CS 319 - Object-Oriented Software Engineering

Analysis Report

RUNNING JON

Group

ERKAN ÖNAL

NİHAT ATAY

BARIŞ ARDIÇ

MERT KARA

Table of Contents

Table Of Figures

1.Introduction

1.1 Purpose of the System

The Running Jon is a 2D action game. If we compare it to other 3D games or 2D games that are generated using an existing game engine, it stays down a bit. However; there are many games whose graphics are not that cool, even very bad but people really like them. Therefore; we want to achieve this. To achieve this, what is going to help us attract people to this game is that as most of the people do like Game of Thrones tv series. We will integrate its key characters into game and we will use their voices in our game to make it sound like real Game of Thrones atmosphere. We will even use their well-known expressions in the game as sound. In addition, to attract more gamers, we will have the enemies acting with a complex movement pattern. Not that trivial ones that many space intruders games have. Therefore; it will be more challenging for users and they will hopefully like that. In addition, we think that the clever movement patterns of the enemy will test and challenge the reflexes of the user and his attention span.

1.2 Design Choices

The inspiration for the game flow and mechanics is the classical arcade game; Space Invaders with contribution from the recent independent retro gaming trend. As a result, the emphasis of the development is not on graphics in spite of

the contemporary AAA titles from the industry. A big achievement for the project group is being able to convey the classic arcade feel.

The programming language chosen for the development is Java since it is object oriented and all members of the project group are familiar with it to some extent. Since emphasis is always on the design throughout the project process the end result is expected to have properties such as reusability and maintainability with a well-formed object structure.

The course of the game is divided into levels in order to increase playability with providing the player with a sense of accomplishment even though the game is not beaten yet. The controls are kept basic so that the player's attention is always on the game flow. This enables the project to be a fast-paced game thus in parallel with the arcade spirit.

1.3 Design Goals

Adaptability:

This feature is easy to achieve because of the properties of the Java language. Since Java executables will work on any platform with Java Runtime Environment the effort required for a non-Windows is minimal. Java is known for sacrificing performance in order to achieve adaptability. Such performance tradeoff is not recognizable because Running Jon is a small scale project.

Extensibility:

Since the project has a well-defined object structure it is easier to add more content to the game such as more levels, new enemies and new projectiles or it is easier to make playability improvements such as different game speeds or different background images as well as new in game music.

Usability:

Running Jon is an arcade inspired game therefore having similar controls to an arcade game is important. The purpose is to have a challenging game but not because it is not easy to learn to play but because it is hard to master the gameplay. Since the graphics of the game do not consume large amounts of memory, memory efficiency of the game data is considered less important than the overall game performance. We expect that this aspect will increase game performance.

2. Software Architecture

2.1 Subsystem Decomposition

In order to have low coupling and high coherence, we decided to use three-tier architectural style in our project. We separated the system to three different subsystems as User Interface, Game Management, and Game Entities.

User Interface subsystem is where the interaction with user occurs. It has no responsibility in the game logic. User Interface subsystem is

composed of the classes which are responsible for graphical user interface and sound. Therefore, we can say its main purpose is to draw the necessary objects. It is responsible for presentation.

Game Management subsystem is responsible of the management of the game using Game Entities. This subsystem manages how the objects will interact with each other, how the game will progress, etc. In other words, it is the subsystem of the logic of the game. All the classes in Game Management are similar in terms of task, so it has high coherence.

The relation between Game Management and User Interface subsystem is through LevelManager class, which lowers coupling and allows flexibility for any change.

Last subsystem, which is Game Entities, is mainly formed of simple object classes such as Soldier, JonSnow, Bullet etc. It is responsible for providing the objects and data needed for the game. It has no responsibility in providing any interaction. It has high coherence since all classes are similar in terms of purpose and tasks.

Unlike the relation between User Interface and Game Management, the dependency of Game Management and Game Entities has a higher coupling when compared since Game Management have to use all of the Game Entities. However, any change in Game Management doesn’t affect Game Entities.

