  
  
  
  
CS 319 - Object-Oriented Software Engineering  
Analysis Report  
RUNNING JON

Group  
ERKAN ÖNAL

NİHAT ATAY  
BARIŞ ARDIÇ

MERT KARA

İçindekiler

[2 1.Introduction 3](#_Toc466762987)

[1.1 Purpose of the System 3](#_Toc466762988)

[1.2 Design Choices 3](#_Toc466762989)

[1.3 Design Goals 4](#_Toc466762990)

[2. Software Architecture 5](#_Toc466762991)

[2.1 Subsystem Decomposition 5](#_Toc466762992)

[2.2 Architectural Styles 7](#_Toc466762993)

[2.3 Hardware/Software Mapping 8](#_Toc466762994)

[2.4 Data Management 8](#_Toc466762995)

[2.5 Access Control and Security 8](#_Toc466762996)

[2.5 Boundary Conditions 9](#_Toc466762997)

[3. Subsystem Services 9](#_Toc466762998)

[3.1 Design Patterns 9](#_Toc466762999)

[3.2 User Interface Subsystem Interface 11](#_Toc466763000)

[3.3 Game Management Subsystem Interface 12](#_Toc466763001)

[3.4 Game Enitities Subsystem Interface 13](#_Toc466763002)

[3.5 Classes 14](#_Toc466763003)

# 1.Introduction

## Purpose of the System

The main purpose of this game is to entertain like any other game does however Running Jon is a video game in development as a Object Oriented Programming course project. Therefore, the purpose of the project group is to accomplish and maintain a successful development process with the desired object oriented programming practices. As a result, the development process is at least as crucial as the final product.

## 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.

## 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/Playability:**

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 hence playability.

# 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.

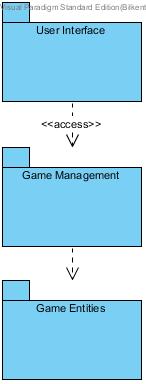
Game Management subsystem is responsible of the management of the game using Game Entities. This subsystem manages how the objects will interact with each other, how the game will progress, etc. In other words, it is the subsystem of the logic of the game. All the classes in Game Management are similar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

## 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 is User Interface, has the highest hierarchcy 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.

## 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, since there is no need for a big database, we will use .txt files to load high scores. Therefore, the operating system should support .txt files.

## 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**

The game will come with an executable .jar file so there will not be a 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.

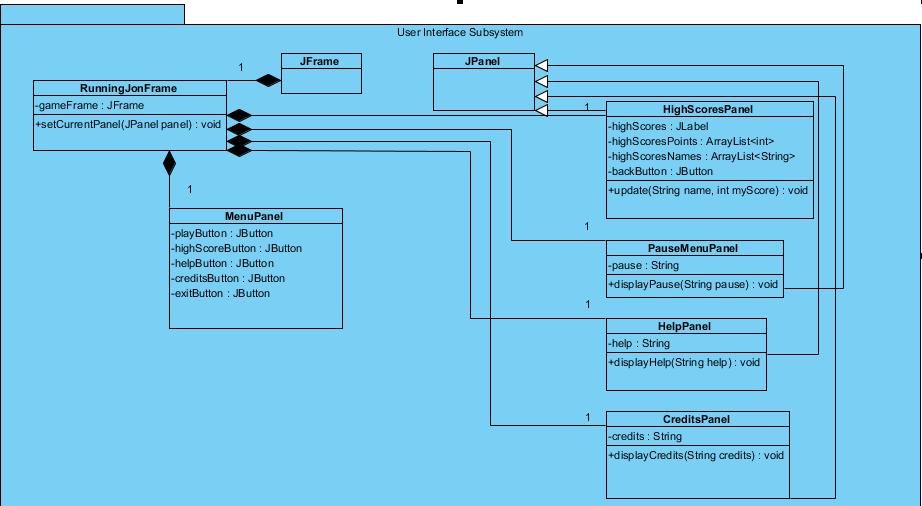
In our design, we use the singleton design pattern for GameEngine class because it is our manager class for the game to maintain the actions in the game. It will detect any most of the changes and update the different values of the game, which are on the Game Management subsystem. Therefore; will need 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:**

