CS 319 - Object-Oriented Software Engineering

System Design Report

Section 02 Group 2c

Space Invaders

Project Group Members

1-Yalchin Aliyev
2-Enes Erol
3-Koray Gürses
4-Arda Gültekin

Supervisor: Uğur Doğrusöz

Contents

1. Introduction

The project we will design and implement an arcade game is called Space Invaders which is a Java language-based video game that has many different versions on the market. Space Invaders was a widely popular arcade game, and still holds its place today as one of the most beloved and most played arcade games. Basically, Space Invaders is a shooting game and the aim is to defeat waves of aliens with laser cannon to earn as many points as possible.

1.1. Purpose of the System

Space Invaders pits your space ship, three lives, and four destroyable shields against an onslaught of aliens from above that only get faster. You move from left to right or vice versa, shooting enemies, while taking cover underneath your shields. The shields can be destroyed by the enemy shots, or by you yourself. While shooting through your own shields gives you more openings to hit the aliens, it also puts your ship at risk. The enemies gradually come down until they reach the bottom. The more you kill, the faster they move. Eventually, when there is only one left, it will move at high speeds as it increases speed, drops down, and reverses direction. As more aliens are defeated, the aliens' movement speed up. Defeating the aliens brings another wave that is more difficult, a loop which can continue without end. Occasionally, a mother ship crosses the top of the screen, awarding the player points for hitting it. Although the point values are thought to be random, players of the arcade game have found out how to make the mother ship award the maximum amount of points.

1.2. Design Goals

- **Visual Design**

The player gets a familiar design from the beginning to the end of the game. It is very simple and easy to start game. Also, game elements icons are very familiar. Even if the player is playing for the first time, does not feel strange.

- **Gameplay**

The player uses right, left arrow keys to move and space key to fire. So playing our game is very simple.

- **Understandability**

The player can easily understand whole game and his/her duty at sight main game screen. Additionally, shapes of game icons help player to understand whole game logic.

- **Cogency**

The game is easy to play but satisfying to keep player inside the game.

- **Performance**

The player does not face with any game play problems while playing game.

- **Difficulty**

The game gets harder during the game. And player has just 3 lives before loses the game. Arranging these difficulties is essential to prevent player getting bored and keep player inside the game.

- **Satisfaction-Joy**

It is important to keep player happy during the game time and after the game finished. The game is successful if player demands to play same game twice.

2. System Architecture

2.1. Subsystem Decomposition

Subsystem decomposition is mainly related with decomposing system into subsystems. This decomposition aims to prevent coupling between different subsystems of the system. By decomposing the software, it is easy to make modification or adding extension when it is needed. Subsystem is like collection of classes, associations of its, operations between itself, events and constraints that are closely interrelated with each other. The objects and classes from the object model are the “seeds” for the subsystems. Via UML diagrams, subsystems are easily modeled as packages.

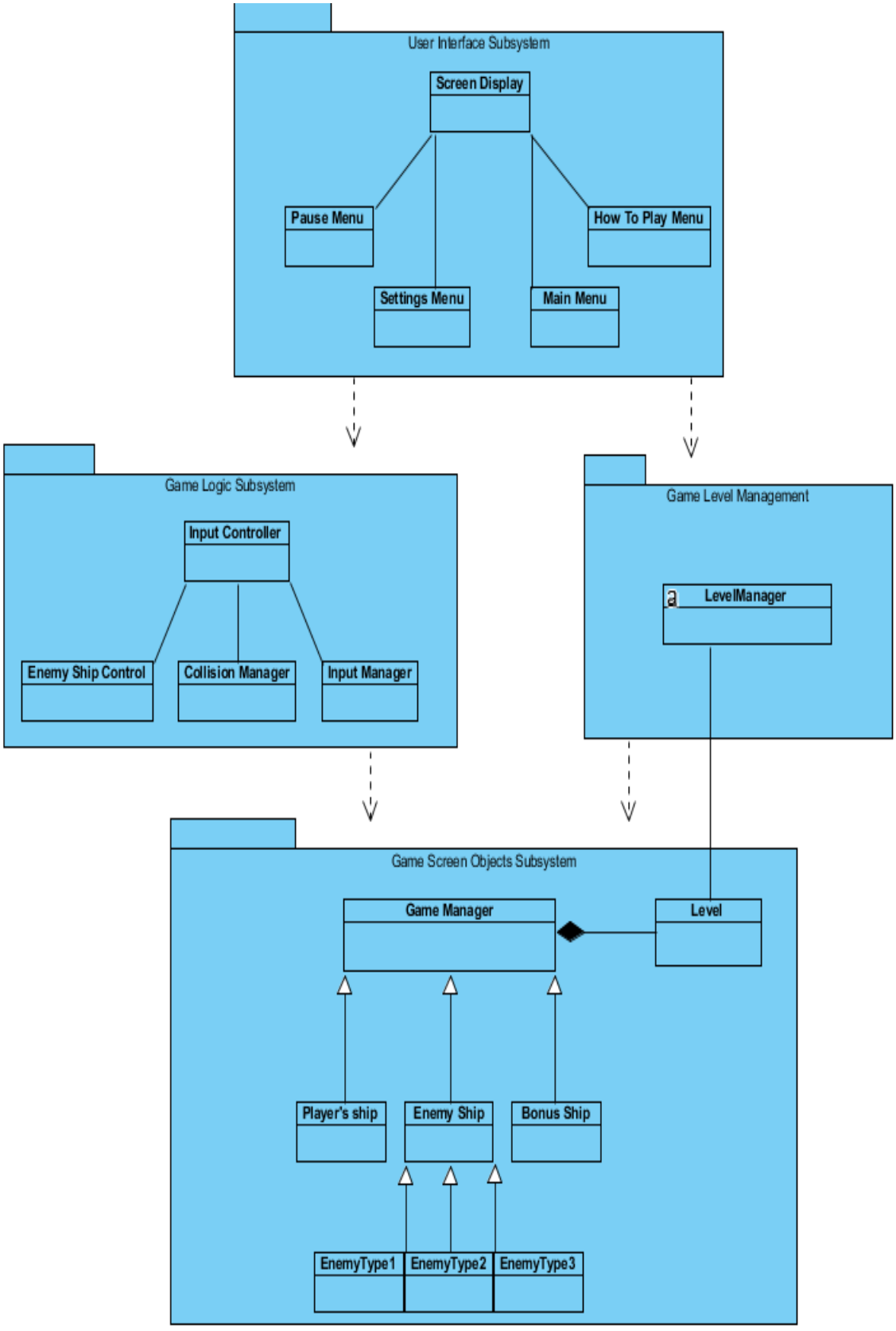
During the decomposition of the system, we use MVC (Model-View-Controller) architectural pattern is great fit to apply on our system. We divided our system based on MVC principles. Meaning that, we decided that our menu classes will be our view since they provide interface for users and our management classes will be our controllers since they control and maintain the game.

