
# CS 319 - Object-Oriented Software Engineering

## Final Report

***Armageddon***

Group 4


| | |
|---|---|
| Özge Karaaslan | *21301835* |
| Onatkut Dağtekin | *21300801* |
| Teyfik Rumelli | *21102161* |
| Saner Turhaner | *21100475* |


**Deadline: 07/12/2015**

*Course Instructor: Özgür Tan*

# Contents

## 1. Introduction

As a group we intended to design and implement a game called *Armageddon* which is a multi-directional shooter space game based on destroying enemy spaceships, asteroids and surviving. There are five levels in the game. Transitions between sequential levels are smooth such that game continues when level goes up. If player passes these five levels successfully the game ends. If player is destroyed by enemy spaceships or asteroids at three times the game is over. A level is cleared when the player destroys all enemy ships which are coming in certain number, otherwise level continues until player destroys ships moving on the screen or player's ship is destroyed with the result that game is over.

Aim of this project is to make users feel the joy of ability games with minimal and smooth graphics. Armageddon is a kind of arcade game; playing Armageddon needs coordination of mentality and sleight of hand together. So users should think faster and decide faster while playing the game. As a group our principle while creating the idea of this game is keep it as simple as possible while making it as much as entertaining and irreplaceable.

In this report we are going to explain the overview of the game, we specify the gameplay contents and we are going to indicate general rules of Armageddon. This report includes the architectural patterns which will be used in our game. As an analysis report it also gives detailed information about functional requirements, non-functional requirements and constraints of the project. In the "System Models" part; inclusive scenarios, use-case models, object models, dynamic models and interfaces of the game will be analyzed. At the end of the report, there are "Glossary" and "References".

## 2. Requirements Analysis

### 2.1. Overview

At the beginning of our game, player controls a spaceship which moves from left to right. First, there are just asteroids in the space which are coming towards our spaceship. Player can skip out from these objects; spaceship can fly any direction on the screen by pressing direction buttons in keyboard. Our essential mission is keeping spaceship alive. Then enemy spaceships come within few minutes. At the very first level of the game enemies are not very harmful. These harmless spaceships coming first are generally fling from right to left and do not shoot, but if we hit any of these spaceships it will decrease our shield strength. In progress, enemies are going to be more harmful starting with first level.

The player, controlling a spaceship, must destroy twenty enemy spaceships in each level in order to level goes up. These twenty spaceships are certain ships which do not go away until player destroys them. They come from right side of the screen like others but when they come they start moving at y-axis on the right side and shooting. Apart from that, fuel of the spaceship that player controls is limited and it decreases constantly. Enemy spaceships enter the playfield from the right side of the screen. The aim of the enemy spaceship is simply to destroy the player's ship. The level is completed when all the enemy spaceships are destroyed. During the game, there will be space station in each level that supply fuel, repair the shield and give specific ammunition which are controlled by one of “a” or “s” buttons of keyboard. Extra ammunition will be shown on the left down of the screen with its number and specified button.

While the game is on, the game music is playing constantly and there will be fire and explosion sounds. The player can stop the game at any time and simple pause menu appears when the game is paused. Player can adjust music and sound levels using this panel and continue afterwards. Moreover, after every level is passed a short password reflects on the screen for the game can be loaded later from that level. It allows the player to resume the game afterwards via

“Enter a Password” option of main menu. User collects variety of points when it destroys space objects. These points are recorded after the game is over if user’ points are enough to get into the first ten highest scores. And also there is “High Scores” option in the main menu for players to see top ten highest scores.

Armageddon consists of 5 levels. At every level the enemy spaceship types are varied. Their appearance will also differ too. In addition to this our spaceship ammunition will be enhanced level to level. For instance, default type gun destroys GoblinY enemy spaceships with 3 times shoot. But Laser destroys them by 2 shoots. Besides, rocket destroys a GoblinY enemy at first shoot. Shoot appearances will also differ in the game.

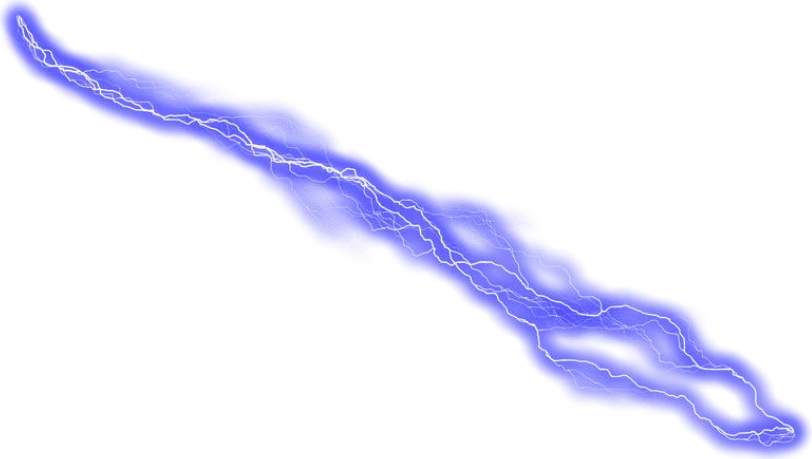
### 2.1.1. Ammunition Types

- Default: Default type of ammunition of the player’s spaceship. It is weakest ammunition of the game. Its power of fire is 1 unit.

- Laser: This weapon will be given at second level, in the space station. Its power of fire is 2 units, so it is two times more powerful than default ammunition. When it is given it replaces with default ammunition. Therefore it is not bonus weapon.

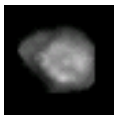
- Rocket: It is given at third level, in the space station. Station gives one hundred rockets as bonus ammunition in this level. Its power of fire is 3 units.

- Electric: It is given at fourth level, in the space station. It is not numerable weapon. When the player takes this weapon there will be charging bar on the underside of the screen. When this bar is fully charged player can use this weapon and when it is used lightning flashes and it destroys all of enemies on the screen at that time.



### 2.1.2. Enemy Types

- Asteroid: Space objects which are coming towards player's spaceship. When player hits one of them it decreases vast amount of shield strength.



- GoblinX: It is most harmless enemy spaceship in the game. It flies through right to left without shooting. But when it hits shield will be damaged tragically.

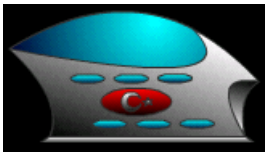
- GoblinY: Enemy spaceship that comes from right side of the screen but never goes away until player destroys it. These spaceships come in certain number for certain levels which is at most twenty. When players destroy all of them level goes up.

- GoblinZ: Just like GoblinY spaceship, but it goes from right to left on up and down sides of the screen. It shoots crosswise but appearance is same with GoblinY.
- BomberX: This is a powerful enemy spaceship which launches rocket. This weapon is same with our ammunition which is called rocket. Just like GoblinX these spaceships goes through right to left on the screen but unlike GoblinX, they shoot on their routes.

- BomberY: Just like GoblinY these are coming from right side of the screen and then moving towards y-axis until they die. They are coming in limited numbers for each level. Appearance of them is same as BomberX.
- BomberZ: Its way and shooting route is same as GoblinZ and its weapon is same with BomberX and BomberY. Appearance is slightly different than other Bomber's.

### 2.1.3. Special Types

- Station: Station is where user can load fuel, repair space ship's shield and get some sophisticated bullets.



## 2.2. Functional Requirements

- User will be able to move the spaceship with direction keys and shoot with A,D and space key.
- User can change the settings of the game such as muting the sound.
- User can check the highscore board.
- User can stop the game and continue whenever they want.
- User can enter a password to start from a different level.
- User can see the credits from the main menu.

## 2.3. Non-Functional Requirements

- To increase performance we will use limited hard drive space and store passwords and high scores on text files. By reading those information from the text files it will be easier to create a table and use it whenever necessary by doing so the application will not waste memory.
- Since the game is for single player, it will not use a server which means there is no need to wait for response from other sources.
- The game will not have complex structure so the user can understand it easily. Simple controls mean that the user does not need to have lightning reflexes to play this game.
- The game can be opened whenever user wants to play it and since it is a single player game user has only himself/herself to compete against.
- The graphics of the game will be smooth and it will contain animations to make it more user firendly and enjoyable.

## 2.4. Constraints

- The game's language will be English.
- The game will be developed and implemented in Java.
- The space this game will occupy will not go beyond 30 MBs.

## 2.5. Scenarios

**Scenario #1: Starting Game**

Player Saner requests to start game by clicking “New Game” button from Main Menu. After that system initializes game panel into the frame and game panel sends message to system to create objects. System initializes objects and gets images of objects; background, spaceship that player uses, enemy spaceships, space station and asteroids from the file directory of the game. Then system locates all objects to appropriate locations indicated in the panel according to level number. While loading the level, system reads last settings from the text file. Lastly, system starts the game loop and repaints all the objects related to Level and updates the game panel.

**Scenario #2: Shooting War**

Player Ozge requests to start the game by clicking “New Game” button from Main Menu. Assuming that steps of the previous sequence diagram are performed, in the system whole game dynamics are arranged; current location of the spaceship that Ozge uses is taken, space objects and bullets of the enemy spaceships that are on the screen are manipulating, spaceship that Ozge uses is moving and shooting. System also checks collisions; assuming that player Ozge’s spaceship has been collided with the bullet of the enemy spaceship and is damaged, Ozge directs the spaceship using direction keys from the keyboard in order to avoid the enemy bullets. After that Ozge shoots and her bullet is collided with the enemy then the enemy spaceship is destroyed.

**Scenario #3: Changing Settings**

Player Ozge request to start a game by pressing the “New Game” from Main Menu. Assuming that the steps of the Start Game sequence diagram are performed, Ozge wants to change the game setting as she desires. In order to do that Ozge pauses the game by pressing Esc key from the keyboard. Small Pause Menu shows up on the game screen and game is stoped. Ozge changes sound

and music volumes by sliding bars on this panel. New volume settings are arranged by the Pause Menu. After that Ozge clicks to “Return to Game” button and game continues.

**Scenario #4: Entering Station**

Player Saner request to start a game by pressing the “New Game” from Main Menu. Assuming that the steps of the Start Game sequence diagram are performed, Saner's spaceship runs out of fuel and his shield is damaged during the game. Saner presses down button from the keyboard and spaceship enters the station in order to repair his shield and load fuel. Game is paused when spaceship is in the space station. Then it gets an extra bullet from space station which is specific to the current level. After a certain period spaceship leaves the station and game continues.

## 2.6. Use Case Model

### 2.6.1. Use Case Diagram

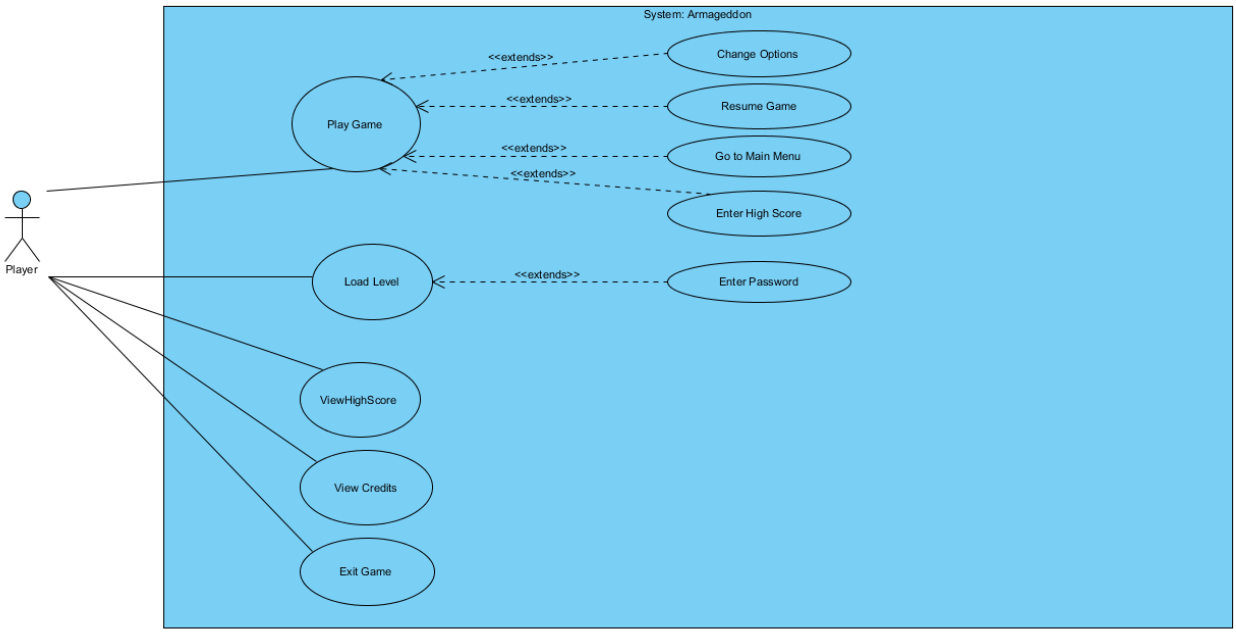



Figure 1: Use Case Diagram for the System

### 2.6.2. Use Case Descriptions

Use Case - 1

**Name:** PlayGame

**Actors:** Player

**Entry Condition:** The game is started and on the main menu player should choose “Play Game” operation.

**Exit Condition:**

- All the five levels are completed successfully.
- Player has lost all of his/her lives.
- Player stops the game by pressing “Esc” key from keyboard and decides to return main menu by mouse left click.