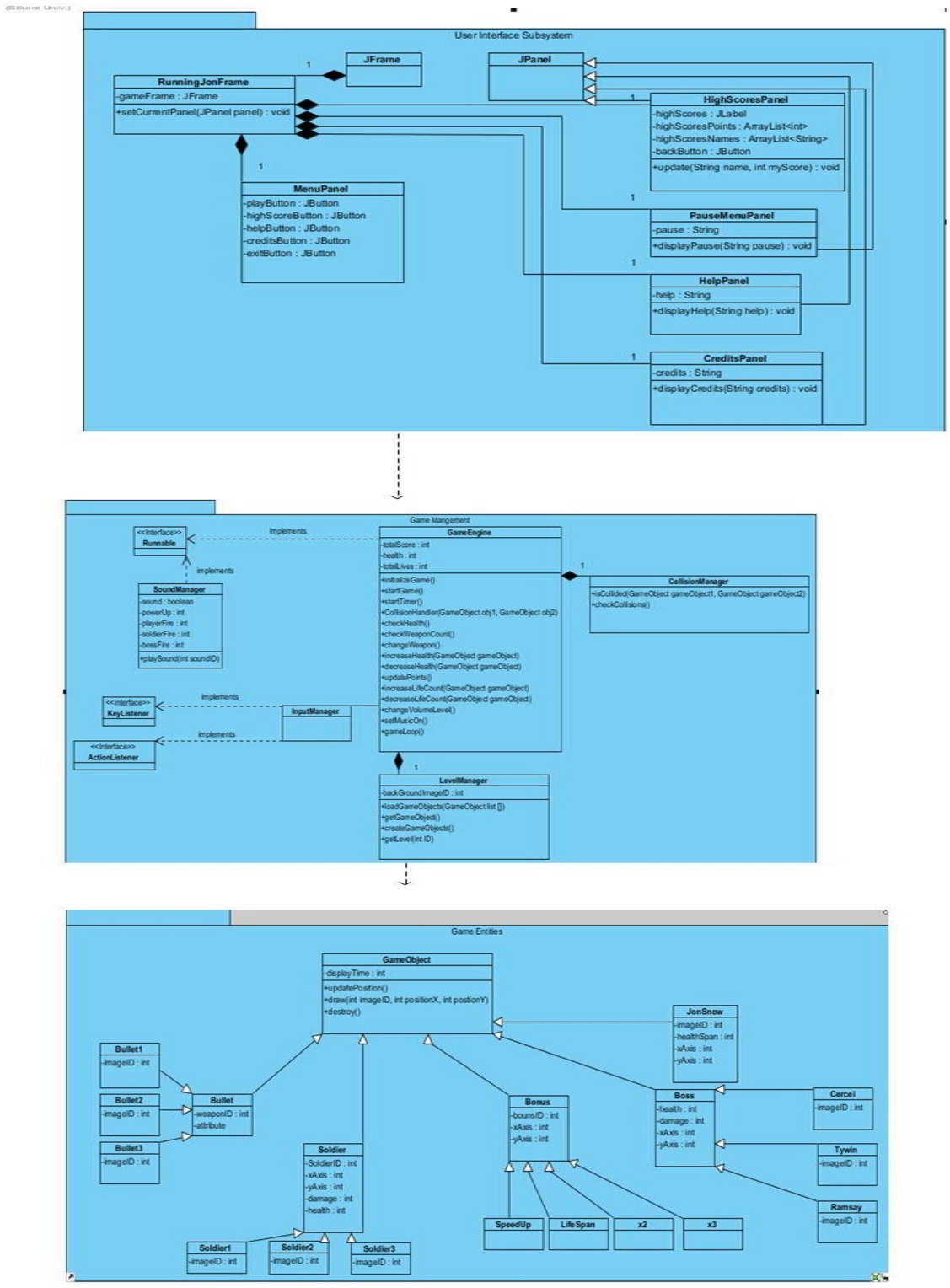

Figure 2: Detailed Subsystem Decomposition

2.2 Architectural Styles

We decomposed the system into three layers, as User Interface,Game Management and Game Entities. These three layers are decomposed hierarchically. Top layer, which is User Interface, has the highest hierarchy since it is not used by any other layer above. It is responsible for the interaction with the user. The following layer is Game Management. This layer is responsible for the actual game logic. Our bottom layer is Game Entities layer, in which all the necessary entity objects are brought together. Our layer decomposition also proposes the closed architectural style, in which a layer can only access to the layer below it. The figure below shows the layers.