The core components or classes required for the core game engine are easily identifiable and it is also relatively simple to map their interactions between themselves and the game screen. If we analyze the original Space Invaders we can easily see that the main classes (or actors) required are the:

- Player
- Enemies
- Bullet
- Enemy Bullet
- Barriers
- Menu

With these fundamental classes it is also easy to classify what high level attributes are associated with them. For example, it is obvious that both the player, enemies and barriers will have some sort of health attribute. We ignore exactly how the health for each of these classes works at this stage and focus on just the attributes. Another less obvious attribute (despite it staring you in the face) is the position of every graphic on the screen. Therefore, all of the above-named classes will require a position attribute so the game engine knows where everything is to draw it.

Finally, each of the actors that is required to move will need to have a speed attribute defined to limit how quickly they move about in 2-Dimensional space. Therefore, with this simple “say what you see” approach to defining classes and attributes I have already set the basis of the class design which can then be progressed to discuss their methods.


User Interface Subsystem
Screen Display
Pause Menu
How To Play Menu
Settings Menu
Main Menu
Game Logic Subsystem
Input Controller
Enemy Ship Control
Collision Manager
Input Manager
Game Level Management
LevelManager
Game Screen Objects Subsystem
Game Manager
Level
Player's ship
Enemy Ship
Bonus Ship
EnemyType1
EnemyType2
EnemyType3

2.2. Hardware/Software Mapping

Space Invaders software will implement on PC and programmed in Java. The object of the program is to handle all necessary game logic, to make the game function as planned. A model of the game, programmed in Java, was used to evaluate the needed functionality. From the model it was derived that functions to initialize the hardware were needed as well as one main game loop. From the loop all the different functions handling the game logic itself were called upon. Functions such as moving and spawning sprites, collision detection and polling keyboard events. The structure of the software was designed to be single threaded. The program is running with a main loop to handle the game logic functions. In addition to this it reads the input from the keyboard after receiving an interrupt. The received information from the keyboard only describes if a key has been pressed or released. Therefore, extra functionality was added to the function handling the interrupt from the keyboard. The player should be able to move and shoot simultaneously

2.3. Persistent Data Management

Space Invaders keeps its data in an ordinary file. This data includes game instances, objects, pictures or music. So, for Space Invaders game we will not use database or any specific data storage. Additionally, other important 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

2.4. Access Control & Security

Space Invaders is a simple driven java game which format is jar. So, we do not need access control authentication methods. The player can easily open game than start to play. Also, there will not any proxy check protocols because the network or internet connection will not be necessary for Space Invaders. Basically, any player can play the game. Therefore, there will not be any access control and security measurement in order to prevent data leaks. The highest scores will be unique for each player, so highest scores will not keep the highest score made in the all of the players, but it will be kept unique for each player

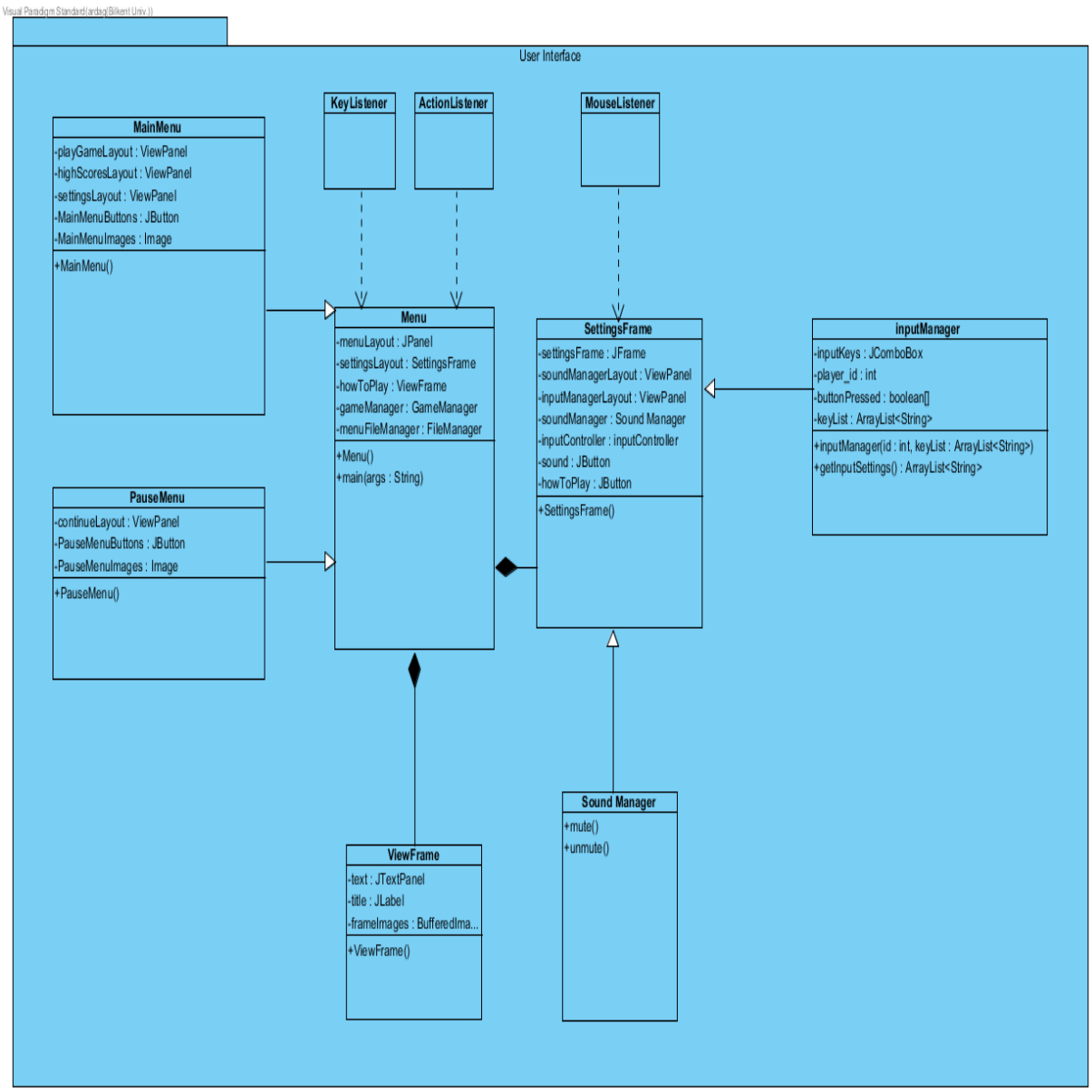
2.5. Boundary Conditions

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

Space Invaders 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. After that, user can terminate the game by clicking “Exit” button in the main menu.

3. Subsystem Services

3.1. User Interface Subsystem

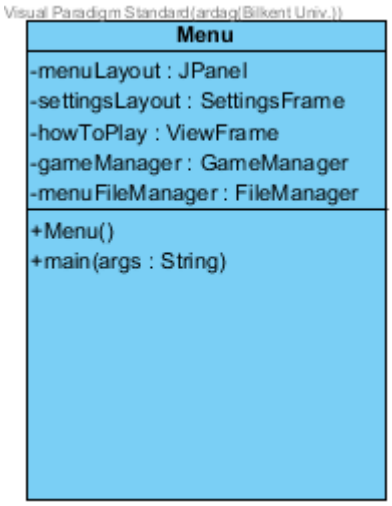


User interface subsystem has seven classes in it. It's main responsibility is the establishment of the interface between the user and the system. Our "Menu" class is the Façade class of the User Interface Subsystem. Only changes in the Game Management subsystem should have happen through "Menu" class. "FileManager" class doesn't effect functionality.