**Main Flow of Events:**

1. Level 1 is constructed by default and player starts to play game.
2. Player uses direction keys from keyboard for direction of the spaceship and “a”, “s”, space buttons from keyboard for shooting.
3. Spaceship shoots and destroys enemy spaceships and it is damaged by fires of the enemy spaceships and asteroids that hit it.
4. Fuel of the spaceship is decreasing constantly, it enters to space station and station gives extra ammo and fuel to spaceship.
5. Player finishes all the levels successfully. Greeting message shows up at the end.
6. Score of the player is displayed. If the score is higher than tenth high score, player will be asked to enter a nickname. If so player enters a nickname (If score is not higher enough then the game returns main menu automatically).

7. The score is recorded and high score table is updated accordingly.
8. Player returns to the main menu by clicking related button on the high score menu.

**Alternative Flow of Events:**

- Player has lost all of his/her lives. Game over message is displayed on the screen.Then high score menu shows up automatically. Player returns to the main menu by clicking related button on the high score menu.
- Player stops the game. Small options panel is displayed on the game screen. Player returns to Main Menu by clicking “Return Main Menu” button on this panel.

Use Case - 2

**Name:** ChangeOptions

**Actors:** Player

**Entry Condition:** The game is started and player stops the game by clicking “Esc” key from keyboard.

**Exit Condition:**

- Player returns to the Main Menu by clicking “Return to Main Menu” button.
- Player returns to the game by clicking “Return to Game” button.

**Main Flow of Events:**

1. Small options panel is displayed on the center of the game screen. On this panel there are three buttons: “New Game”, “Return to Main Menu” and “Return to Game” and also two sliding bars that are related to sound and music voices.
2. Player navigates through provided menus.
3. Player changes music and sound voices by sliding these bars.
4. New settings will be saved by the system.

5. Player returns to the main menu by clicking “Return to Game” button.

**Alternative Flow of Events:**

- Player does not change the settings and returns to game by clicking “Return to Game” button.
- Player clicks the “New Game” button and level 1 is constructed by the system.
- Player returns Main Menu by clicking “Return to Main Menu” button.

Use Case - 3

**Name:** ViewHighScores

**Actors:** Player

**Entry Condition:** The game is opened and player is on the main menu to choose an operation.

**Exit Condition:** Player returns to the Main Menu by clicking “Return to Main Menu” button.

**Main Flow of Events:**

1. Player selects the “View High Scores” option.

2. System displays top 10 high scores with their user names.

**Alternative Flow of Events:**

- Player wants to return back to the Main Menu and presses “Return to Main Menu” button.

Use Case - 4

**Name:** LoadLevel

**Actors:** Player

**Entry Condition:**

- The game is opened and player is on the main menu to select an operation.
- Player is already playing a game.
- Player already passed at least one level.
- The player has the appropriate password to start at the desired level.

**Exit Condition:**

- Player enters the password correctly and load level.

**Main Flow of Events:**

1. Player selects the “Enter a Password” option to continue to play.
2. System asks for a password.
3. Player enters the password.
4. Player continues to play the game from the point where s/he passed.

**Alternative Flow of Events:**

- Player cannot find a level to load since there is none and returns to the main menu.
- Player cannot enter the correct password, an error message is displayed.

## 2.7. User Interfaces

In this part, panels and visual of objects will be given.

### 2.7.1. Main Menu

When user runs the game menu screen will be shown first. Menu screen contains New Game, Password, High Scores and Credits options. By clicking one of these options, user can move desired panel.


Figure 2: Main Menu

### 2.7.2. Password

If user selects “PASSWORD” option, this screen will be shown. In this screen, there are an input box, which takes password from user, and a keyboard. By the help of this keyboard, user can enter the password given at the end of the game. After entering the password, if the entered password is correct, according to entered password user can start game from specific level.


Figure 3: Password Menu

### 2.7.3. High Scores

If user selects “HIGH SCORES” option, this screen will be shown. In this screen, user can see table of the best ten score ever reached before. These scores are on display with the “User Name”-name of user-, “Date/Time” -date of this score was achieved- , “Level” -last reached level- and “Score”-total points user gained while playing-.

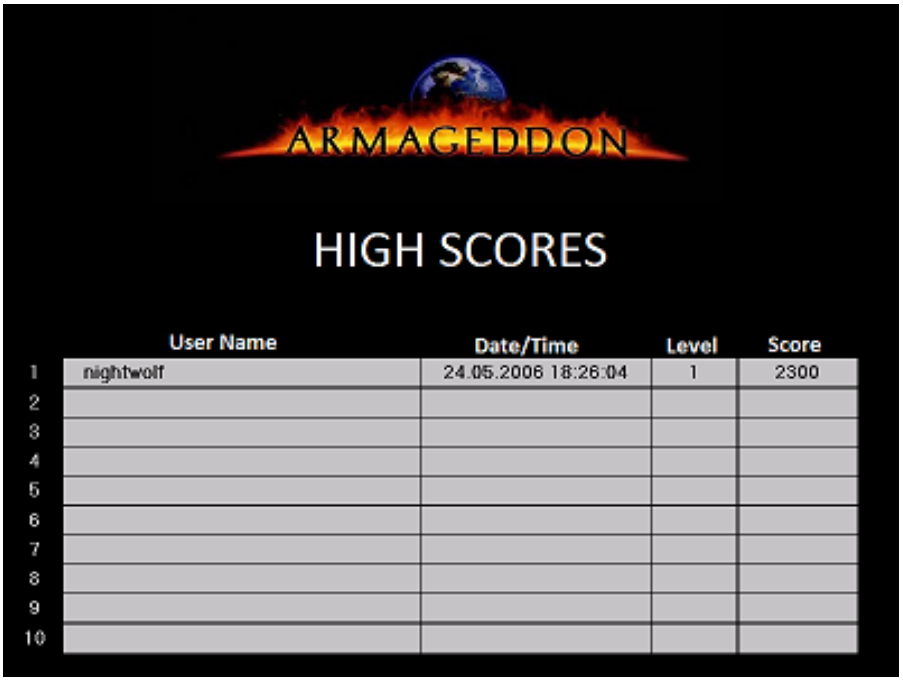



Figure 4: High Scores Table

### 2.7.4. Credits

When user selects “CREDITS” option, this screen will be shown. In this screen, there will be a textbox which contains some brief information about the game and development team.

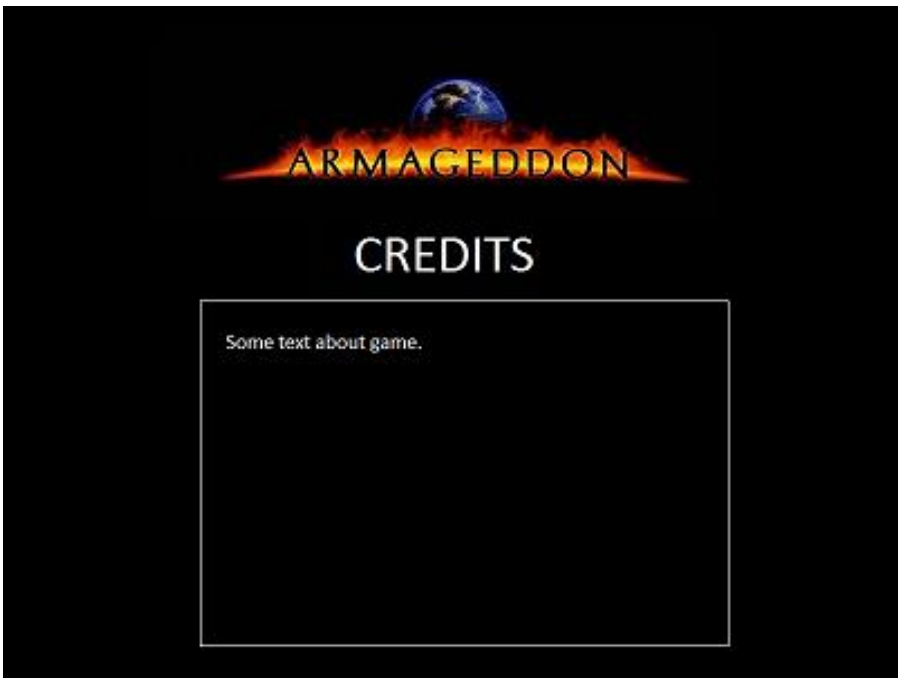



Figure 5: Credits

### 2.7.5. Game

When user selects “NEW GAME” option, this screen will be screened. This screen is where user plays the game so it contains various objects which will be shown one by one below. Left top of the page, level of game and total score that user collects will be shown while playing.


Figure 6: Gameplay View

### 2.7.6. Pause Menu

User can reach this screen from game menu by pressing “Esc” button. With the help of this screen user can modify music settings and sound settings. Also new game can be started. After the completing desired settings, user can back to game by clicking “Return to Game” or back to main menu by clicking “Return to Main Menu”.