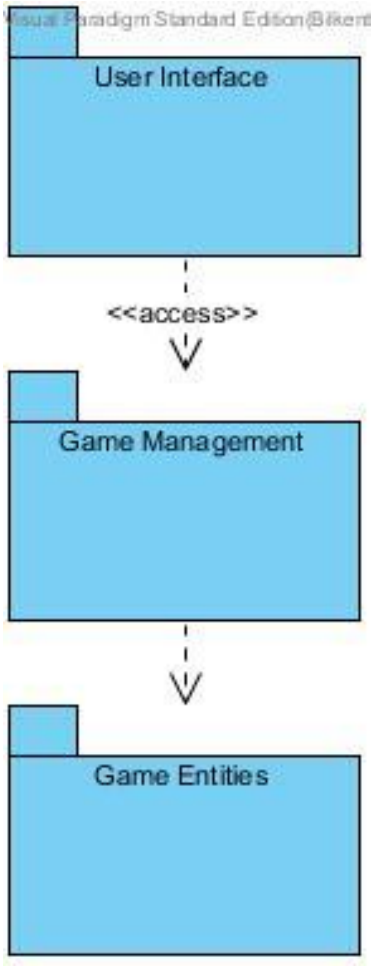

Figure 3: Layers

2.3 Hardware/Software Mapping

Running Jon will be implemented in Java. The latest JDK (1.8) will be used. As hardware configuration, our game needs a basic keyboard to control the character Jon. A basic computer with Java Virtual Machine will be enough for Running Jon. For storage issues, we do not need to use a complex database system. The part we need to store is just highest 10 scores so we will just have .txt file to load a new high score to it and when displaying high scores, we will read them from this .txt file.

2.4 Data Management

Running Jon does not need a complex database system since too little data is needed to be managed. High score list will be used from text files which will be saved in disk. We also plan to store sounds and game object images in hard disk drive with proper and simple sound, image formats such as .gif, .waw.

2.5 Access Control and Security

Running Jon does not require any kind of network connection. Anyone who has the .jar file will be able to play the game. Therefore, there will not be any kind of restrictions for access. Also since the game does not include any user profile, there will be no issues for security.

2.5 Boundary Conditions

Initialization

To open the game and play it, there will be .jar file. After saving it to its local drive, user can double click to open and play it so there will not be need for installation process.

Termination

The game can be terminated clicking “Quit Game” button in main menu. The close button at right-top of program will also terminate the game.

Errors

If an error occurs such that game resources would not be loaded such as sound or images, the game won’t start. If the game doesn’t not respond because of a performance issue, player will have to loose data.

3. Subsystem Services

3.1 Design Patterns

Singleton Design Pattern:

Singleton pattern is a design pattern that restricts the instantiation of a class to one object and provides a global point of access to that instance. It is a creational pattern. This is very useful when exactly one object needed to coordinate the actions between classes. All the time we will know that we have exactly one object of a specific class. The system can operate more efficiently if there is just one object.

In our design, we use the singleton design pattern for GameEngine class because it is our manager class for the game to maintain the actions

in the game. It will detect any most of the changes and update the different values of the game, which are on the Game Management subsystem. Therefore; will need it most of the time and by using singleton design pattern, we create one instance of the GameEngine class in GameEngine class and we use it in different classes.

Façade Design Pattern:

Façade Design pattern is a design pattern that hides the complexities of the system and provides an interface to the client using which the client can access the system. It is a structural pattern. This pattern adds an interface to existing subsystem to hide its complexities.

In our design, we use the façade design pattern in LevelManager class for Game Entities subsystem. Therefore; LevelManager handles the operations on the objects of the subsystem entity objects depending on the needs of subsystem Game Management. In Façade design, façade objects have to be singleton so our LevelManager class will also be singleton.

