How To Play" screen.

- **public String getCreditsDoc():** This method gets the credits from credits text file documents and assigns it to ViewFrame instance to represent "Credits" screen.

- **public String[] getSettingsDoc():** This method gets the player input settings from settings text file and assigns it to a String array. Each element in the array represents an input key.

- **public void setSettings(int id, String[]):** This method gets the changed input settings from SettingsFrame and saves the new input settings to settings text file. Specifically, "Battle City" game has two text files for settings text file since there is a multiplayer option in the game.

3.3. Game Objects Subsystem

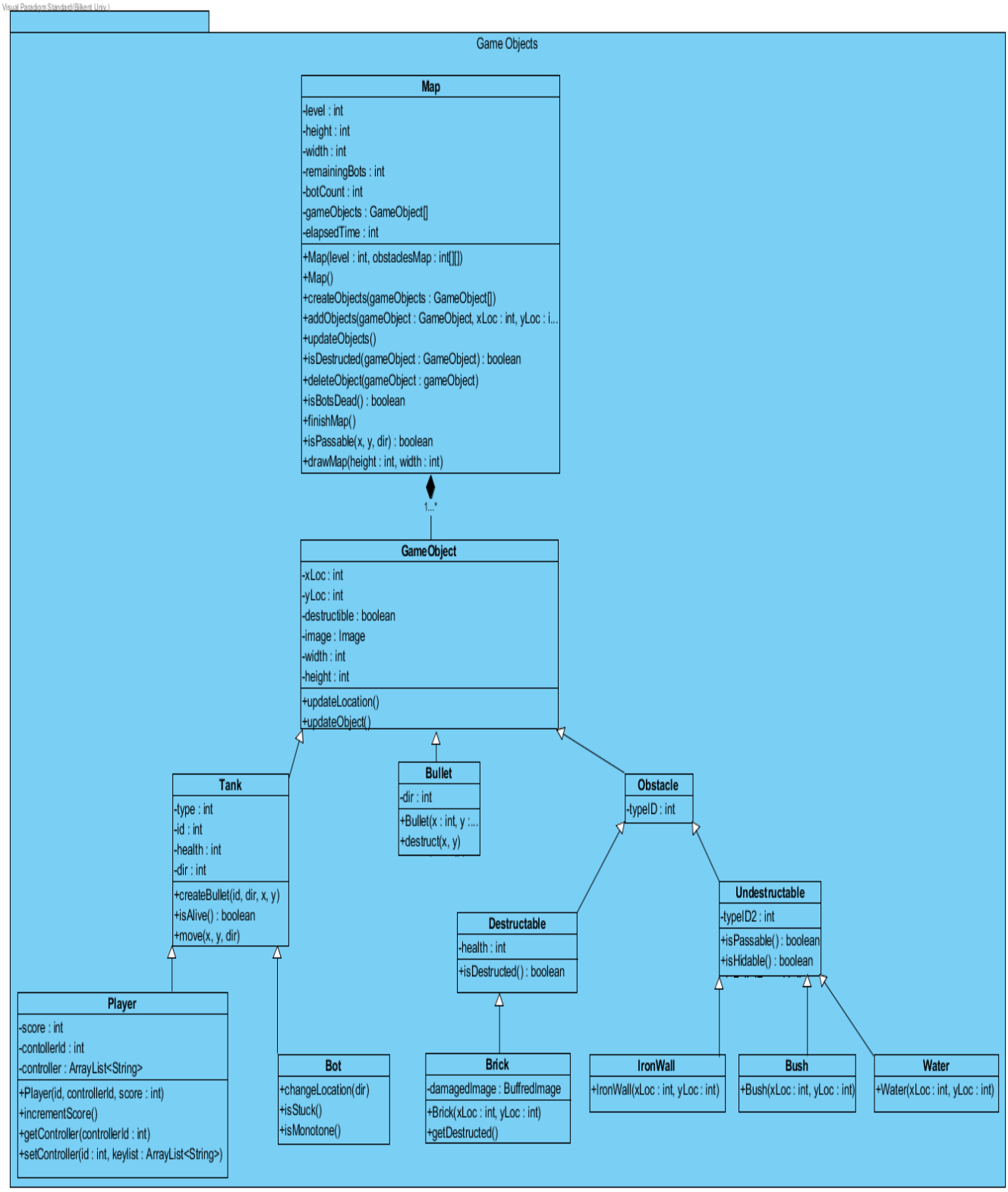

Figure 20-Game Management Subsystem

Game Objects Subsystem consists of thirteen classes and is responsible for the display of the game's main objects on the screen. The main objects of the game are tanks (players and bots) obstacles: bricks, bushes, rivers (or water), iron walls and bullets. According to the objects the user is going to see on the screen, the subsystem holds the classes of them inside respectively: Tank – Player and Bot, Bullet, Destructible – Brick, Undestructable – IronWall, Bush and Water.