Figure 7: Pause Panel

# 3. Analysis

## 3.1. Object Model

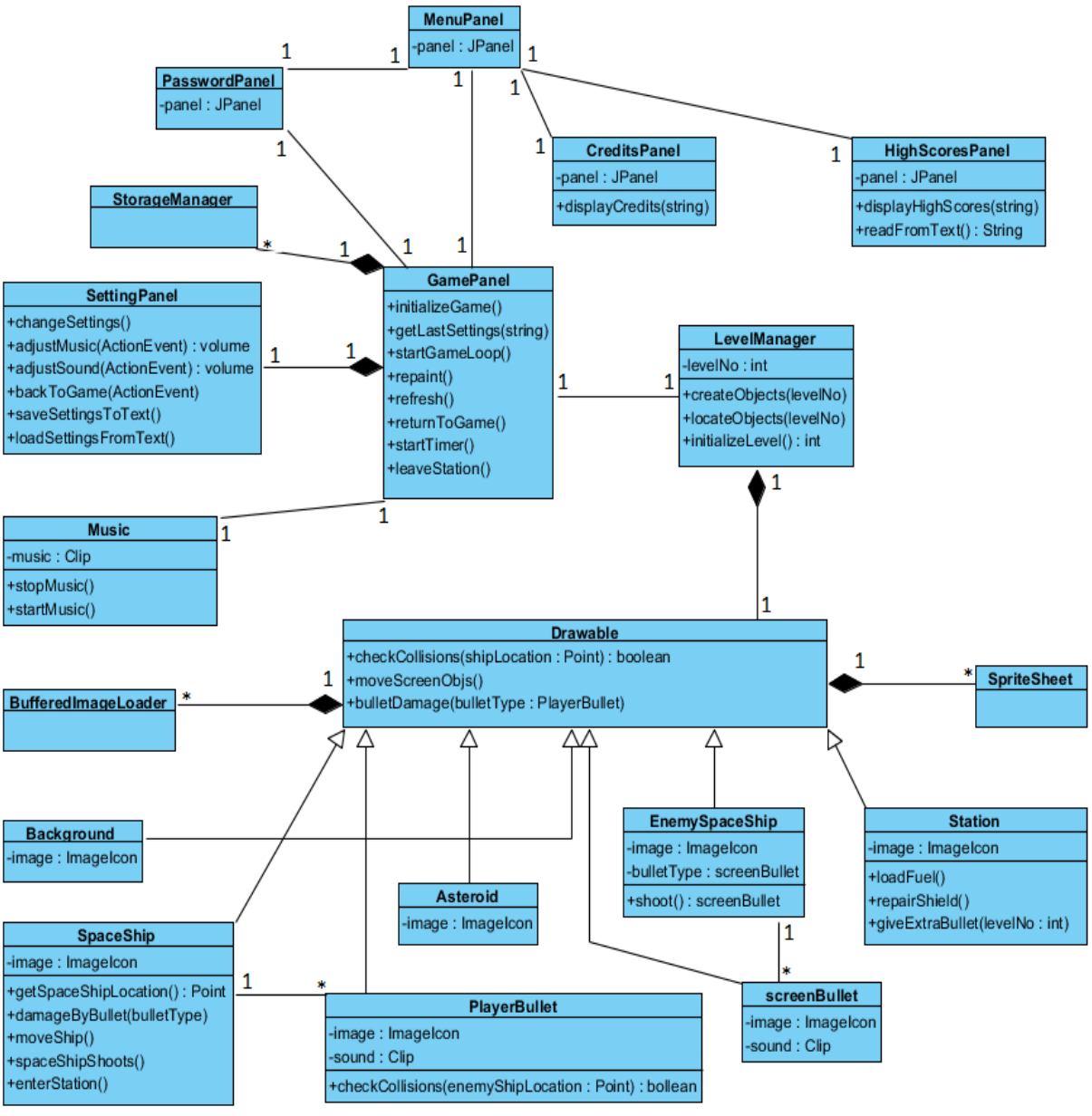



Figure 8: Class Diagram for the System

The object model of ARMAGEDDON game is illustrated above. This diagram contains 17 classes.

- **GamePanel:** This is the most significant class. This class is where game is organized and interacted with user at most. When user plays the game images are shown by this class panel. GamePanel controls inputs and object relations.
- **LevelManager:** LevelManager class manages game difficulty according to level number. It decides type of enemies, their numbers and holds them.
- **screenObject:** screenObject is the most important class for visualization because this class is where collisions are checked according to location of objects, and also where objects can move. Below this class there are seven entity classes. These classes have different image icons, speed, direction and some operations.
    - SpaceShip: SpaceShip class hold image of spaceship that user controls. With the help of this class user can move and shoot to destroy enemies.
    - PlayerBullet: When the user shoots, this class is called and move until a collision occurs.
    - Asteroid: Asteroid is a type of obstacle.
    - screenBullet: screenBullet is bullets that enemies fire.
    - EnemySpaceShip: This class consist enemy spaceships with different ship types, bullet types and line of action.
    - Station: Station is where user's spaceship enters and gets bonuses.
    - Background: Background is movable object for getting movable background view.
- **Music:** Music class contains game music and sounds of some objects.

Other panels consist of extended JPanel. They interact with user and do some operations that are explained interface part.

## 3.2 Dynamic Models

### 3.2.1 State Chart Diagram

**State Chart Diagram for the Ship:**

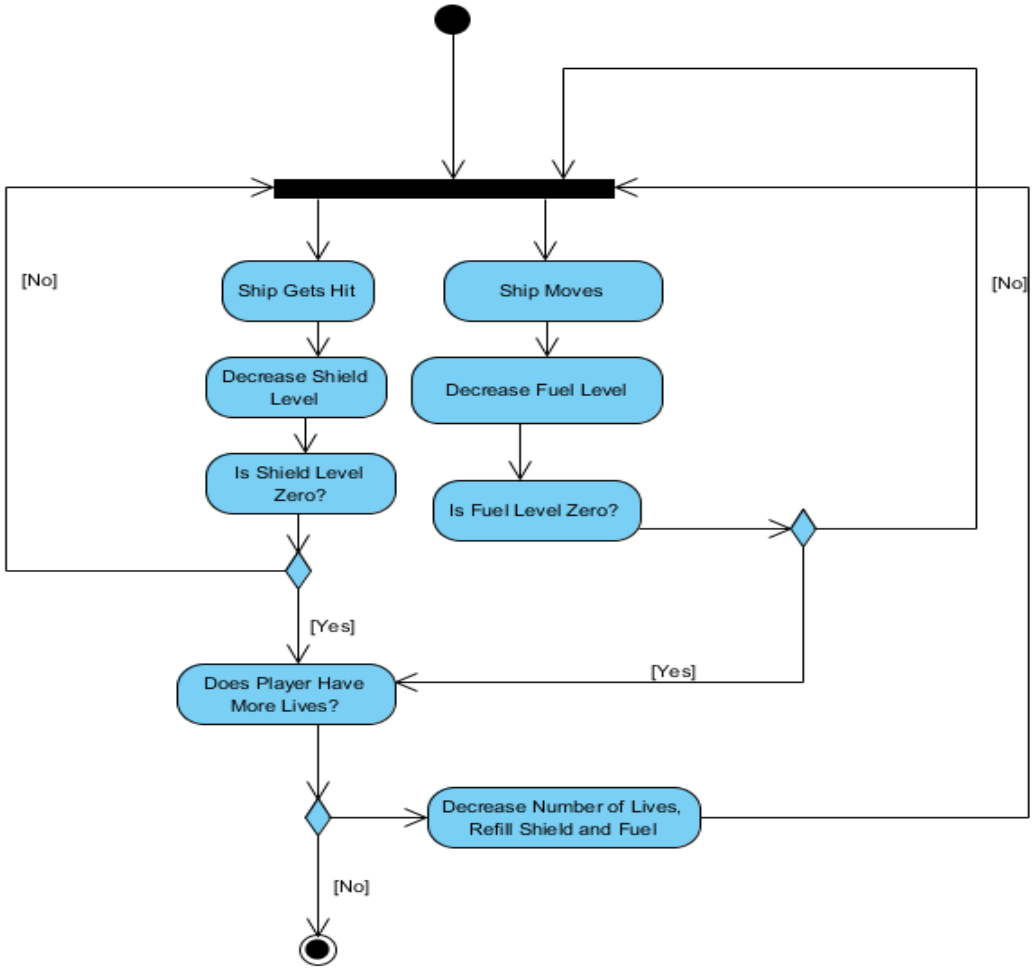



Figure 9: State Chart Diagram for the Ship

This state chart diagram explains the behavior of the ship the player uses. If the user uses direction keys to move the ship the fuel level will decrease. The ship's fuel level will also decrease automatically since the ship moves forward continuously. When the ship collides with one of the enemy ship's bullets its shield will decrease. If either of shield or fuel is fully depleted, one life will be removed from the player and if the player has no more lives, the ship is destroyed and the game ends.

### 3.2.2 Activity Diagram

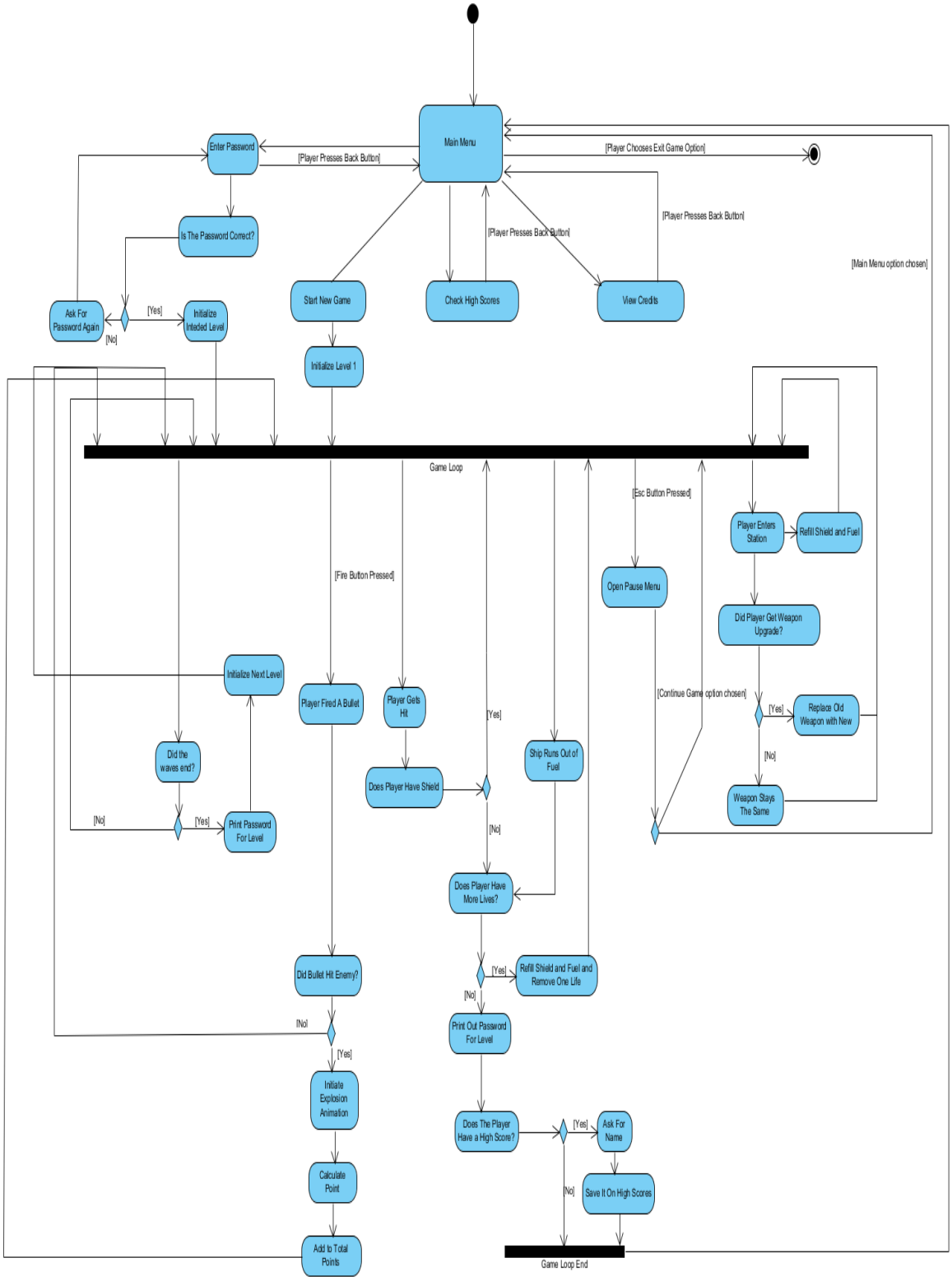



Figure 10: Activity Diagram for the Game

This state chart shows the main flow of the game. The game is initialized from the main menu and the user has options to enter a password, start a new game, view high scores, view credits and exit game. The user can go back to the main menu from all of these options except start a new game option and exit game option. The system checks if the user has entered the correct password. If it is correct, the system initializes the desired level and starts the game loop, else asks for the password again. If the user selects the new game option the system initializes the first level and starts the game loop. If the user has fired a bullet, the system checks for collisions. If a bullet hits an enemy, the system initiates an explosion animation and adds the appropriate point to the score. If the player gets hit, the system checks the current shield level. If the current shield is zero, the system checks the number of lives. If the number of remaining lives is zero the system prints out a password for the level and asks for a name if the player has done a high score and goes back to the main menu. If the player has lives left, the ship's fuel and shield level is refilled. If the player presses the escape the pause menu shows up and the player can change the settings of the game, continue the game or return to the main menu.Else the game continues normally. When a player enters a station the fuel and shield level is refilled and the ship might get a weapon upgrade if it does get an upgrade, the new weapon takes the place of the old one. If a level ends (i.e., the waves have ended) the system checks if the ship has survived, if it does the system initializes the next level. If the user decides to exit the game from the main menu, the game closes.