3.2 User Interface Subsystem Interface

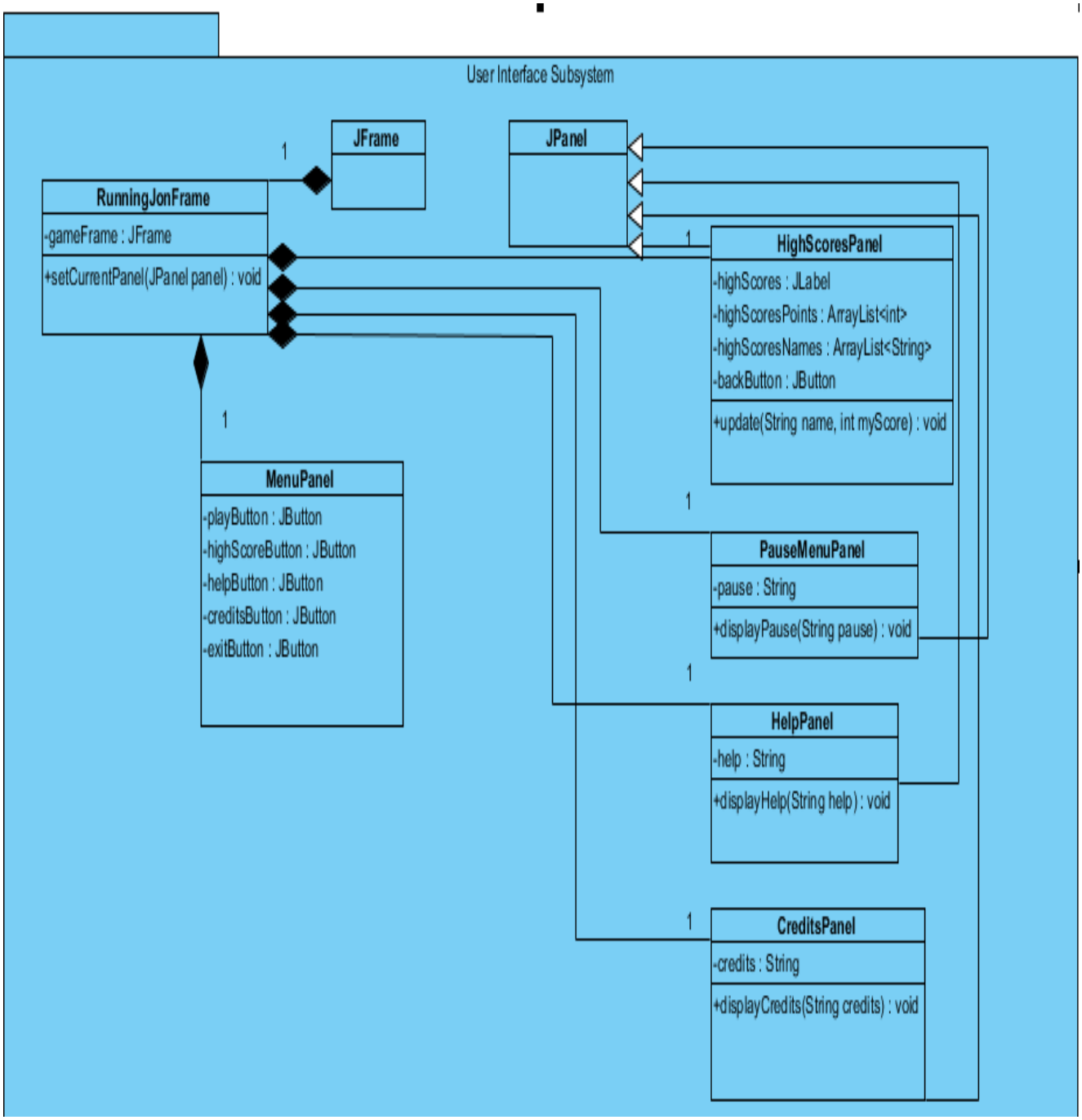

Figure 4: User Interface Subsystem

3.2.1 MenuPanel Class

Figure 4: MenuPanel Class

Attributes:

private JButton playButton: Button for play option.

private JButton highScoreButton: Button to open high scores.

private JButton helpButton: Button to open help.

private JButton creditsButton: Button to open credits.

private JButton playButton: Button to exit game.

3.2.2 RunningJonFrame Class

Figure 5: RunningJonFrame Class

Attributes:

private JFrame gameFrame: This attribute holds the game frame.

Methods:

public void setCurrentPanel(JPanel panel): This method is to change panel in frame.

3.2.3 HighScoresPanel Class

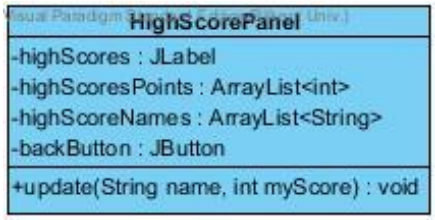

Figure 6: HighScorePanel Class

Attributes:

private JLabel highScores: This attribute label for high scores.

private ArrayList<Integer> highScorePoints: This attribute holds the high scores.

private ArrayList<String > highScoreNames: This attribute holds the names of players who scored highest points.

private JButton backButton: Button to go back to menu.

Methods:

public void update(String name, int myScore): This method updates high scores with new score and players name.

3.2.4 PausePanel Class

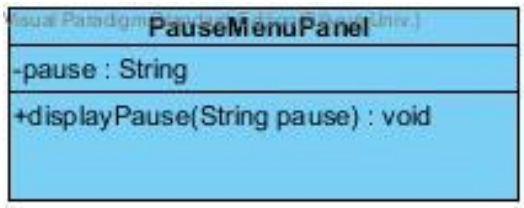

Figure 7: PauseMenuPanel Class

Attributes:

private String pause: This attribute holds the text in the pause.

Methods:

public void displayPause(String pause): This method displays the text for the pause menu.