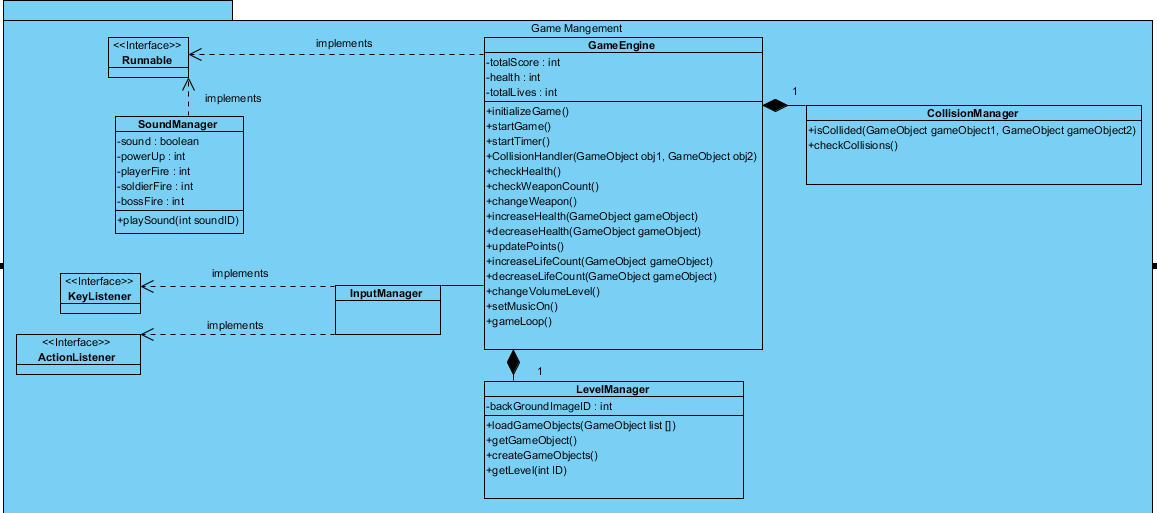
Façade Design pattern is a design pattern that hides the complexities of the system and provides an interface to the client using which the client can access the system. It is a structural pattern. This pattern adds an interface to existing subsystem to hide its complexities.

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.

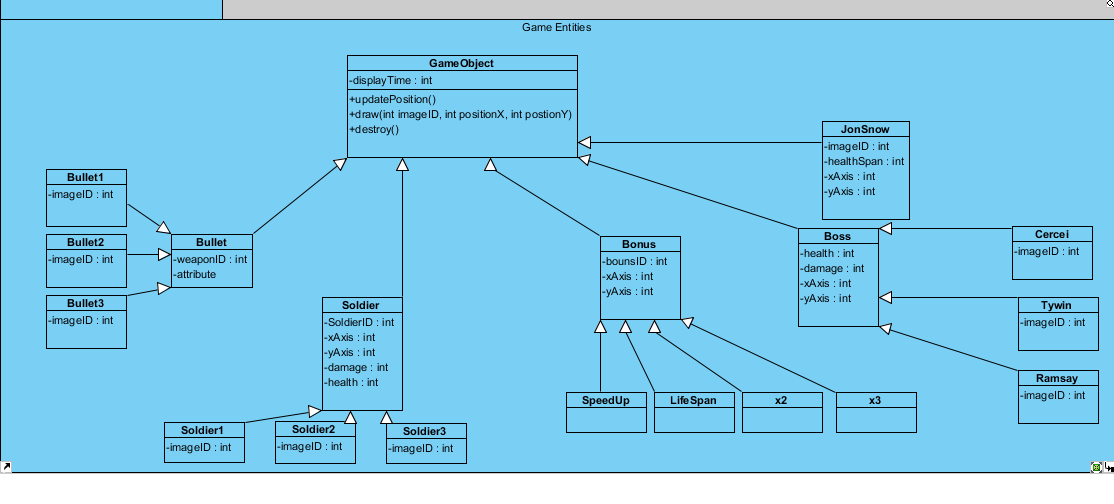
## 3.2 User Interface Subsystem Interface



## 3.3 Game Management Subsystem Interface



## 3.4 Game Enitities Subsystem Interface



## 3.5 Classes

**Menu Class**

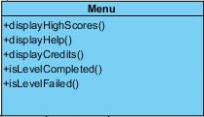


Figure: Menu Class

***Methods:***

**public void displayHighScores():** This method adds highScores panel to frame to display highest 10 scores in the game.

**public void displayHelp():** This method adds Help panel to frame to display how to play instructions and information about bonuses.

**public void displayCredits():** This method adds Credits panel to frame to display it.

**public void isLevelCompleted():** This method checks whether level is completed or not.

**public void isLevelFailed():** This method checks whether level is failed or not.

**Collision Manager Class**



Figure: 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.

**Game Engine Class**

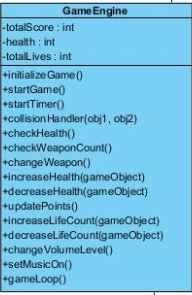
****

Figure: 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

healthwhen 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.

**Sound Manager Class**

****

Figure: Sound Manager Class

***Methods*:**

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

**Input Manager Class**

**C:\Users\Nihat\Desktop\Lectures\CS 319\Running Jon\Classes\Input Manager Class.png**

Figure: Input Manager Class

**Level Manager Class**

****

Figure: 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.

**Bullet Class**

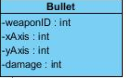
****

Figure: 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.

**Soldier Class**

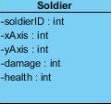
****

Figure: 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.

**Bonus Class**

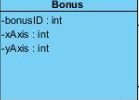
****

Figure: 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.

**Boss Class**

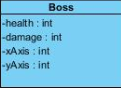
****

Figure: 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.

**Jon Snow Class**

****

Figure: 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.

**Game Object Class**

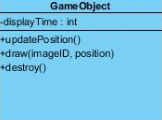
****

Figure: 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.