InputManager and SoundManager classes can be reached via the SettingsFrame class, their common methods and attributes are available inside the SettingsFrame class. Similarly, PauseMenu and MainMenu classes can be reached via the Menu class. ViewFrame class helps displaying 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 "High Scores" and "Settings" screens from the Menu.

There are three buttons in the Menu display. These buttons are "Play Game", "High Scores" and "Settings". Each screen will be initialized with the help of the instances of the classes mentioned before. For example, when the user enters the Settings screen, the user will be able to see the settings for which are set for the game.

3.1.1 Menu Class



- **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 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

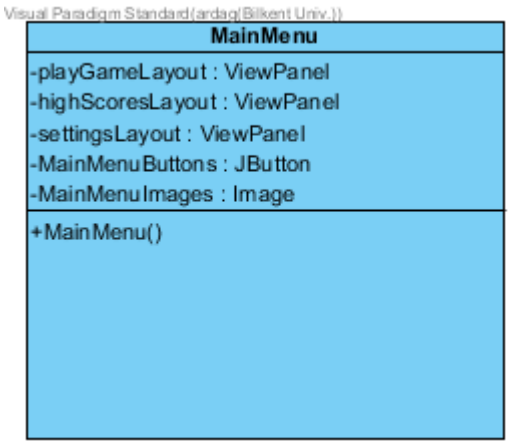
Constructors:

- **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



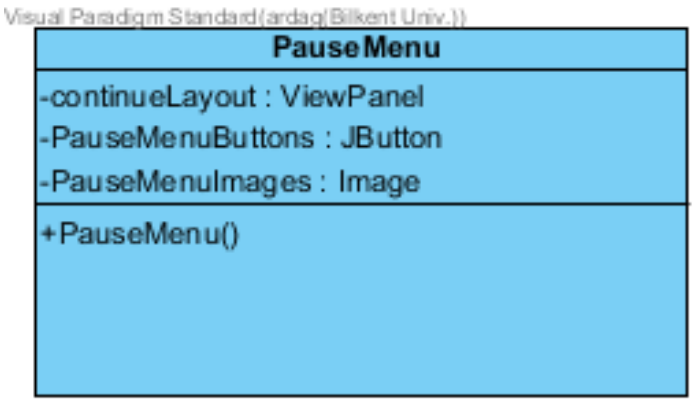
Attributes:

- **private ViewPanel playGameLayout:** This attribute will provide the object layout in the MainMenu, via ViewPanel.
- **private ViewPanel highScoresLayout:** This attribute will provide the HighScore object layout in the MainMenu, via ViewPanel.
- **private ViewPanel settingsLayout:** This attribute will provide the Settings 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



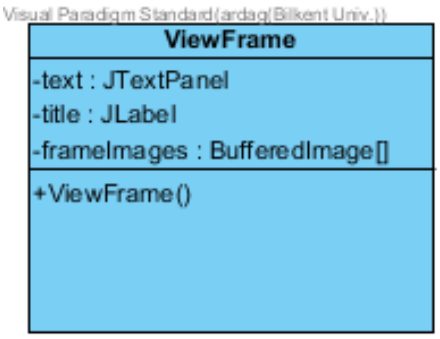
Attributes:

- **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



Attributes:

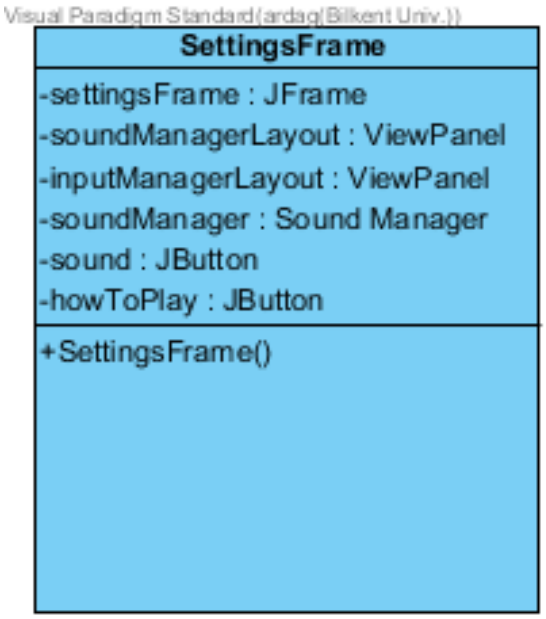
- **private JTextPanel text:** This attribute proves text field for the frame. The necessary texts will be taken by fileManager class and integrated into necessary frames.
- **private JLabel title:** This attribute provides title for the frame.(Ex: “How to Play”)

- **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.

3.1.5 SettingsFrame Class



Attributes:

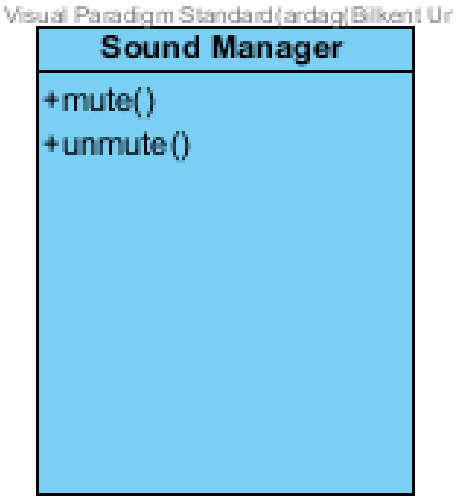
- **private JFrame settingsFrame**: This attribute will be used to create the frame for the Settings screen.
- **private ViewPanel soundManagerLayout**: This attribute will provide the SoundManager object layout in the SettingsFrame, via ViewPanel.
- **private ViewPanel inputManagerLayout**: This attribute will provide the InputManager 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 JButton sound**: This attribute provides the user with the 'sound' button, which aims to save the sound changes made in the Settings screen by the user (mute or unmute).
- **private JButton howToPlay**: This attribute provides the user with how to play screen from the Settings screen.

Constructors:

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

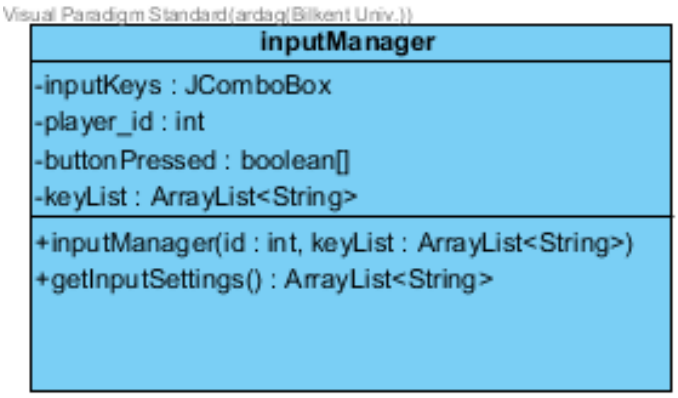
3.1.6 SoundManager Class



Methods:

- **public void mute():** This method mutes the song when the user switches to the mute box.
- **public void unmute():** This method unmutes the song when the user switches to the mute box.