3.2.5 HelpPanel Class

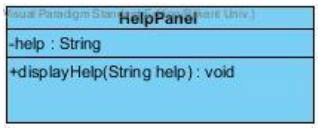

Figure 8: HelpPanel Class

Attributes:

private String help: This attribute holds the help text.

Methods:

public void displayHelp(String help): This method is to display help text.

3.2.6 Credits Class

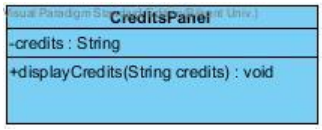

Figure 9: CreditPanel Class

Attributes:

private String credits: This attribute holds the credits text.

Methods:

public void displayCredits(String credits): This method is to display credits text.

3.3 Game Management Subsystem Interface

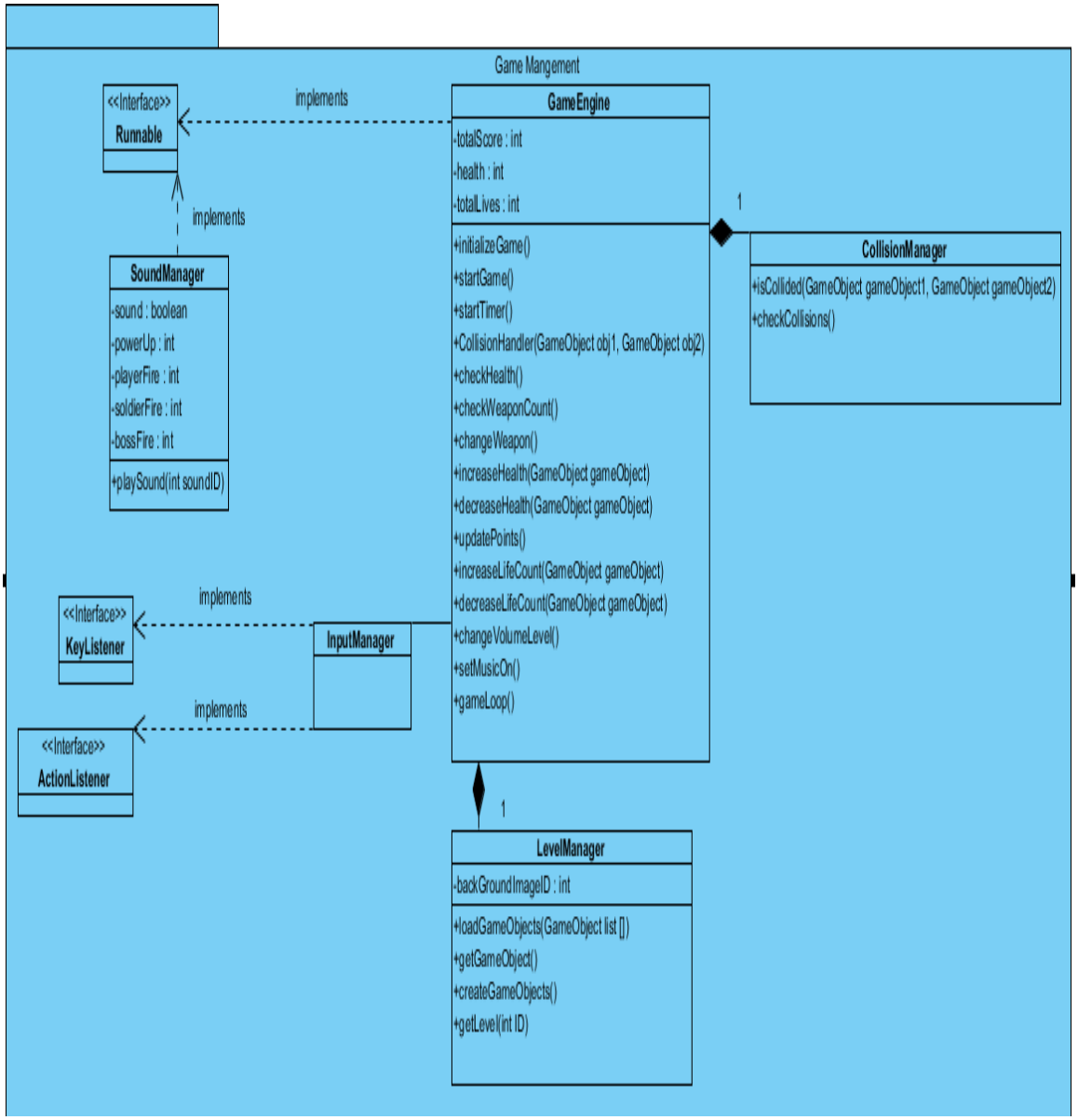

Figure 10: Game Management Subsystem

3.3.1 Collision Manager Class

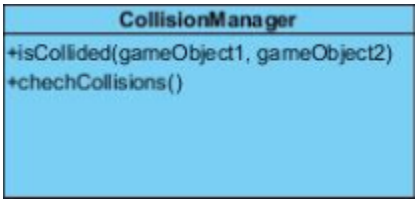

Figure 11: Collision Manager Class

Methods:

public boolean isCollided(GameObject gameObject1, GameObject gameObject2): This method checks collision between two game objects. If there is collision is returns true, otherwise returns false.

public void checkCollisions(): This method checks collisions during game play.