### 3.2.3 Sequence Diagram

**Scenario #1: Starting Game**

Player Saner requests to start game by clicking "New Game" button from Main Menu. After that system initializes GamePanel into the frame and GamePanel sends message to LevelManager to create objects. LevelManager initializes objects andgets images of objects; background, spaceship that player uses, enemy spaceships, space station and asteroids from the file directory of the game.Then GamePanellocates all objects to appropriate locations indicated in the LevelManager

according to LevelNo. While loading the level, GamePanel reads last settings from the text file. Lastly, GamePanel starts the game loop and repaints all the objects related to Level and updates the game panel.

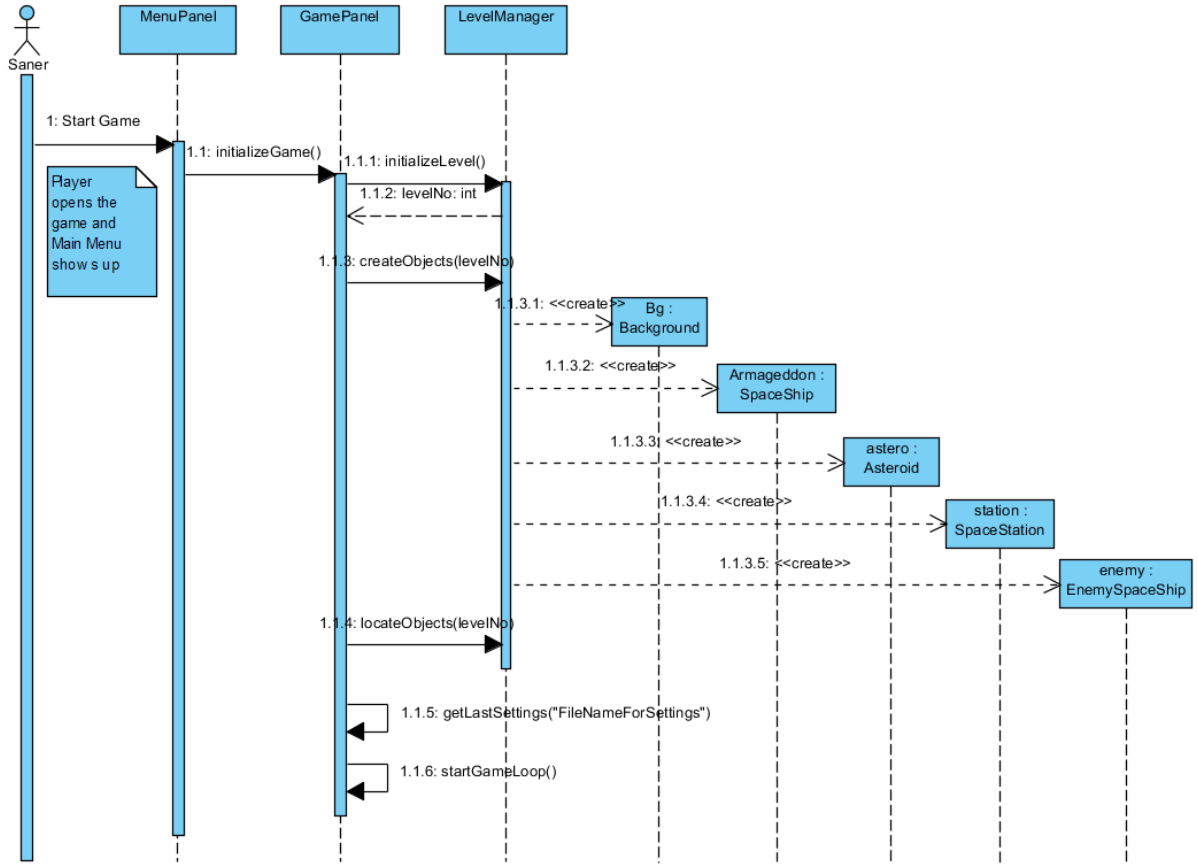



Figure 11: Sequence Diagram for Starting Game Scenario

### Scenario #2: Shooting War

Player Ozge requests to start the game by clicking “New Game” button from Main Menu. Assuming that steps of the previous sequence diagram are performed, in the Game Panel whole game dynamics are arranged; current location of the spaceship that Ozge uses is taken, space objects and bullets of the enemy spaceships that are on the screen are manipulating, spaceship that Ozge uses is moving and shooting. GamePanel also checks collisions; assuming that player Ozge’s spaceship has been collided with the bullet of the enemy spaceship and is damaged, Ozge directs the spaceship

using direction keys from the keyboard in order to avoid the enemy bullets. After that Ozge shoots and her bullet is collided with the enemy then the enemy spaceship is destroyed.

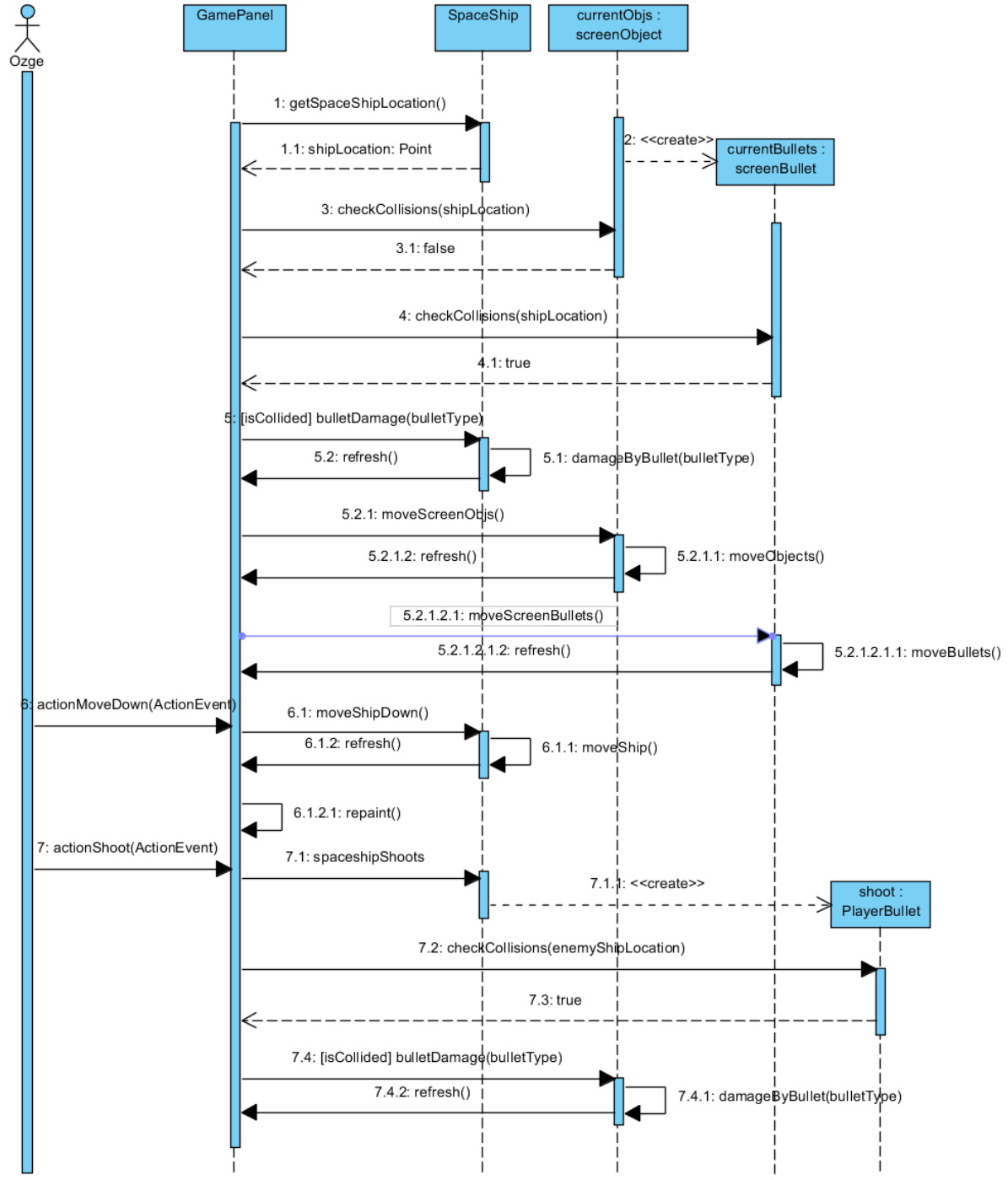



Figure 12: Sequence Diagram for Shooting War Scenario

**Scenario #3: Changing Settings**

Player Ozge request to start a game by pressing the “New Game” from Main Menu. Assuming that the steps of the Start Game sequence diagram are performed, Ozge wants to change the game setting as she desires. In order to do that Ozge pauses the game by pressing Esc key from the keyboard. Small SettingPanel shows up on the game screen and game is stopped. Ozge changes sound and music volumes by sliding bars on this panel. New volume settings are arranged by the SettingPanel. After that Ozge clicks to “Return to Game” button and game continues.

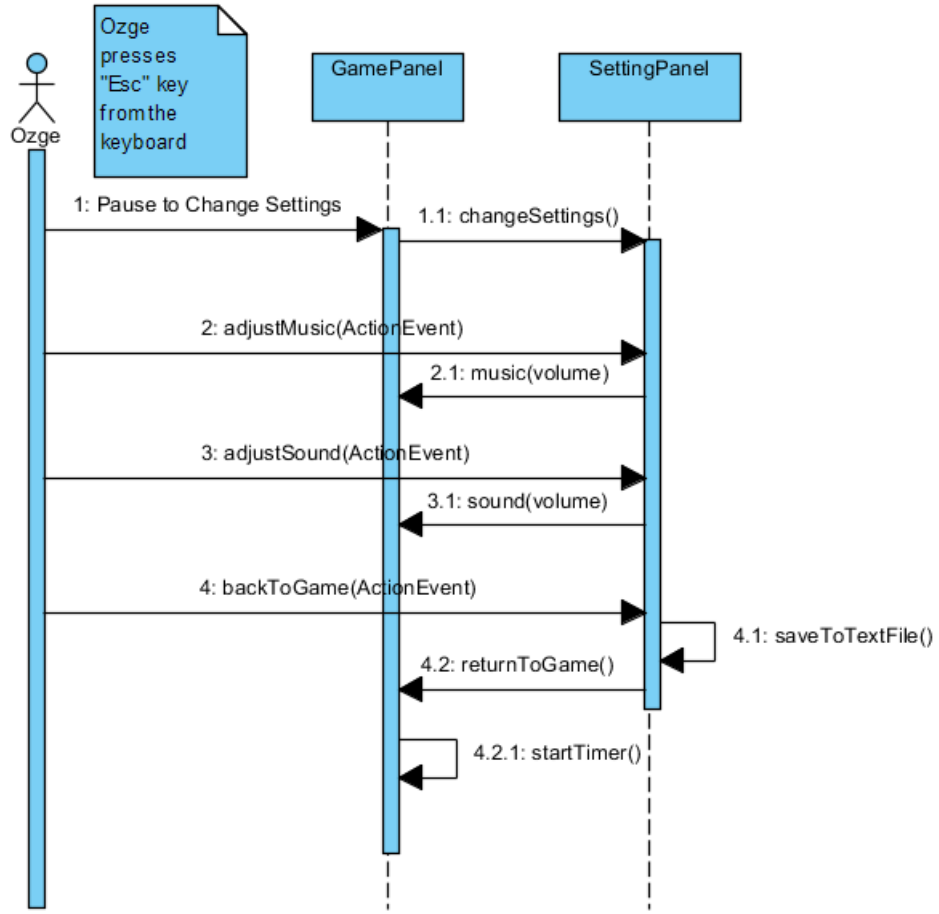



Figure 13: Sequence Diagram for Changing Settings Scenario

**Scenario #4: Entering Station**

Player Saner request to start a game by pressing the “New Game” from Main Menu. Assuming that the steps of the Start Game sequence diagram are performed, Saner's spaceship runs out of fuel and

his shield is damaged during the game. Saner presses down button from the keyboard and spaceship enters the station in order to repair his shield and load fuel. Game is paused when spaceship is in the space station. Then it gets an extra bullet from space station which is specific to the current level. After a certain period spaceship leaves the station and game continues.