3.1.7 InputManager Class



Attributes:

- **private JComboBox inputKeys:** This attribute is the JComboBox array which will keep the default keys inside, and display them to the user.
- **private int player_id:** This attribute will keep the id of the player inside.
- **private boolean[] buttonsPressed:** This attribute is the boolean array, which keeps the track of the buttons whether they are pressed or not.
- **private ArrayList keyList:** This attribute will provide the user with the list of the keys.

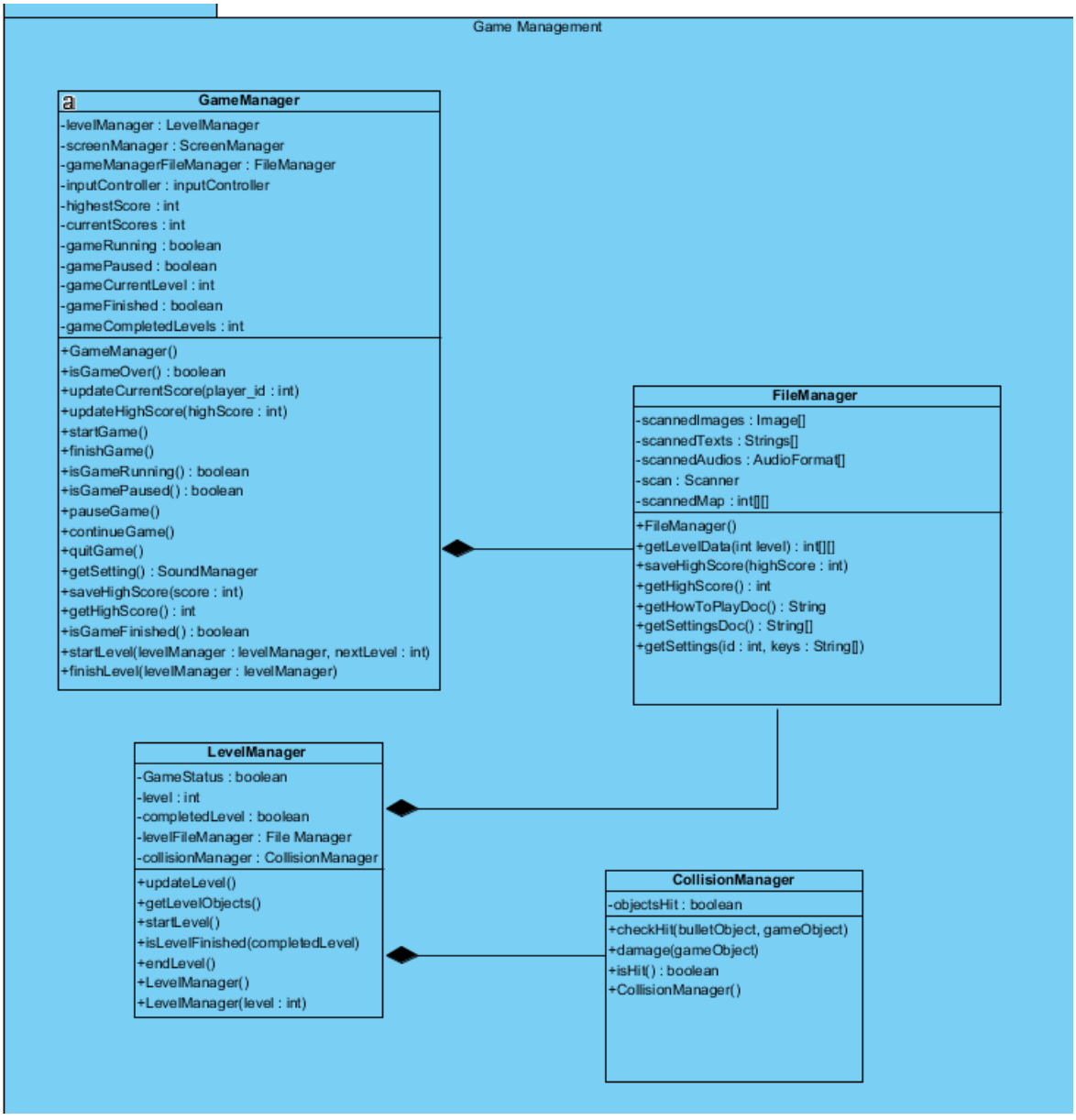
Constructors:

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

Methods:

- **public void getInputSettings():** This method returns the ArrayList and provides the class with the changed and new inputs for the keys.

3.2 Game Management Subsystem

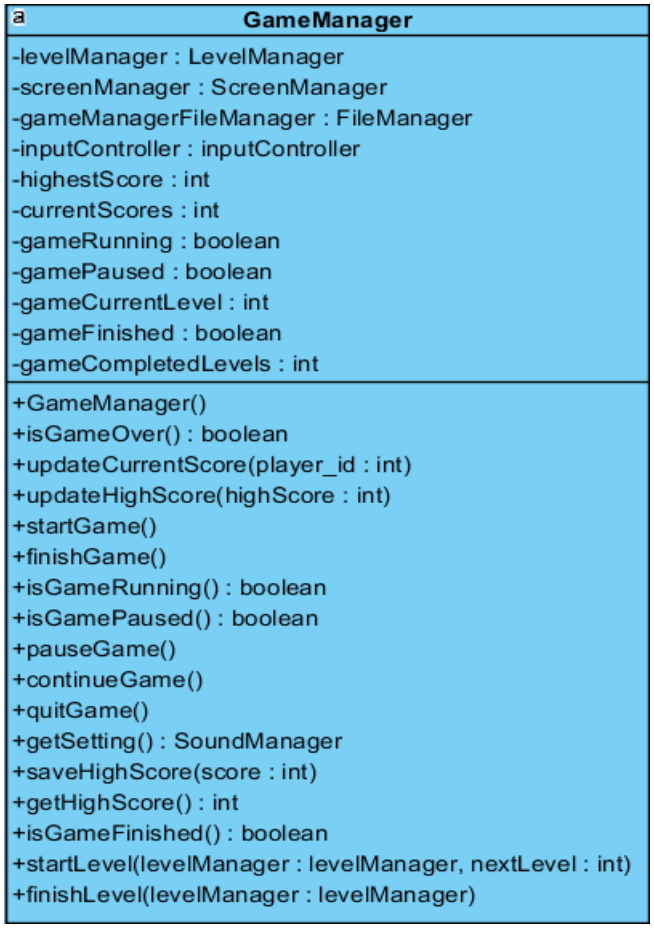


Game Manager Subsystem(GMS) includes five classes and GMS is responsible for the game logic and managing the game mechanism such as collision between bullet and the ship (player's or enemy's) and display.

In Game Manager Subsystem, GameManager is the main class that holds the game logic instances such as File Manager and Level Manager. GameManager determines the level of the game, gets the score of the player and sets sound by reading the files in FileManager.

LevelManager manages the Level class which makes the Level class a façade class. LevelManager also contains the instance of the CollisionManager. CollisionManager determines whether the ships have been shot or not, and also, they can determine whether the shields have been shot by the enemy ship. For example, if an enemy ship has been hit, isHit() method will be passed to LevelManager .