3.3.2 Game Engine Class

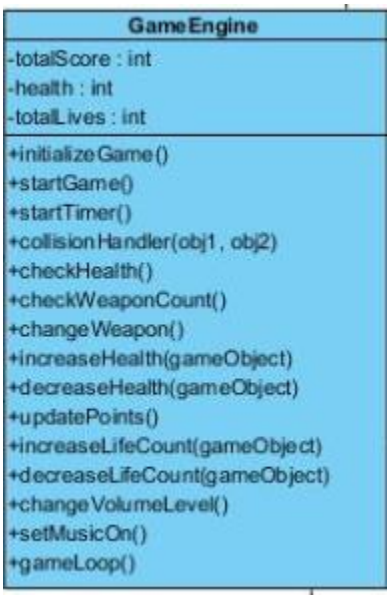

Figure 12: Game Engine Class

Attributes:

private int totalScore: totalScore attribute keeps record of score during game play.

private int health: health attribute keeps health of Jon Snow (player) during game play.

private int totalLives: totalLives attribute represents total life count of player. If it is 0, game overs.

***Methods*:**

public void initializeGame(): This method initializes game with default attributes.

public void startGame(): When user hit the Start Game button, this method starts game with default attributes.

public void startTimer(): This method keeps time record for bonus objects. After a specific time, bonus object vanishes.

public boolean collisionHandler(GameObject obj1, GameObject obj2): This method handles collisions between two game objects.

public int checkHealth(): This method checks players health during game play.

public int checkWeaponCount(): This method checks how many weapon left during game play.

public void changeWeapon(): This method allows to player to change weapon type during the game. If user gets bonus object to change weapon, this method handles weapon changing.

public void increaseHealth(GameObject gameObject: This method increases player's

health when player gets extra health bonus.

public void decreaseHealth(GameObject gameObject): This method decreases player's health when player is shooted by enemies' bullets.

public void updatePoints(): This method updates player's point during the game. Player gains point when s/he kills soldiers or bosses.

public void increaseLifeCount(): This method increases life count of player when player gets extra life bonus.

public void decreaseLifeCount(): This method decreases life count of player when player's health is 0.

public void changeVolumeLevel(): This method changes volume level of game.

public void setMusicOn(): This method allows to user set music on or off.

public void gameLoop(): This method handles general game play.

3.3.3 Sound Manager Class

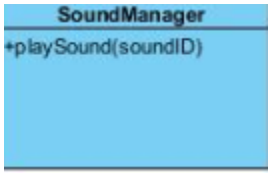

Figure 13: Sound Manager Class

***Methods*:**

public void playSound(int soundID): This method plays the sound whose ID is given in the parameter.

3.3.4 Input Manager Class

Figure 14: Input Manager Class

3.3. 5 Level Manager Class

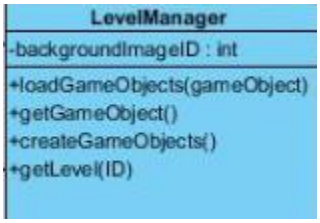

Figure 15: Level Manager Class

***Attributes*:**

private int backgroundImageID: This keeps the ID of background image depending on level.

Methods:

public void loadGameObjects(GameObject gameObject): This method loads the necessary game object given in parameter for level.

public GameObject getGameObject(): This method returns the game object.

public void createGameObject(): This method handles the creation of necessary objects in a new level, depending on the level.

public void getLevel(int ID): This method returns the level number.