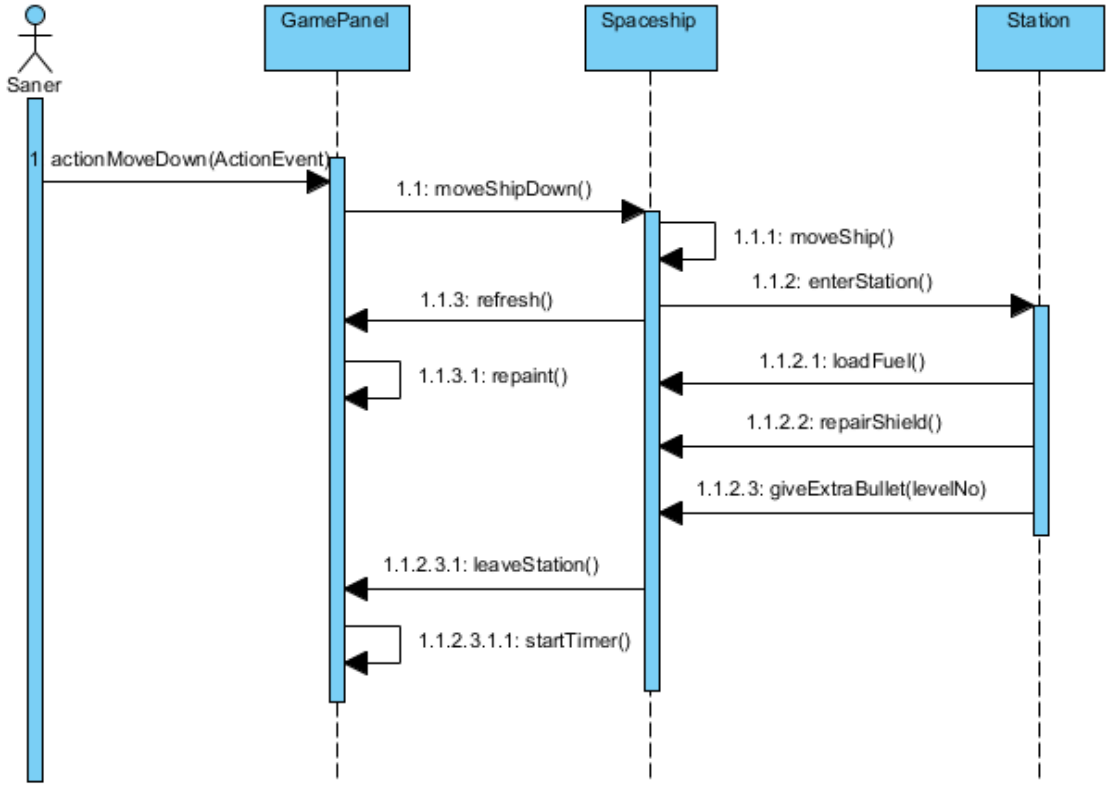



Figure 14: Sequence Diagram for Entering Station Scenario

# 4. System Design

## 4.1. Design Goals

**Ease of Use:** The system for Armageddon will be user friendly and every user can understand the instructions. The interface of the system will make the experience of playing this game more enjoyable.

**Modifiability:** To provide better experience to users, this goal is essential. If a level too hard to pass, developers should be able to modify the conditions of the level to make it playable.

**Ease of Learning:** The instructions of the game is simple to remember and with smooth graphics it even makes it easier and more enjoyable.

**Reliability:** The system should not lose any data if it crashes so we need to make the system reliable.

**Good Documentation:** Good documentation is fundamental for the system for other developers to understand it and make changes on it. Use of comments here is important as well.

**Minimum Number of Errors:** To create a reliable system, errors should be minimum. These errors may cause loss of data and unreliable game play so it is important for the systems continuity to create an error free system. To create this type of system testing for the project is needed.

**Trade- Offs:**

**Functionality:** To create a system that has instructions which can be easily understood by the user, it is inevitable that the system will become simpler which means that there won't be too many complex functions.

**Rapid Development:** There is a rapid development interest in implementation, the effectiveness of the system should be improved with good documentation and minimum number of errors.

## 4.2. Sub-System Decomposition

### 4.2.1. Top Level Decomposition

The system will contain three different subsystems for control manangement, view management and entity management. For view management, there will be two subsystems which will be Game Screen subsystem and Menu Screen subsystem. For entity management(model), there will be two subsystems which are Drawable subsystem and Level subsystem. For control management, there will be a Controller subsystem which consits of Settings Handler subsystem, Game Control subsystem, Storage Management subsystem and Image Loader subsystem. A system that has low coupling would be better however we had to create relationships between subsystems. It can be seen on the diagram that there are connections between Game Screen, Menu, Level and Game Control Subsystem and a relationship between Drawable and Image Loader Subsystem. Façade pattern will be used to reduce coupling as well.

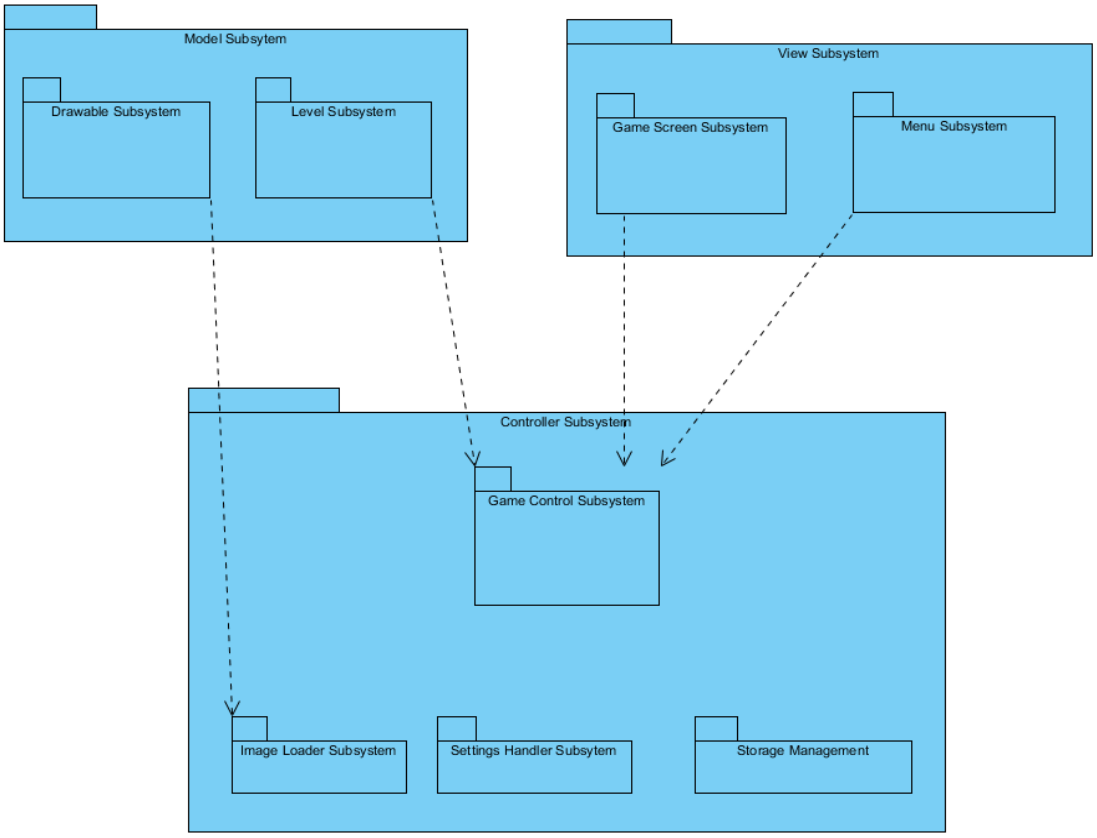


Figure 15: Package Diagram of the System

### 4.2.2. View Subsystem

This subsystem is very important for the system since the user will be interacting with the GUI. This subsystem has two subsystems called Menu Screen Subsystem and Game Screen Subsystem. The general look of the GUI management subsystem is as follows;

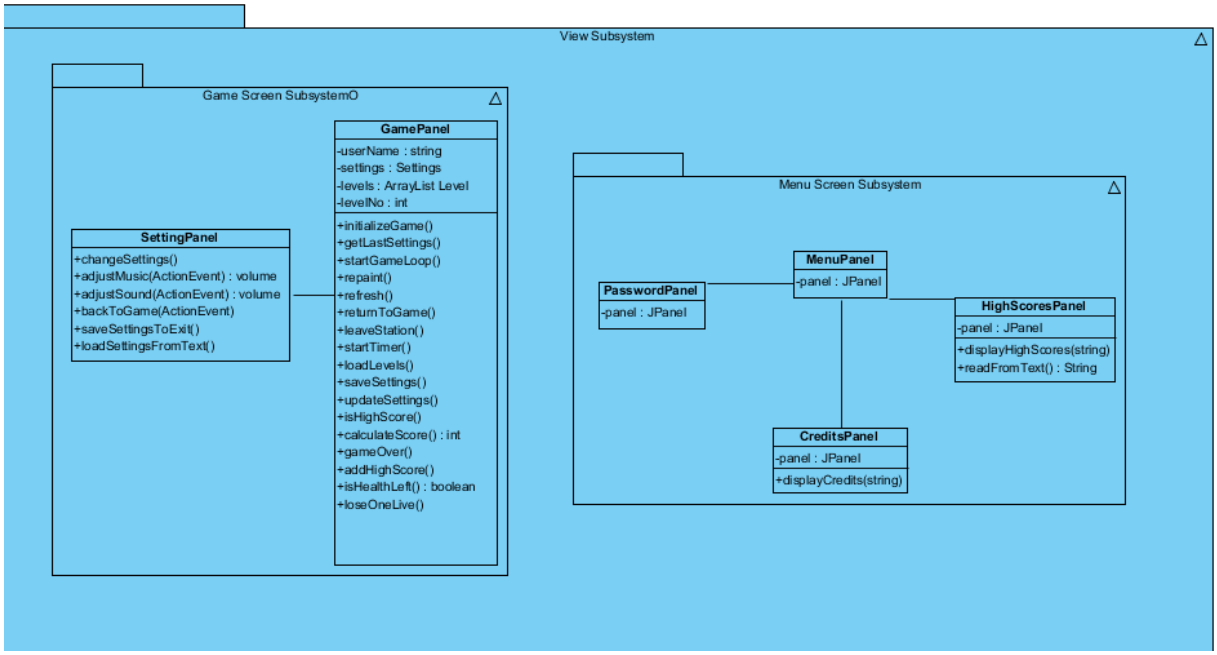



Figure 16 :Package Diagram of View Subsystem

**Game Screen Subsystem**

Game Screen Subsystem represents the game screen which will be shown to the user while he is playing the game. The interactions with objects will occur in the Game Screen Subsystem. Therefore, this subsystem contains only the GamePanel class which will be used to draw images of the objects on the screen and will show the interactions between objects to the user graphically. The package diagram of the Game Screen Subsystem could be found below.