Main classes as Tank, Bullet and Obstacle are linked to the GameObject class which saves the frame for the map, linked to the Map class which holds all the objects and keeps track of their movements inside. For example, if the user destroys the brick, the location of the obstacle will be sent to GameObject class, and passed to Map class, which will later update the map and objects with the updateObjects() method.

3.3.1. Map Class

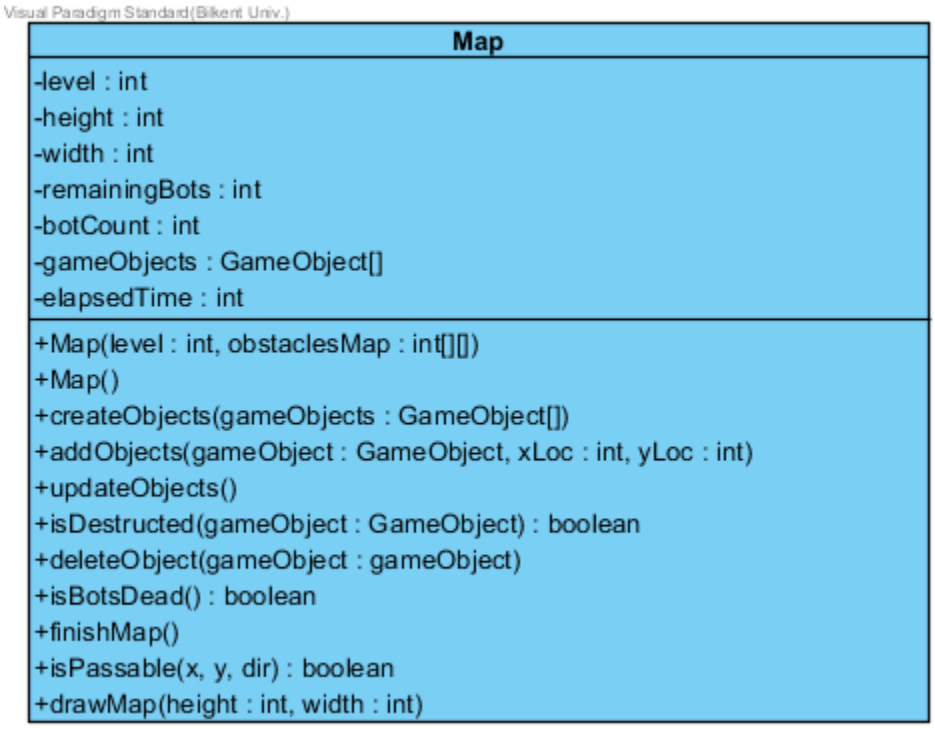

Figure 21 - Map Class

Attributes:

- **private int level:** This attribute will hold the map level.

- **private int height:** This attribute will be used to determine map size.
- **private int width:** This attribute will be used to determine map size.
- **private int remainingBots:** This attribute will hold the remaining bots in the current level.
- **private int botCount:** This attribute will hold the total number of bots for this level.
- **private GameObject[] gameObjects:** This attribute is the GameObject array, which holds the every object on the current map.
- **private double elapsedTime:** This attribute will hold the elapsed time during the game.

Constructors:

- **Map():** Default constructor of the Map class.
- **Map(level: int, obstaclesMap: int[][]):** This is the constructor which creates the new map with the level value.