3.4 Game Enitities Subsystem Interface

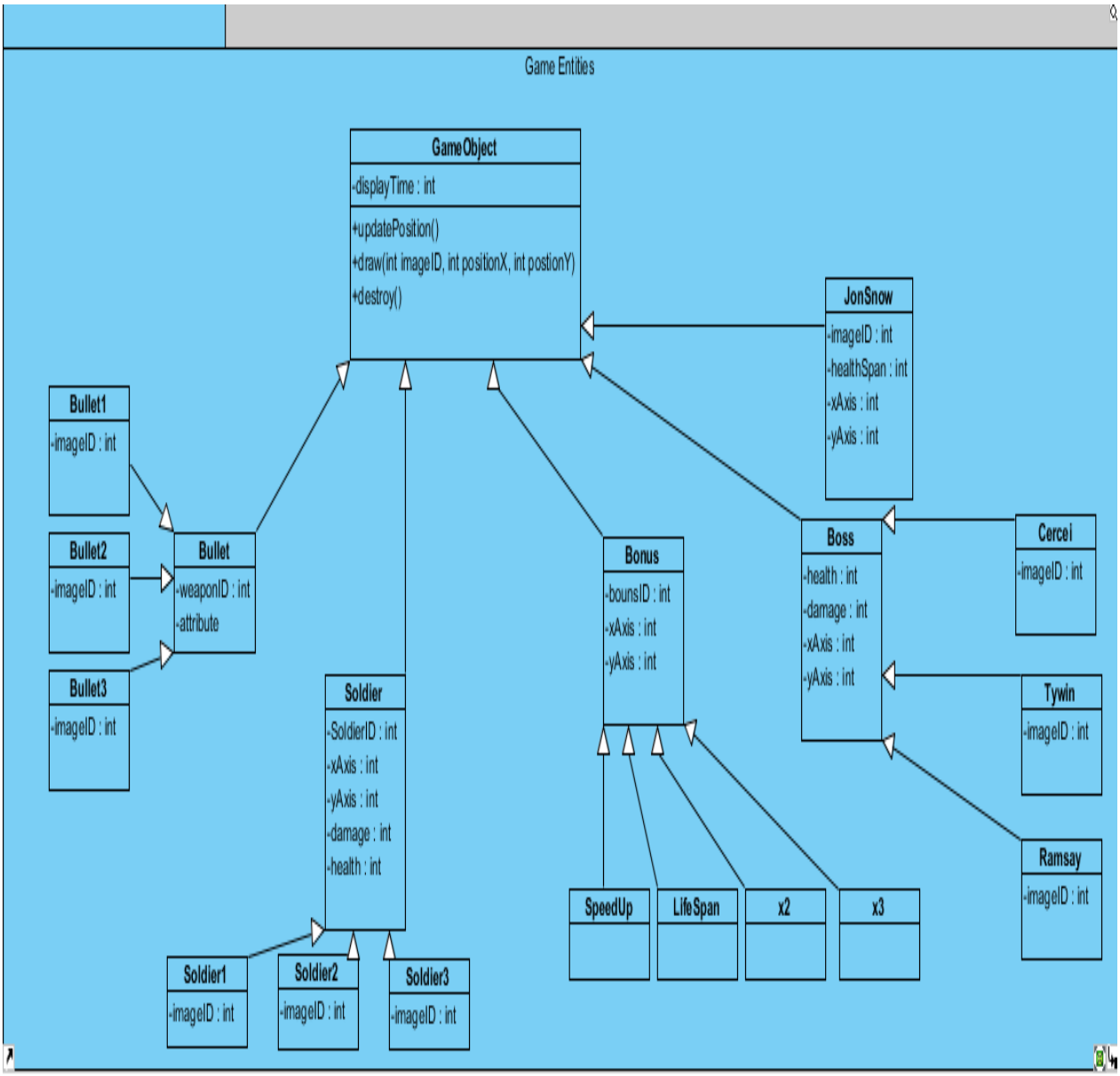

Figure 16: Game Entities Subsystem Interface

3.4.1 Bullet Class

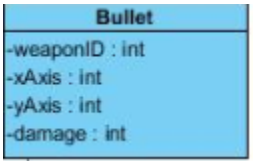

Figure 17: Bullet Class

***Attributes*:**

private int weaponID: This attribute holds the type of weapon, as an integer.

private int xAxis: This holds the x-axis position of weapon.

private int yAxis: This holds the y-axis position of weapon.

private int damage: This attribute holds how much damage the weapon can cause.

3.4.2 Soldier Class

Figure 18: Soldier Class

***Attributes*:**

private int soldierID: This attribute holds the type of soldier, as an integer.

private int xAxis: This holds the x-axis position of soldier.

private int yAxis: This holds the y-axis position of soldier.

private int damage: This attribute holds how much damage the soldier will cause.

private int health: This attribute holds the health points of soldier.