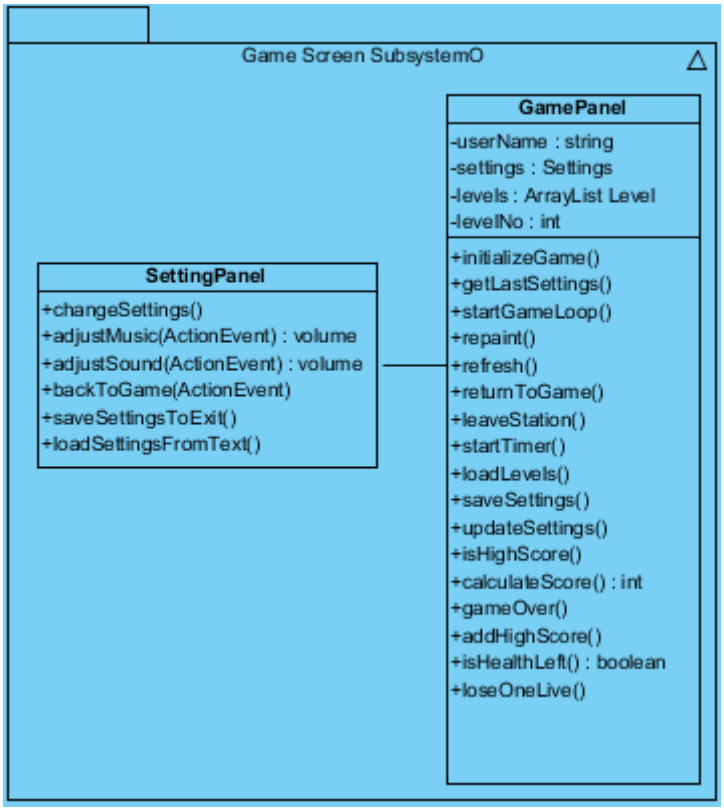



Figure 17 : Package Diagram of Game Screen Subsystem

**Menu Subsystem**

This subsystem represents the menu manager of the game, classes in the subsystem provides graphical representation of menu navigations. User can interact with main menu, credits, high scores, password menus.

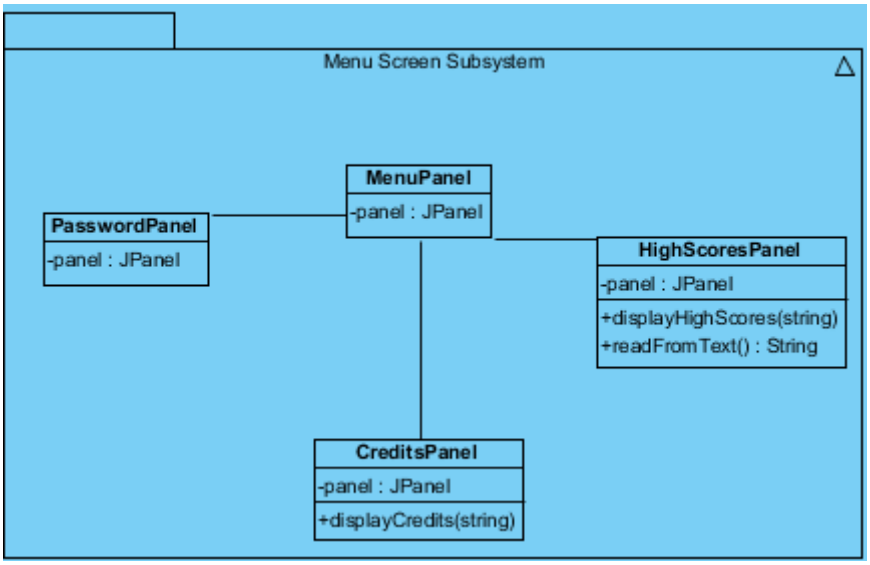



Figure 18 : Package Diagram of Menu Subsystem

### 4.2.3. Controller Subsystem

Controller subsystem consists of four different subsystems. These are Game Control Subsystem, Storage Management Subsystem, Settings Handler Subsystem, and Image Loader Subsystem. The package diagram of the Game Management Subsystem could be found below:

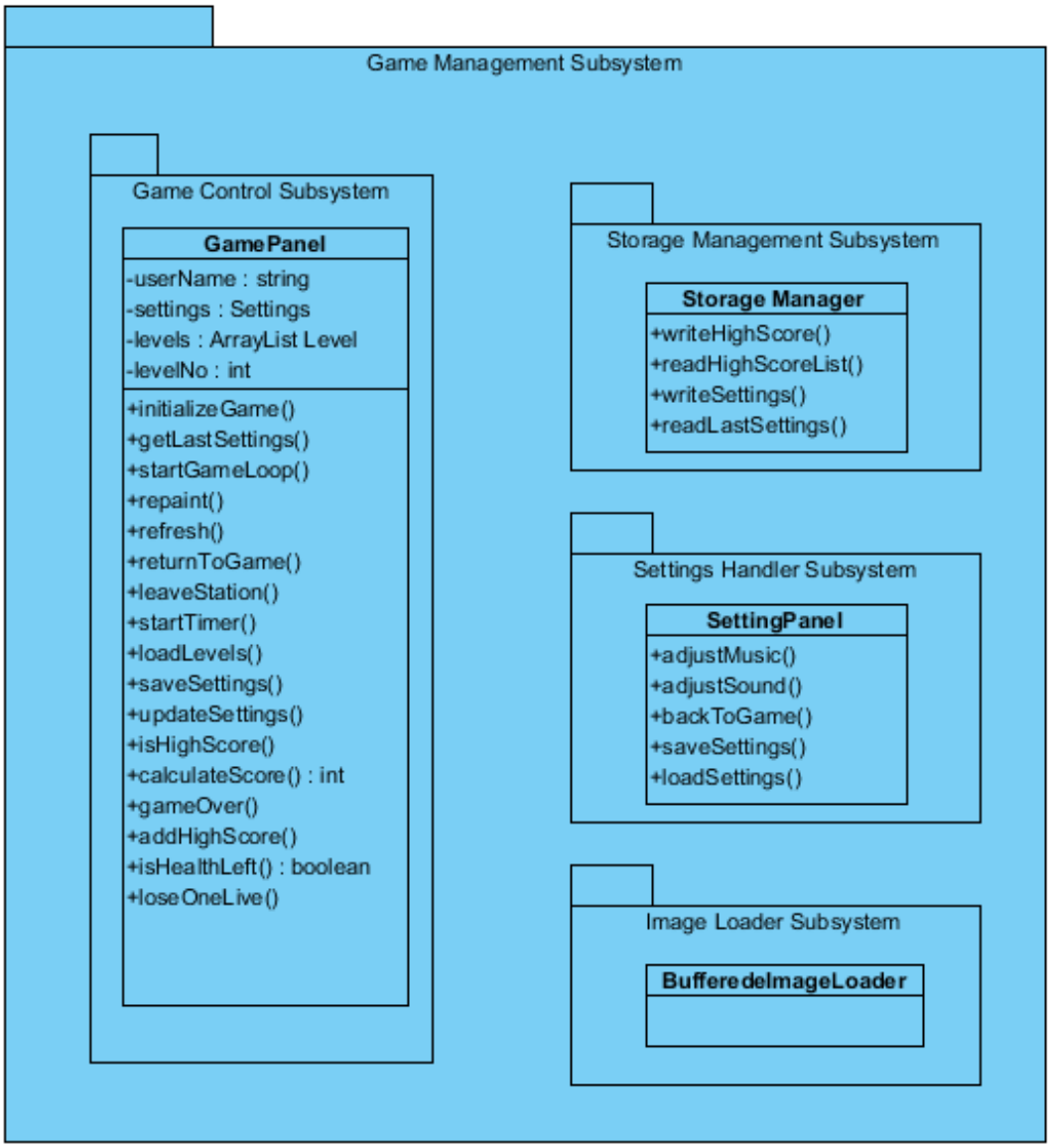



Figure 19: Package Diagram of Game Management Subsystem

**Game Control Subsystem**

Game Control Subsystem only contains the GamePanel class which handles and controls whole game streaming of the Armageddon. This class has an objective that arranges and assigns duties to other classes of the system. Therefore it could be called control class of the system. The package diagram of Game Control Subsystem could be found below:

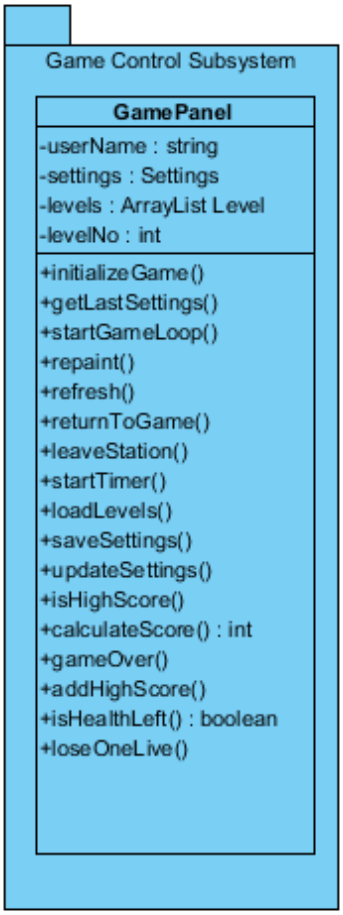



Figure 20: Package Diagram of Game Control Subsystem

**Settings Handler Subsystem**

Settings Handler Subsystem controls the sound and music volumes of the game. Other subsystems which will be use settings will interact with this subsystem. This subsystem only has the SettingPanel class. The package diagram of the Settings Handler Subsystem could be found below:

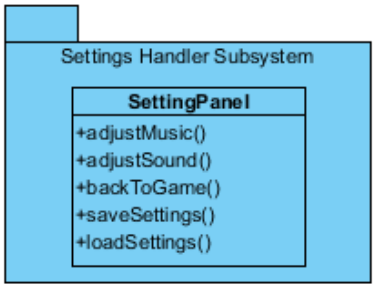



Figure 21: Package Diagram of Settings Handler Subsystem

**Image Loader Subsystem**

During the game images are consistently used in order to communicate with the user. Therefore, we define Image Loader subsystem with the helper class BufferedImageLoader to load images from particular files to the system as Java's Buffered Image. The package diagram of the Image Loader subsystem could be found below:

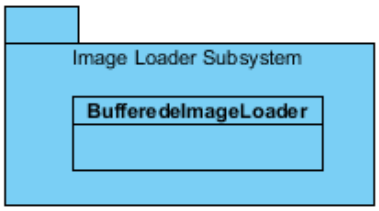



Figure 22: Package Diagram of Image Loader Subsystem

**Storage Management Subsystem**

Storage Management subsystem will be used to store persistent data which are list of high scores and settings as text files if any changes occur. We won't be using database in order to store the data. Because there is no need to construct a database for this kind of limited information; high scores are just ten lines, in each line there are names and corresponding scores and settings are stored as two integer value that are corresponding to volume of music and sound. More explanation could be found at the section Persistent Data Management. Storage Management subsystem has only the class named Storage Manager. The package diagram of this subsystem could be found below:

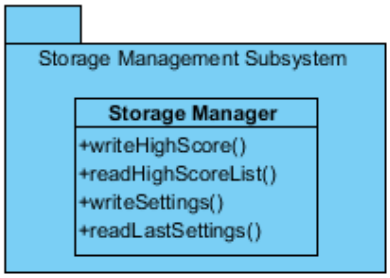



Figure 23: Package Diagram of Storage Management Subsystem

### 4.2.4. Model Subsystem

This subsystem consists of two subsystems. These are Drawable Subsystem and Level Subsystem. Model Subsystem represents the entity objects of the system and its components will be modified during system operations.