3.2.1. GameManager Class



Attributes:

- **private LevelManager levelManager:** This attribute is the instance of the LevelManager class and provides a connection between the logic of the game and the level design.
- **private ScreenManager screenManager:** Provides the game display on the screen.
- **private FileManager gameManagerFileManager:** This attribute is the instance of the FileManager class and it is connected with GameManager through the files.
- **private InputController inputController:** This attribute is the instance of the InputController class and controls the user inputs during the game.

- **private int highestScore:** This attribute keeps track of the highest score of the player in the game, and saves it in the HighScore table.
- **private int currentScores:** This attribute keeps track of the user's score during the game.
- **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 tracks the level that player is playing.
- **private boolean gameFinished:** This attribute keeps track of whether the game is finished or not.
- **private int gameCompletedLevels:** This attribute keeps the score of the player and keeps the score in HighScore table and gets the boolean value from the gameFinished.

Constructor:

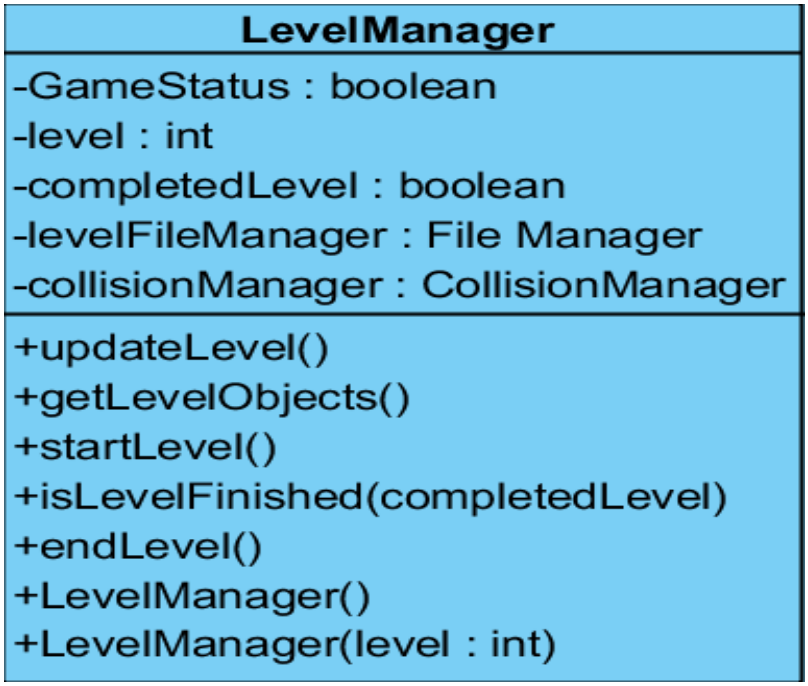
- **GameManager():** Default constructor for the GameManager class, which simply initializes the class.

Methods:

- **private boolean isGameOver():** This method checks whether the game is over or not.
- **private void updateCurrentScore():** This method updates the score of the player.
- **private void updateHighScore(highestScore : int):** This method updates the high score of the player.
- **private void startGame():** This method starts the game.

- **private void finishGame():** This method finishes the game.
- **private boolean isGameRunning():** This method checks whether the game is running or not.
- **private boolean isGamePaused():** This method checks whether the game is paused or not.
- **private void pauseGame():** This method pauses the game.
- **private void continueGame():** This method continues the game if the game is in a pause state.
- **private void quitGame():** This method quits the game.
- **private getSetting():** This method gets the sound settings which would be on or off from the Sound Class.
- **private void saveHighScore(score: int):** This method saves the high score of the player.
- **private int getHighestScore():** This method returns the highest score of the player.
- **private boolean isGameFinished():** This method checks whether the game is finished or not.
- **private void startLevel(levelManager: LevelManager, nextLevel: int):** This method starts the next level, if the user finish the level.
- **private void finishLevel(levelManager: LevelManager)**: This method finishes the level.

3.2.2. LevelManager Class



Attributes:

- **private boolean gameStatus:** This attribute saves the current state of the game. (In other words, checks whether the game is running or paused etc.)
- **private int level:** This attribute keeps the count of the level that the player is currently playing.
- **private boolean CompletedLevel:** This attribute checks whether the game is completed or not.
- **private FileManager levelFileManager:** This attribute is the instance of the FileManager class and provides the levelManager with the files from the FileManager.
- **private CollisionManager collisionManager:** This attribute is the instance of the CollisionManager.

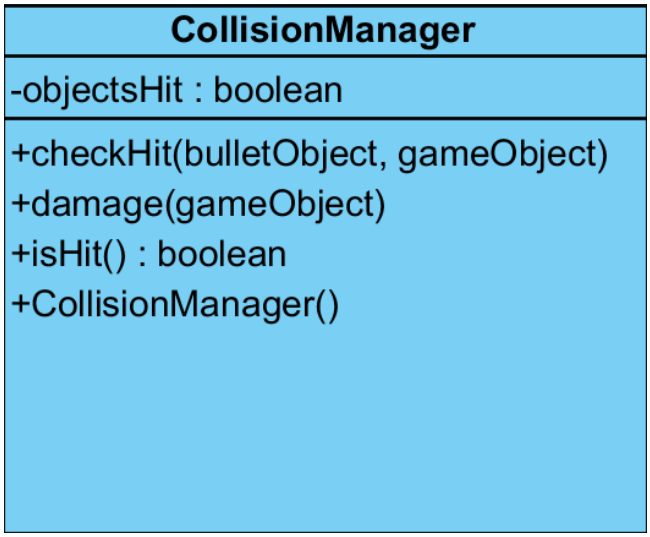
Constructors:

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

Methods:

- **private void updateLevel():** This method updates the map for each level the player has passed.
- **private Object[] getLevelObjects():** This method returns the instances of objects depending on the map.
- **private void startLevel(level : int):** This method starts the new level with the specific level count.
- **private boolean isLevelFinished(completedLevel : int):** This method checks whether the map is finished initializing.
- **private void endLevel():** This method ends the level.

3.2.3. Collision Manager Class



Attributes:

- **private boolean objectsHit():** This attribute will be used to determine that whether there is a hit.