Methods:

- **public void createObjects(gameObjects: GameObject[]):** This method creates the objects on the map.
- **public void addObjects(gameObject: GameObject, xLoc: int, yLoc: int):** This method adds objects on the current map with locations provided.
- **public void updateObjects():** This method will update all objects on the map.
- **public boolean isDestructed(gameObject: GameObject):** This method decides whether a game object is destructed by a bullet or not.
- **public void deleteObject(gameObject: GameObject):** This method removes an object from map according to the info coming from isDestructed(gameObject: GameObject) method(whether it is destructed or not).
- **public boolean isBotsDead():** This method checks whether all bots are dead.
- **public void finishMap():** This method finishes the map if all bots of current level is dead and invokes new Map.
- **public boolean isPassable(x: int, y: int, dir: int):** This method decides whether the movement of a tank or bullet is possible or not.

- **public void drawMap(height: int, width: int):** This method applies the Graphical User Interface to the map which will be displayed.

3.3.2. GameObject Class

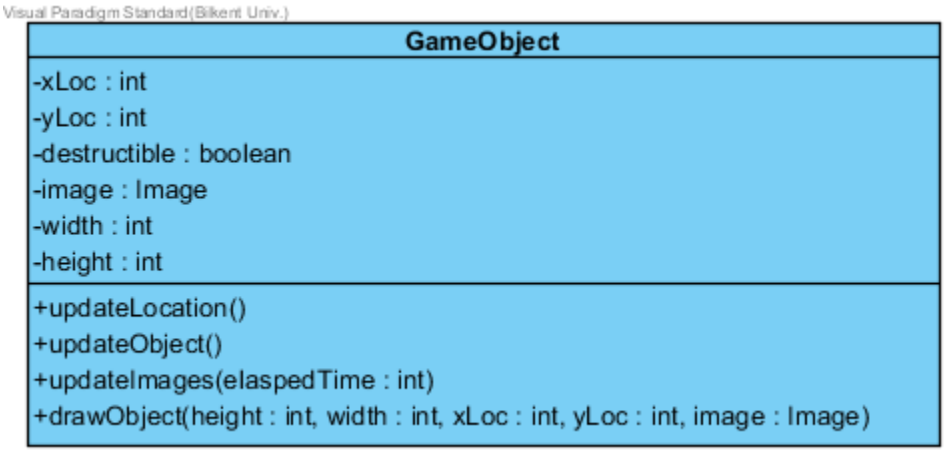

Figure 22- Game Object Class

Attributes:

- **private int xLoc:** This attribute holds the left point of the object.
- **private int yLoc:** This attribute holds the top point of the object.
- **private boolean destructible:** This attribute holds the behaviour of object against bullets.
- **private Image image:** This attribute holds image file for objects.
- **private int width:** This attribute holds the width of the object.
- **private int height:** This attribute hold the height of the object.

Methods:

- **public void updateLocation():** This method changes the x and y values of the object.
- **public void updateObject():** This method changes the width and height of the object.
- **updateImages(elapsedTime: int):** Updates the object's location with given frequency.
- **public void drawObject(height: int, width: int, xLoc: int, yLoc: int, image: Image):** This method draws image of the object in pixels calculated by position and size of the object.

3.3.3. Tank Class

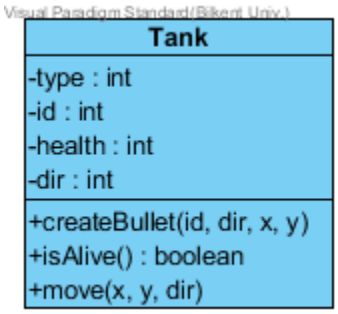

Figure 23 - Tank Class

Attributes:

- **private int type:** This attribute holds the type of the tank which could be player or bot
- **private int id:** This attribute holds the id of the tank, which determines the owner of the bullets.
- **private int health:** This attribute holds the current health of the tank.
- **private int dir:** This attribute determines the direction of the bullet and image(front of tank).

Methods:

- **public void createBullet(id: int, dir: int, x: int, y: int):** This method creates a bullet object.
- **public boolean isAlive():** This method determines whether the tank is dead.
- **public void move(x: int, y:int, dir: int):** This method requests the movement on map.

3.3.4. Player Class

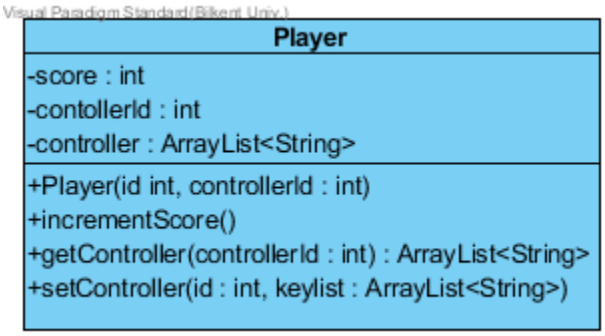

Figure 24 - Player Class

Attributes:

- **private int score:** This attribute holds the score of the player, which is incremented by killing bots and finishing levels.
- **private int controllerId:** This attribute holds the player ID.
- **private ArrayList<String> controller:** This String ArrayList holds the users current control keys for movement and fire bullet.