3.4.3 Bonus Class

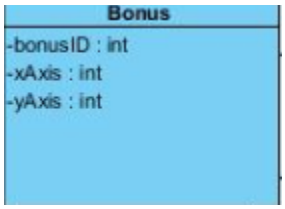

Figure 19: Bonus Class

***Attributes*:**

private int bonusID: This attribute holds the type of bonus.

private int xAxis: This attribute holds the x-axis position of bonus.

private int yAxis: This attribute holds the y-axis position of bonus.

3.4.4 Boss Class

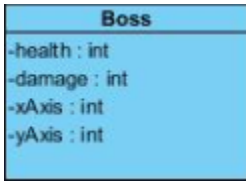

Figure 20: Boss Class

***Attributes*:**

private int health: This attribute holds the health of boss. When it is 0, the boss dies.

private int damage: This attribute holds the damage point of boss. This will be the damage decreased from Jon's health if Jon is hit.

private int xAxis: This attribute holds the x-axis position of boss.

private int yAxis: This attribute holds the y-axis position of boss.

3.4.5 Jon Snow Class

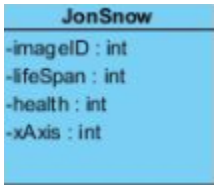

Figure 21: Jon Snow Class

***Attributes*:**

private int imageID: This attribute holds the id of Jon's image that will be used.

private int lifeSpan: This attribute holds the life span of Jon's before soldiers get to him. It serves as time until the user kills the soldiers.

private int health: This attribute holds the health points of Jon.

private int xAxis: This attribute holds the x-axis of Jon's. There is no y-axis since it will be constant.

3.4.6 Game Object Class

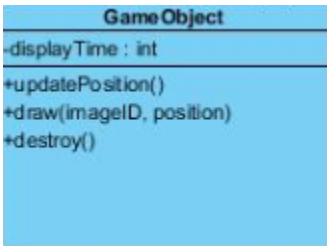

Figure 22: Game Object Class

Attributes:

private int displayTime: This attribute holds the time passed.

Methods:

public void updatePosition(): This method is responsible for the movements of game objects. It updates the x and y axis of the object it is called for. When user presses arrow keys, this method is called for JonSnow object.

public void draw(int imageID, int position): This method is responsible for the position of objects. It redraws objects on updated positions so that objects move.

public void destroy(): This method is responsible of cleaning. When a soldier dies, for example, this method updates the screen with the dead soldier gone.

Descriptions of the Interactions between Classes According to Use Cases

For the start of every use case, first class that is going to be instantiated is "Menu" class. This class has the JButtons for the options to proceed at the start of the program: Play, Options, Help, High Scores and it has also Pause panel that is to pause in the game.

Play: When user wants to start a new game, he just clicks the Play button of Menu and after then, Menu class calls creates the GameEngine class. In GameEngine class, to start the game, firstly, initializeGame() method is called and the game parameters are initialized and GamePanel is set as the current JPanel of the Frame Menus is set as GamePanel. GamePanel will be set according to values of each level, as RunningJon has 3 levels. Therefore; when game started, from the level manager, GameEngine class will get the data of level one and it will render its componenets along the game according to this data. This data includes where are the starting points of all the enemies, Jon and the Bosses. It also includes the movement pattern of each enemy. Then, timer is started. To maintain the gameplay, updatePoints() method of the GameEngine class is always called along the game to reflect the changes immediately to GUI. The changes are made according to user's input and these inputs will be got by the use of InputManager class whose instance is created when the user clicked Play button. When the game proceedes, if there is a collision between two types of object, the methods of the CollisionManager is called: isCollided(gameObject1, gameObject2), checkCollisions(). The collision could be friendly; it is between a bonus and Jon or deadly; between jon and enemy or jon and bullet. This difference is also checked in isCollided method. Whether the game is over or not over is always checked in

gameLoop() method of the GameEngine, it checks whether user has the enough lives to continue or not.

Help: In Menu class, if user clicks the Help button, Help panel will be set as the current panel. It will display the all necessary information and the tips about the game and it has a back button. When clicked, Menu panel is again loaded.

Credits: In Menu class, if user clicks the Credits button, Credits panel will be set as the current panel. It will display the information about the contributions of the people to the project and some reference points. When clicked, Menu panel is again loaded.

High Scores: In Menu class, if user clicks the High Scores button, High Scores panel will be set as the current panel. It will display the name of the people who made a high score in the game. When clicked, Menu panel is again loaded.

Pause: In Game, if user wants to pause, he clicks to esc button and this is understood by input manager and timer is stopped. The current panel is set as Pause panel. It will just display Pause text. When clicked esc button again, the timer is started and current panel is set as GamePanel again.