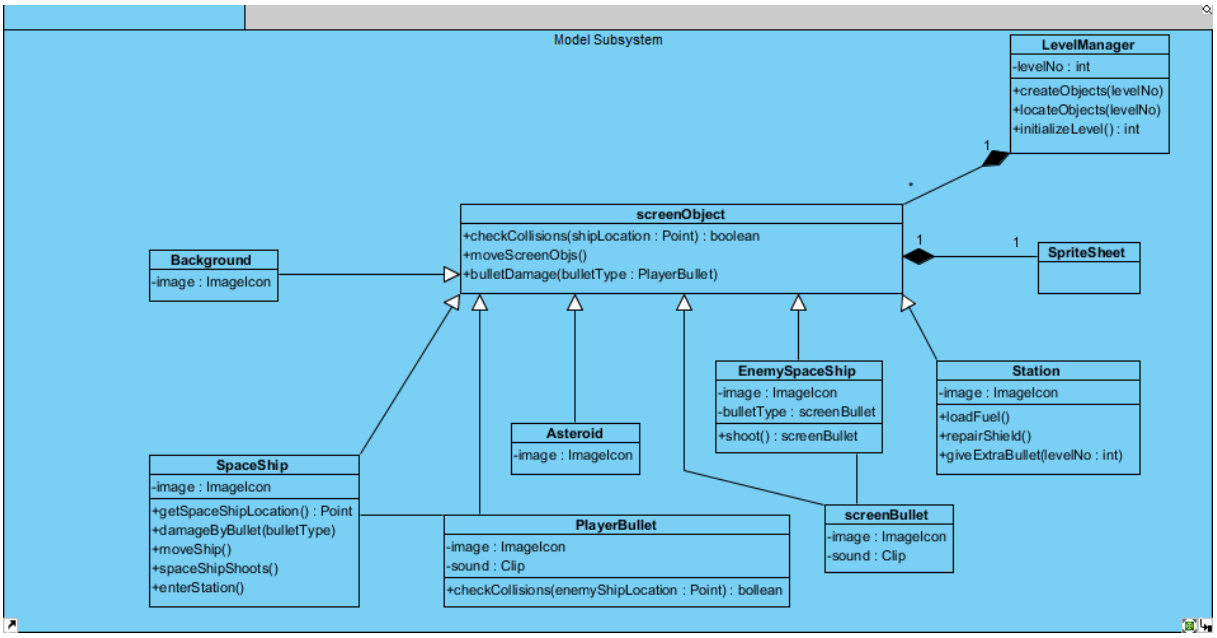



Figure 24: Package Diagram of Model Subsystem

**Drawable Subsystem**

This subsystem contains all the Drawable objects with their features. Interaction with the user and the system arrangements will be done by Drawable objects therefore they are crucial for the system. Drawable is responsible of representing the drawing content as well as the specification of the components. The package diagram of the Drawable subsystem could be found below.

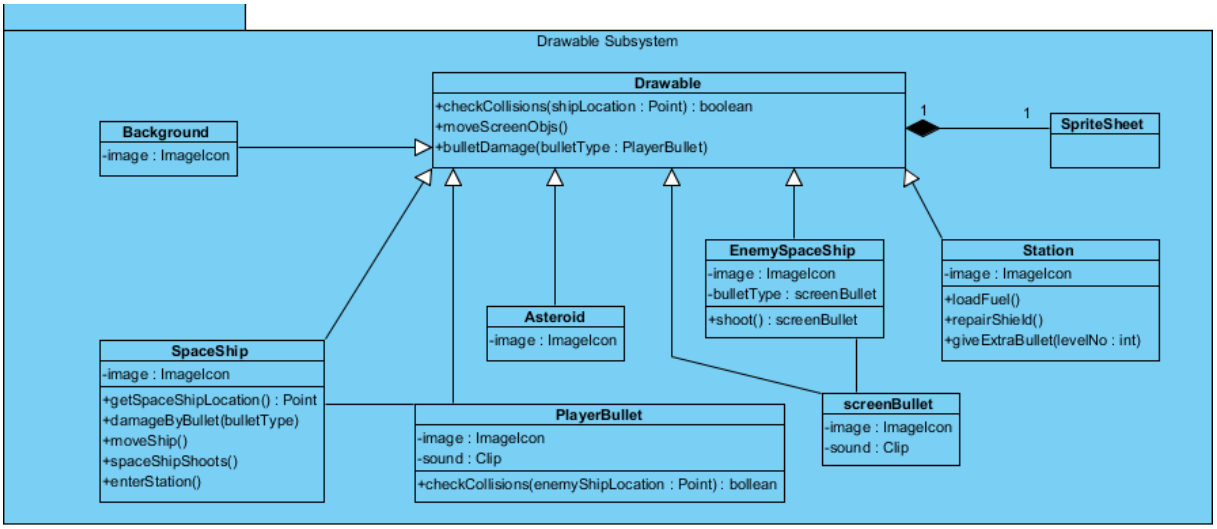



Figure 25: Package Diagram of Drawable Subsystem

**Level Management Subsystem**

This subsystem only has the Level class because it represents the organization of levels. Levels will be created first and stored in the files of the system. Stored levels will have the enemy spaceship count of the level, number of the level and the drawable objects like spaceships with their special point on the level. The package diagram of the Level Management Subsystem could be found below.

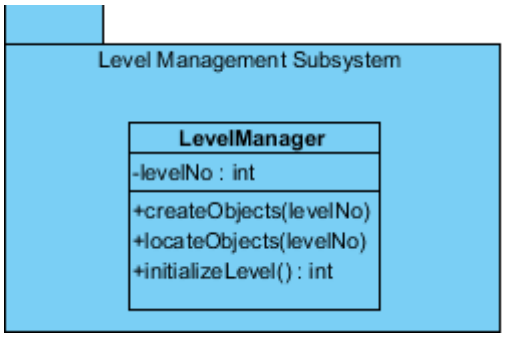



Figure 26: Package Diagram of Level Management Subsystem

## 4.3. Architectural Patterns

In our system design we use two different architectural patterns. Those are MVC (Model-View-Controller) and Layer patterns.

### 4.3.1. MVC (Model-View-Controller)

The user interacts with the controller and gives some inputs to the system. The controller subsystem modifies objects of the model class and changes their states. Those models are used by view and view subsystem outputs some graphical features to the user. In addition to this there are some controller systems that organize the main functions of the system. The aspects of the subsystems could be found in “Sub-System Decomposition” section of the report. The Model-View-Controller pattern of the system could be found below:

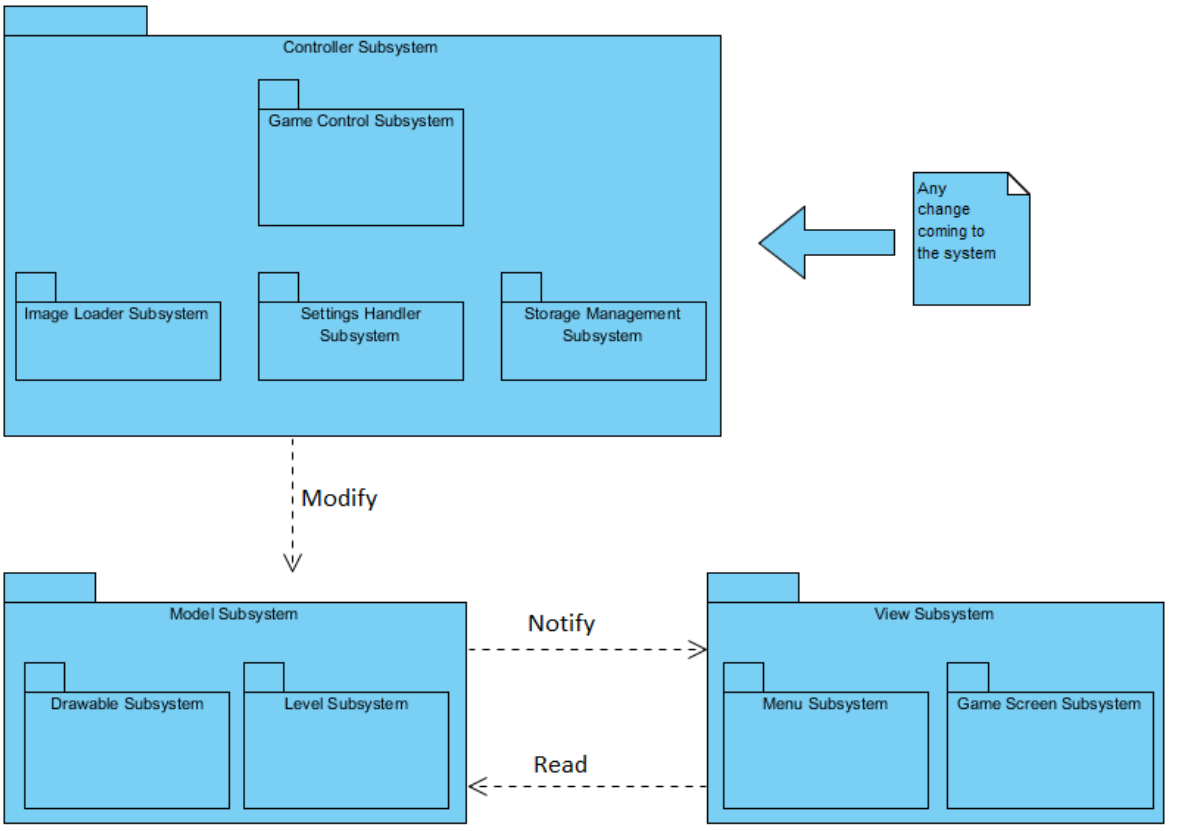



Figure 27 : Package Diagram of MVC Pattern

### 4.3.2. Layer Pattern

There is also layer pattern in our system design. Layer pattern presents the communication and the data flow between the packages. In the below diagram, “Menu Subsystem” and “Game Screen Subsystem” can interact with the “Game Control Subsystem” which is the main control class of the whole system. “Game Control Subsystem” can interact with the “Settings Handler Subsystem”, ”Storage Management Subsystem”, and “Image Loader Subsystem” which are explained at the section Controller Subsystem. Lastly “Game Control Subsystem” can use “Level Subsystem” and “Drawable Subsystem” which are drawn at the bottom of the below diagram. The Layer pattern could be seen below:

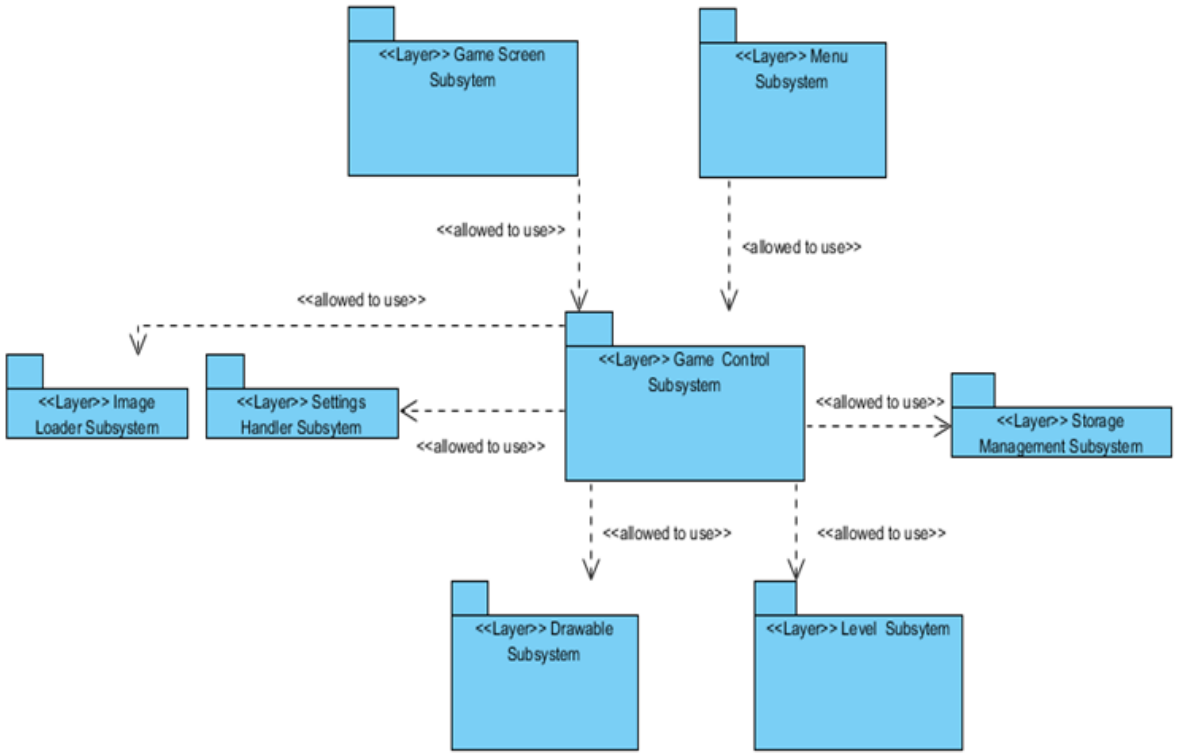



Figure 28: Package Diagram of Layer Pattern

## 4.4. Hardware/Software Mapping

Armageddon game will be played on the one computer with one user, therefore our game will be stand-alone system. Our deployment diagram for this game is below.