Constructors:

- **Player(id: int, controllerId: int):** The constructor of player class determines ID of the players.

Methods:

- **public void incrementScore():** This method updates the score of player when a level finishes or player kills a bot.
- **public void ArrayList<String> getController(controllerId: int):** This method return the controller input keys based on player's id.
- **public void setController(id: int, keylist: ArrayList<String>):** This method changes the controll keys of the given player.

3.3.5. Bot Class

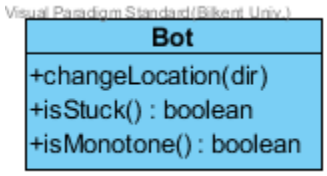

Figure 25 - Bot Class

Methods:

- **public void changeLocation(dir: int):** This method changes the x or y value of tank of bot depend on the direction.
- **public void boolean isStuck():** This method determines whether bot is trying to move on an unpassable object. If it is, method changes the direction.
- **public void boolean isMonotone():** This method determines whether bot is moving the same direction for a long time. If it is, method changes the direction.

3.3.6. Bullet Class

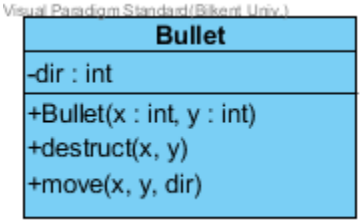

Figure 26 - Bullet Class

Attributes:

- **private int dir:** This attribute holds the owner of the bullet.

Constructors:

- **Bullet(x: int, y: int, dir: int):** Constructor of the bullet. Creates a bullet from the given position through to given direction.

Methods:

- **public void destruct(x: int, y: int):** This method causes destruction on the collision point which will harm every destructible object at that point.
- **public void move(x: int, y: int, dir: int):** This method provides the movement logic for the bullet.

3.3.7. Obstacle Class

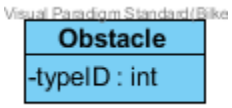

Figure 27 - Obstacle Class

Attributes:

- **private int typeID:** This attribute determines the type of the obstacle.

3.3.8. Destructible Class

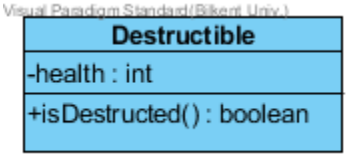

Figure 28 - Destructible Class

Attributes:

- **private int health:** This attribute holds the health of the destructible object.

Methods:

- **public boolean isDestructed():** This attribute determines if the health of the object is less than or equal to 0.

3.3.9. Brick Class

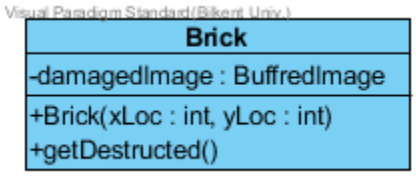

Figure 29 - Brick Class

Attributes:

- **private BufferedImage damaged:** This attribute holds the damaged image of brick. Each damage changes the image.

Constructors:

- **Brick(xLoc: int, yLoc: int):** Constructor of brick object, which not passes the tanks, can be destructed by bullets. Default health of the brick is determined in this constructor.

Methods:

- **public void getDestructed():** This method removes the object if if isDestructed() is true.
- **public void getDamage():** This method decreases the current health of the brick.

3.3.10. Indestructible Class

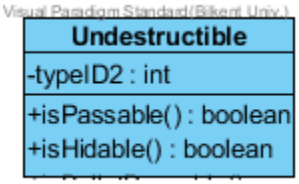

Figure 30 – Undestructible

Attributes:

- **private int typeID2:** This attribute determines the type of undestructible obstacle.

Methods:

- **public boolean isPassable():** If the obstacle is bush, this method returns true.
- **public boolean isHideable():** If the obstacle is bush, this method returns true.

- **public boolean isBulletPassable():** If the obstacle is water, this method returns true.