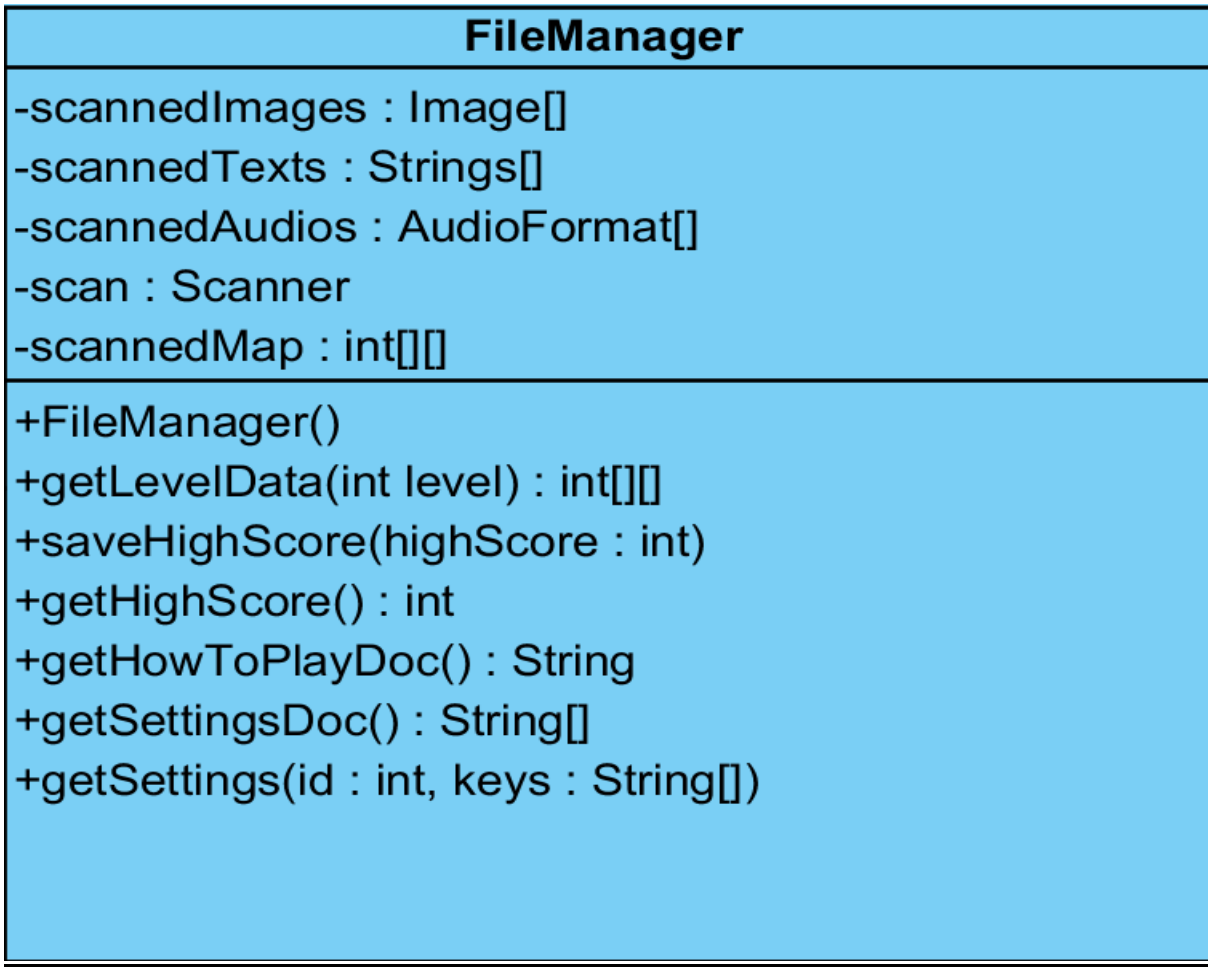
Constructor:

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

Methods:

- **public void checkHit(bulletObject, gameObject):** This method will implement the process of the hit by bullet.
- **public boolean isHit():** This method will return true if there is a hit between ships.
- **public void damage(gameObject):** This method will shows the damage from the ships on the game objects such as shields.

3.2.4 FileManager Class



Attributes:

- **private Images[] scannedImages:** This attribute keeps the scanned images in an Image array to send the images to the draw methods.
- **private String[] scannedTexts:** This attribute keeps the scanned texts in a String 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.

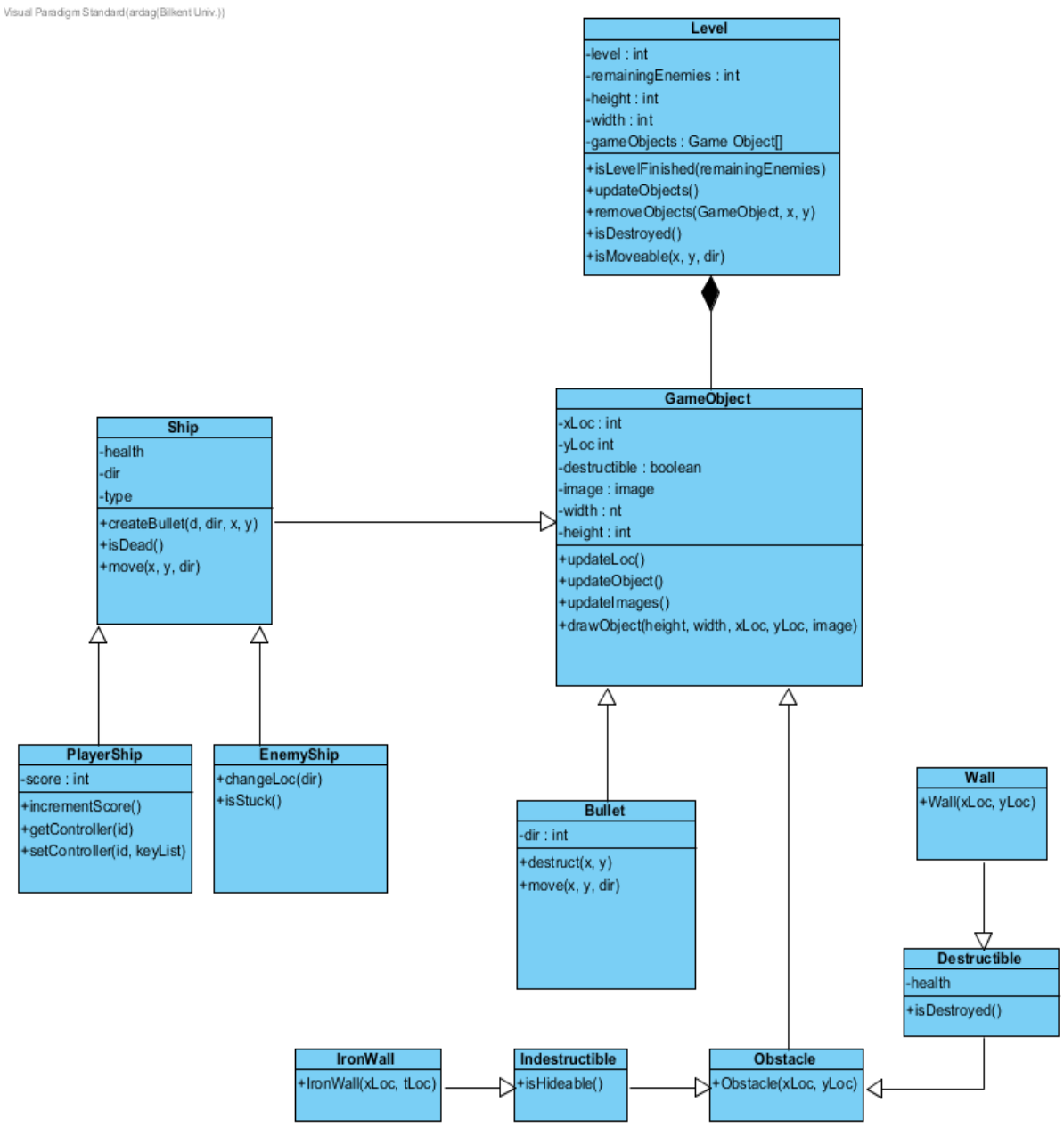
Constructor:

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

Methods:

- **public int[][] getLevelData(int level):** This method gets the level data from levelData text by taking the speed of the enemy ships and their fire rate as parameter and assings it to scannedMap attribute.
- **public void saveHighScore(int highScore):** This method gets the high score from the GameManager and saves it to high score text file.
- **public int getHighScore():** This method gets the high score from high 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[] 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(String[]):** This method gets the changed input settings from SettingsFrame and saves the new input settings to settings text file.