3.3.11. IronWall Class

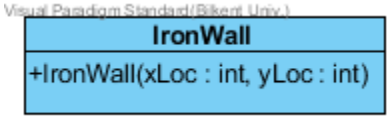

Figure 31 - Iron Wall Class

Constructors:

- **IronWall(xLoc: int, yLoc: int):** Constructor of IronWall object, which passes nothing.

3.3.12. Bush Class

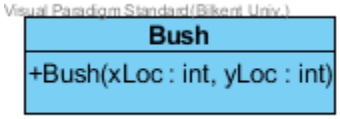

Figure 32 - Bush Class

Constructors:

- **Bush(xLoc: int, yLoc: int):** Constructor of bush object.

3.3.13. Water Class

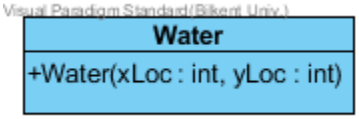

Figure 33 - Water Class

Constructors:

Water(xLoc: int, yLoc: int): Constructor of water object, which passes bullets but not tanks.

4. Low-level Design

4.1.Object Design Trade-Offs

Understandability vs Functionality:

As it was mentioned before, our system is expected to be easy to learn and understandable game. In respect to this, we had to decrease the complexity and functionality of the game by eliminating confusing and complicated options (i.e. too many options for power-ups, many options for keyboard input etc.) so that, the players would not bother to understand them, but would enjoy the simple understandable game.

Memory vs Maintainability:

During the design and the analysis of the game, we tried the factor out the common parts of the game objects and make use of abstraction as much as possible. Our main goal in this design thinking was to maintain the game objects easily as possible. However, by doing that we some object will have some methods and attributes which they do not need to have. This unnecessary attributes will cause memory allocation more than the game needs for to run a map level. Therefore, the abstraction we have done might cause unnecessary memory usage. On the other hand, since we abstracted the common attributes as much as possible, it would be very easy to maintain the game which will result in better user experience. By benefiting these common attributes of subclasses, we hope that collisions, object updates will be handled easily.

Development Time vs User Experience:

During the analysis of the “Battle City” game, we have decided to implement interface components and graphics with JAVAFX libraries because JAVA’s standard SWING library uses 8 bits for resolution on the other hand JAVAFX uses 32 bits for resolution. We have decided to use JAVAFX however it is harder for us to implement and develop the game with JAVAFX libraries because SWING is commonly used and known library which offers us more resources. By choosing implementation with JAVAFX, development time will be a lot harder for us on the other hand with benefits of JAVAFX, we will have better graphics and user experience.