3.3 Game Objects Subsystem



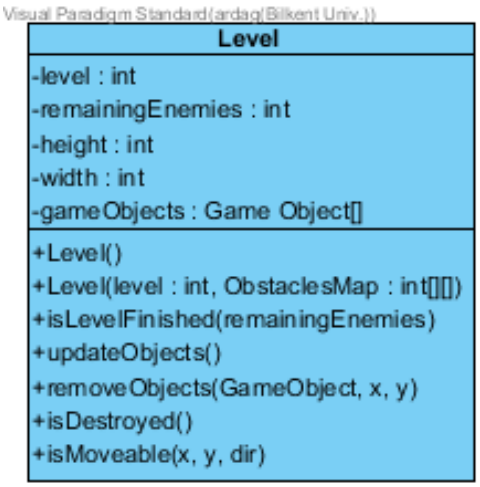
Game Objects Subsystem has eleven classes in it. This subsystem is responsible for the display of the game's main objects on the screen. The main objects of the game are ships (can be a player ship or enemy ship), obstacles: wall or iron

wall and bullets. According to the objects the user is going to see on the screen, the subsystem holds the classes of them inside respectively:

Ship: Player and Enemy, Bullet, Destructible: Wall, Indestructible: Iron wall.

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

3.3.1 Level Class



Attributes:

- **private int level:** This attribute will hold the map level.
- **private int remainingBots:** This attribute will hold the remaining bots in the current level.
- **private int height:** This attribute will be used to determine level/map size.
- **private int width:** This attribute will be used to determine level/map size.

- **private GameObject[] gameObjects:** This attribute is the GameObject array, which holds the every object on the current level.

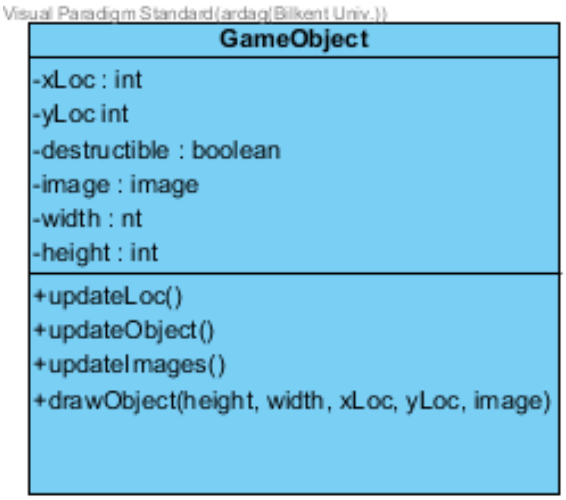
Constructors:

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

Methods:

- **public void updateObjects():** This method will update all objects on the level.
- **public boolean isDestroyed:** This method decides whether a game object is destroyed by a bullet or not.
- **public void removeObjects(GameObject, x, y):** This method removes an object from the level according to the info coming from isDestroyed(gameObject: GameObject) method(whether it is destroyed or not).
- **public void isLevelfinished(remainingEnemies):** This method finishes the level if all enemies of current level is dead and invokes new Level.
- **public boolean isMoveable(x, y, dir):** This method decides whether the movement of a ship or bullet is possible or not.

3.3.2 GameObject Class



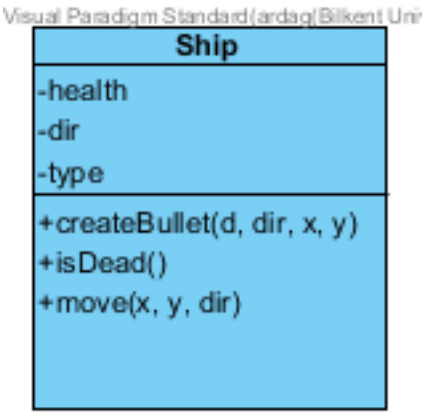
Attributes:

- **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():** Updates the object's location with given frequency.

3.3.3 Ship Class



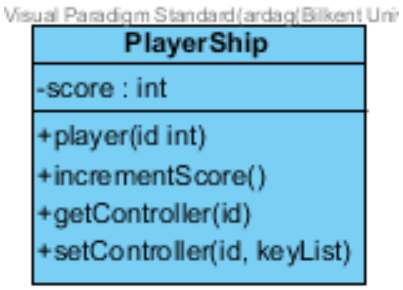
Attributes:

- **private int type:** This attribute holds the type of the ship which could be player or enemy.
- **private int health:** This attribute holds the current health of the ship.
- **private int dir:** This attribute determines the direction of the bullet and image(front of ship).

Methods:

- **public void createBullet(id: int, dir: int, x: int, y: int):** This method creates a bullet object.
- **public boolean isDead():** This method determines whether the ship is dead or not.
- **public void move(x: int, y:int, dir: int):** This method requests the movement on map.

3.3.4 PlayerShip Class



Attributes:

- **private int score:** This attribute holds the score of the player, which is incremented by killing bots and finishing levels.

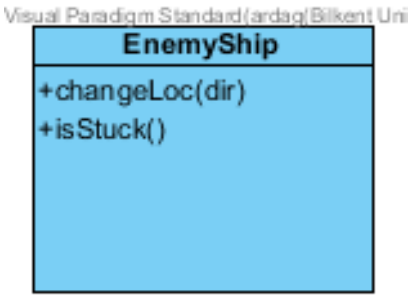
Constructors:

- **player(id : int):** The constructor of the player class determines the id of the player.

Methods:

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

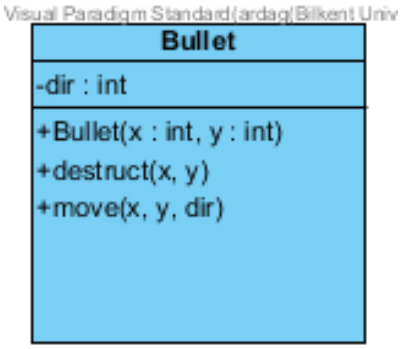
3.3.5 EnemyShip Class



Methods:

- **public void changeLocation(dir: int):** This method changes the x or y value of enemy ship depending on the direction.
- **public void boolean isStuck():** This method determines whether enemy ship is trying to move on an unpassable object. If it is, method changes the direction.

3.3.6 Bullet Class



Attributes:

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

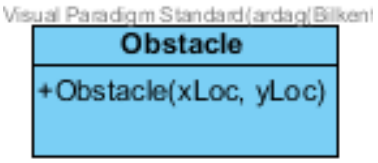
Constructors:

- **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 destroyable object at that point.
- **public void move(x: int, y: int, dir: int):** This method provides the movement logic for the bullet.

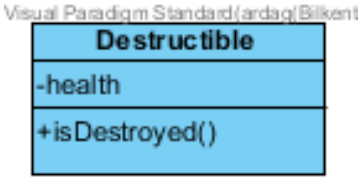
3.3.7 Obstacle Class



Constructors:

- **Obstacle(xLoc : int, yLoc : int):** Constructor of the obstacle. Creates the obstacle on the given position.

3.3.8 Destructible Class



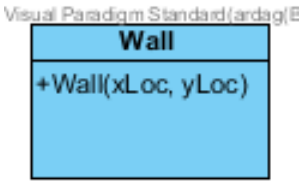
Attributes:

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

Methods:

- **public boolean isDestroyed():** This attribute determines if the object is destroyed or not.

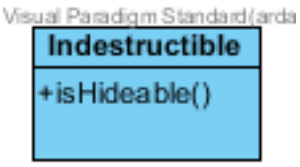
3.3.9 Wall Class



Constructors:

- **Wall(xLoc, yLoc):** Constructor of the wall. Creates the wall on the given position.

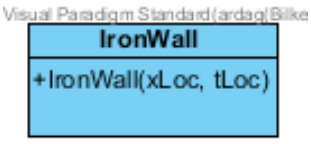
3.3.10 Indestructible Class



Methods:

- **isHideable():** Determines whether the indestructible object is hideable or not. (player can hide behind some indestructible objects).

3.3.11 IronWall Class



Constructors:

- **IronWall(xLoc, yLoc):** Constructor of the iron wall. Creates the iron wall on the given position.

4. Low-level Design

4.1. Object Design Trade-Offs

Understandability vs Functionality:

Main goal of our project is to make a simple game which is addictive at the same time. Because of this, we think that to increase understandability we should decrease functionality. By decreasing functionality what we mean is decreasing number of complex objects and decreasing complexities of the current components of the project. We believe that with good understandability and simple functionality the game will be addictive and enjoyable for its users.

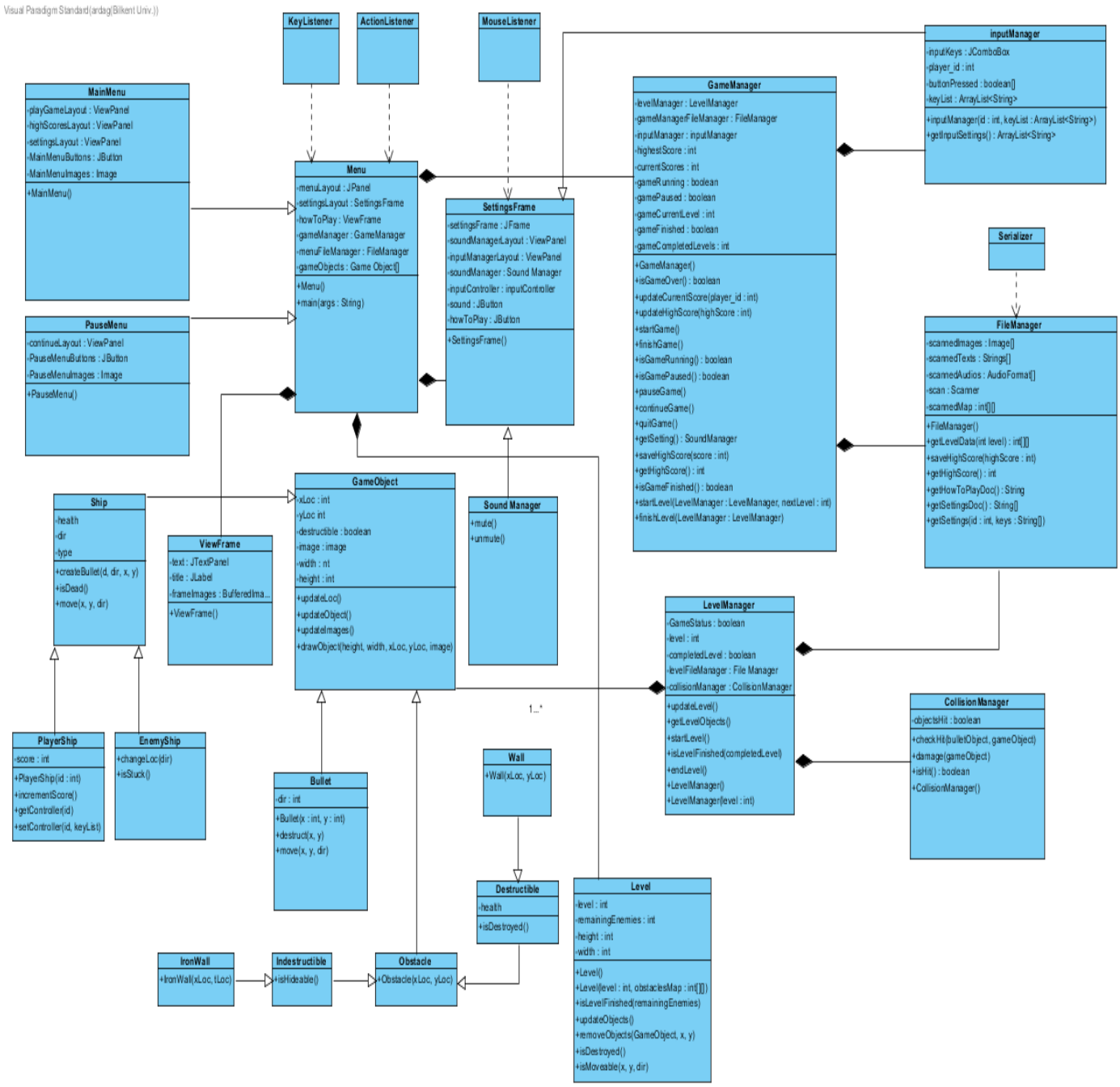
Memory vs Maintainability:

We aimed to make our project as much maintainable as possible. That is why we used a lot of abstraction for class-object design. Having many abstractions might cause to have additional classes that are not that required for the implementation of the main project. Therefore, during the implementation it might cause usage of additional memory.

Development Time vs User Experience:

In this project we don't have much time for implementation that is why we will consider using tools and programming language that we are already familiar with to keep development time as less as possible. Most people use C++ to implement games to have smoother user experience, but we are mostly familiar with Java which is not the best for user experience. Furthermore, the Java library that we will use for development is Swing which is not good for decent game development. We might consider using JavaFX for user interface if we have enough tie for implementation period.

4.2. Detailed Object Design



4.3. Packages

4.3.1 java.util

This package contains some classes such as ArrayList, that will be used to store dynamically allocated data and do simple operations on them efficiently.

4.3.2 java.io

This package will provide functionalities to read and write to a file.

4.3.3 java.swing

This package will provide frames and panels to put game components and show on the screen.

4.3.4 java.awt

This package will provide functionality to draw game components and control the interactions between them.

4.3.5 java.awt.event

This package will prove classes to control user input and other event driven activities.

5. Glossary & References

- Java Docs

  Java™ Platform, Standard Edition 7, API Specification,

  https://docs.oracle.com/javase/7/docs/api/overview-summary.html