4.2. Final object design

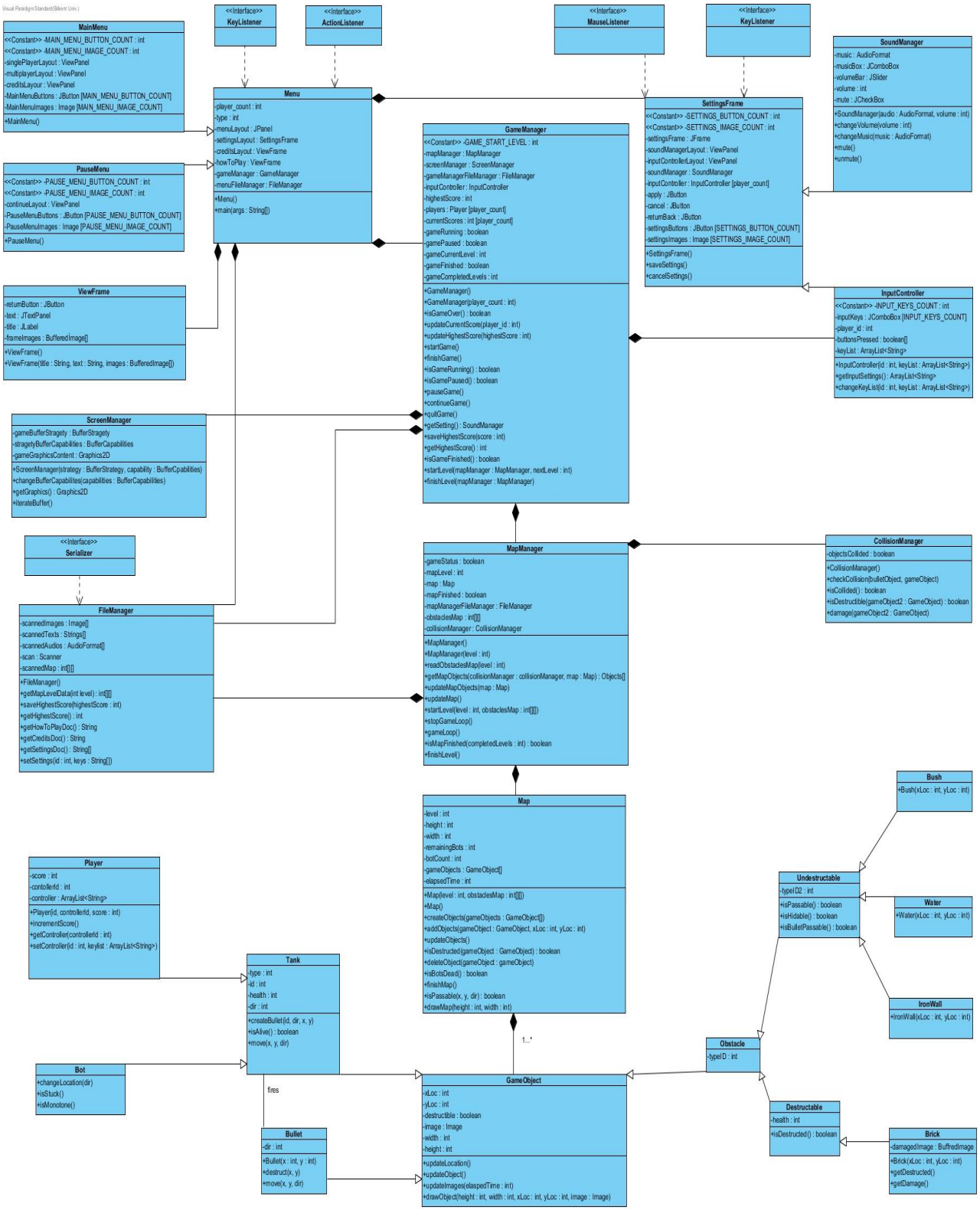

Figure 34-Final Object Design

4.3. Packages

4.3.1. java.util

This package contain ArrayList which will be used as mentioned at classes to control multiple variables, such as control keys of a user, gameObjects currently on map, grid of the obstacles. Also Serializable in this package, which will be used for bytestreaming data from program to hard drive or vice versa. IO, subpackage will be used to interact with folders and .txt files, in order to import default maps.

4.3.2. javafx.scene.layout

This package provides User Interface layouts to fit GUI objects.

4.3.3. javafx.scene.paint

This package provides a set of classes to handle visual output.

4.3.4. javafx.animations

This package provides basic classes to controlling visual output, which is the main reason of our choice of javaFX, rather than java.

4.3.5. javafx.scene.events

This package provides the controll of the events. Delivery of inputs between different hardware components and exception handling is done by this package.

4.3.6. javafx.scene.input

This package provides input handlers for keyboard and mouse.

4.3.7. javafx.scene.image

This package provides the usage of images in program. Even basic obstacles with one color or two color-basic textures are easily drawable, we decided to use images of more complex shapes, such as tanks.

4.4. Class Interfaces

4.4.1. ActionListener

This interface will be invoked whenever an action occurs with receiving the action events. We will implement this interface dependent on the Menu class. It will give action info to Menu class in order to track the action events during the game.

4.4.2. KeyListener

This interface will be invoked whenever a key is typed, pressed or released by the user with receiving keyboard events. The Menu class will implement this interface in order to get keyboard input from the user. Menu class will use this commands from the user and will pass this info to other operative classes.

4.4.3. MouseListener

This interface will be invoked whenever a mouse action is received from the user in order to track the mouse commands. We will implement this interface dependent on the Menu class and the Menu class will operate the corresponding keyboard commands by calling according classes.

4.4.4. Serializable

This interface will be used in order to convert the big data in the memory into a stream of bytes so that it will be much easier to use and transfer this data during the game. We will implement this interface dependent on FileManager class in order to be able to access the saved data and serialize it by the help of this interface.

5. Glossary & References

- [1]JAVAFX Docs

  JavaFX 8, docs.oracle.com/javase/8/javafx/api/toc.htm.