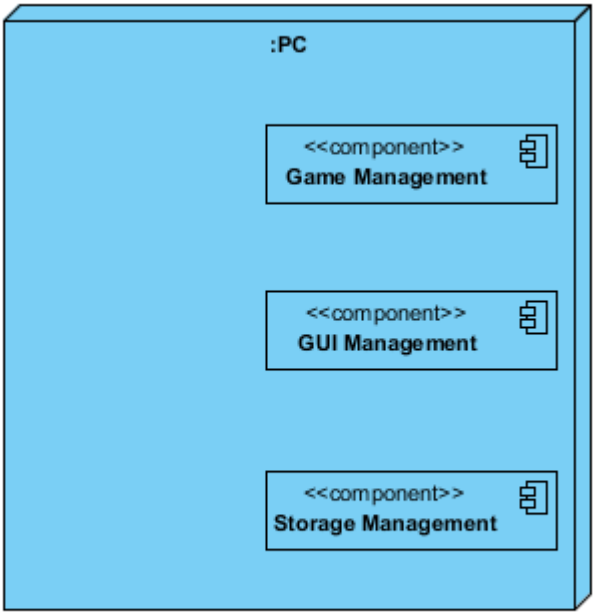



Figure 29 : Stand Alone Deployment Diagram

To implement this project we choose Java programming language. Java is recommended by course instructors. Also Java is well known by group members so this provides ease of implementation for our group. There are numerous benefits of using Java. First advantage is that Java has many useful libraries for this project. Other advantage is documentation. While project is implemented if developer faces a problem, he or she can find many resources for solution of this problem on online environment. Other benefit is Java has compiler which is very good at indicating errors when they occur. The other reason is to develop this project at any system such as Linux or Windows.

## 4.5. Addressing Key Concerns

### 4.5.1. Persistent Data Management

In the game, user can load saved game by using password that are given at the end of the game. These passwords, levels and score they reach will be stored in a plain text file for loading. However, user will not be able to reach or modify this plain text file. Other option user can do is viewing high scores list. These high scores also will be stored in plain text file. Users will not be able to modify this list also. In addition of these, game should store certain image and sound files in order to have good interface. Those files will be stored in specific directories and their locations will be fixed. All these storage files can be accessible by the system when the game runs.

### 4.5.2. Access Control and Security

There will be access control for different actors, classes and objects so that unauthorized data access and manipulation wil be prevented. The only actor in our case is the player who can play the game, change settings, look at the high scores and look at credits. Every user can change settings regardless of his/her identity, to change the high scores, the player has to play the game and get a highscore other than that the player can not access the passwords or the high score list. Other than this issue, there is not much to do about security and access control since the system is not using outside resources.

### 4.5.3. Global Software Control

In this game, user can control this game via specific keyboard inputs or mouse clicks. System takes these inputs and evaluates then if it suits with flow event of Armageddon, system calls necessary methods. These input calculations and system calls are made by Controller Subsystem. After that View Subsystem changes GUI objects according to decisions of Controller Subsystem.

Since different objects decide on the actions by evaluating different events in Armageddon game, the control will be decentralized. There is no centralized controller that decides the whole

event flow itself. We will implement a decentralized design due to the fact that dynamic behavior needs to be distributed among different objects.

### 4.5.4. Boundary Conditions

**Initialization**

- The user double-clicks the executable file of the game and then Main Menu shows up on the screen. There is no login page inside of the game, no profile exist within the game.
- On the Main Menu Panel there are five buttons to perform the following functionalities: “Play Game”, “Load Game”, “View High Scores”, “Credits” and “Exit Game”.
- When user clicks the “Play Game” button game is initialized from the first level and system is loading default settings.
- Alternative to “Play Game” user can clicks “Load Game” button to start from specific level with entering a password corresponding to that level. If password that user enters is not matching with any password registered in the system then no game is loaded.

**Termination**

- During the game play user can stop flow of the game anytime by pressing “Esc” button from the keyboard. From the appearing Pause Menu user can choose the “Return to Main Menu” option then game will be terminated.
- If player loses all his/her lives during the game play or finishes all the levels without getting a high score then game will be over.
- If player finishes all the levels with a score that is higher than the tenth high score then user name will be asked and score will be saved on the high score list. Then game will be over.

**Failure**

- If the user exits from the game without getting a password score will be dropped and it is not recorded to the system as well as game will be discarded. In other words user cannot continue to play that level later on.

### 4.5.5. System Installation and Maintainability

While developing our game we aim to provide users an easy install system such that even the users with no advanced experience should not have any trouble starting the game. To enable such a feature we desire our game to be executed by only clinking a .jar file with no usage of any other files. This way of implementation, however, may increase the size of .jar file in the disk. To solve this problem, we will create two .jar files. While one will have image, music and sound files, the other will be an optional file without the music files.

Additionally, we aim to ensure to maintain the system integrity. To be able to that, we plan to use SHA-256 (Secure Hash Algorithm) algorithm to generate a tag from system's immutable files. This tag will be embedded inside the code of the system. System will check the integrity when user decides to play the game. In order to that system will calculate the current tag and compare it with the one which is embedded. The System will not be executed and give a warning to the user if a disparity happens.

## 5. Object Design

In this section, we will describe the patterns we will use and the reasons behind the usage of those specific patterns. There is also a part where the design patterns are applied to the class diagram of this system. For this project we used Façade pattern, strategy pattern and observer pattern.

## 5.1. Design Patterns

### 5.1.1. Façade Pattern

MVC is applied to this system as a architectural pattern. Among those members communication is needed. Model part includes the information about the game, controller includes methods to handle inputs from the user, view includes the GUI and sound of the system. The view also contains menus, in this case it is nof efficient to change the model then the view. This implies that information transfer is needed between controller and view. MVC supplies the connection between those two elements however there will be a coupling problem. As a solution, façade pattern will be used. Only one instance of façade for each element will be used.(One for model, one for controller and one for view.)

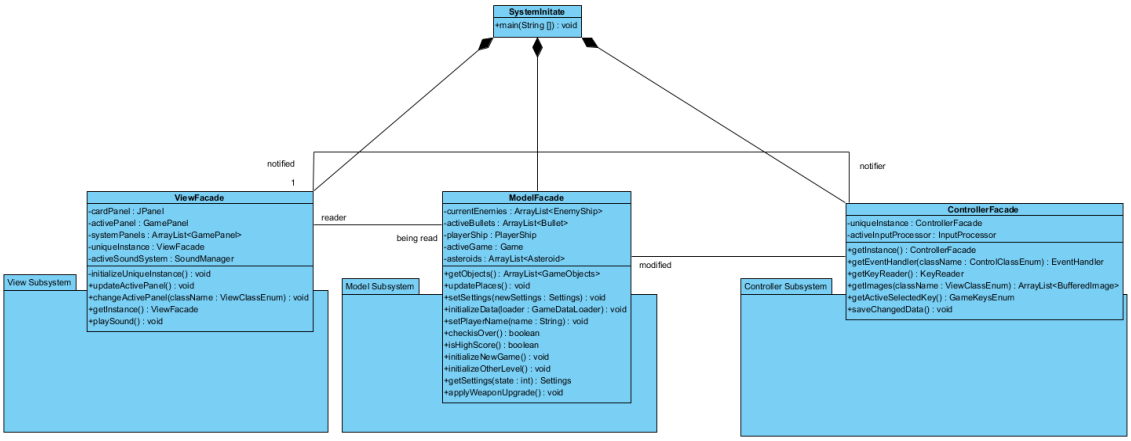



Figure 30: Façade pattern used for connection among MVC components

### 5.1.2. Strategy Pattern

After taking a look at the class diagram it can be seen that there are various types of classes that handle process in different way, such as the power-ups for the weapon of the ship; different

inputs change the model by assigning different values to the model which means different results are reflected to the view. Different panels will have different behaviour depending on which one is at the front by using the paint component and event handlers executing commands differently.

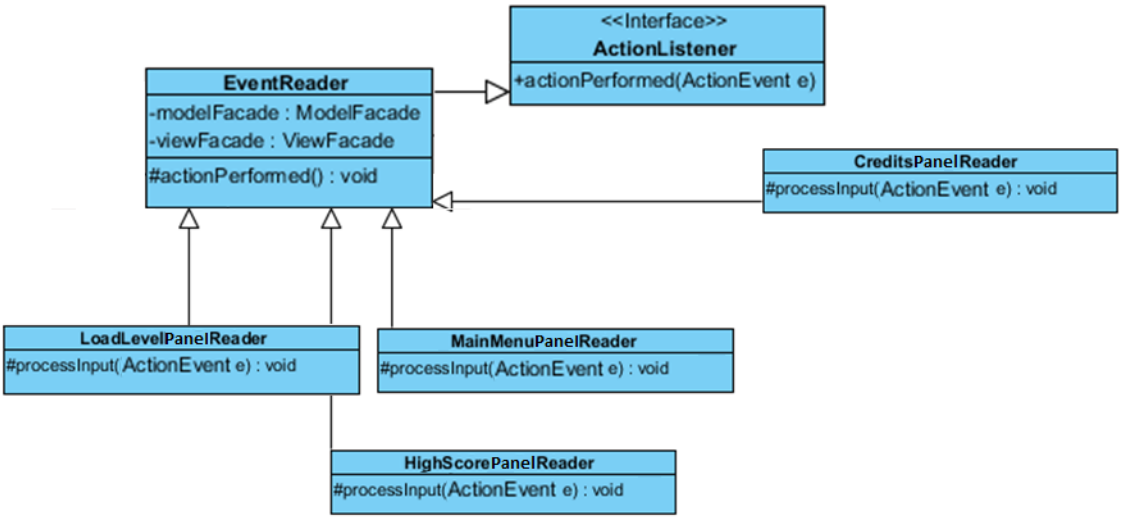



Figure 31: Strategy pattern used for EventReader class

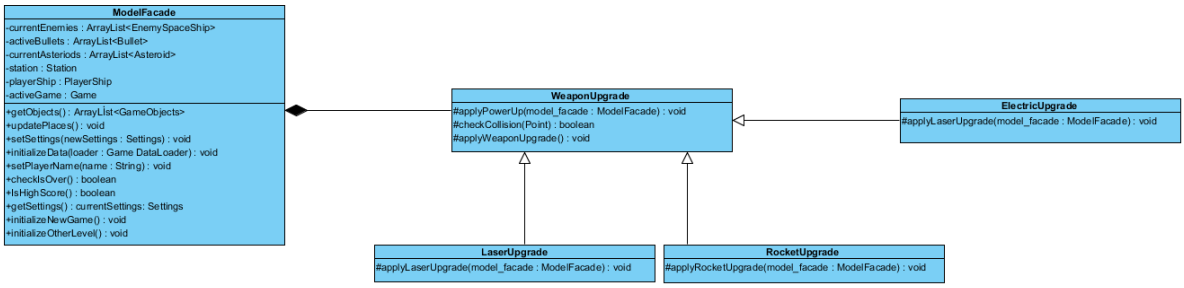



Figure 32: Strategy pattern used for WeaponUpgrade class

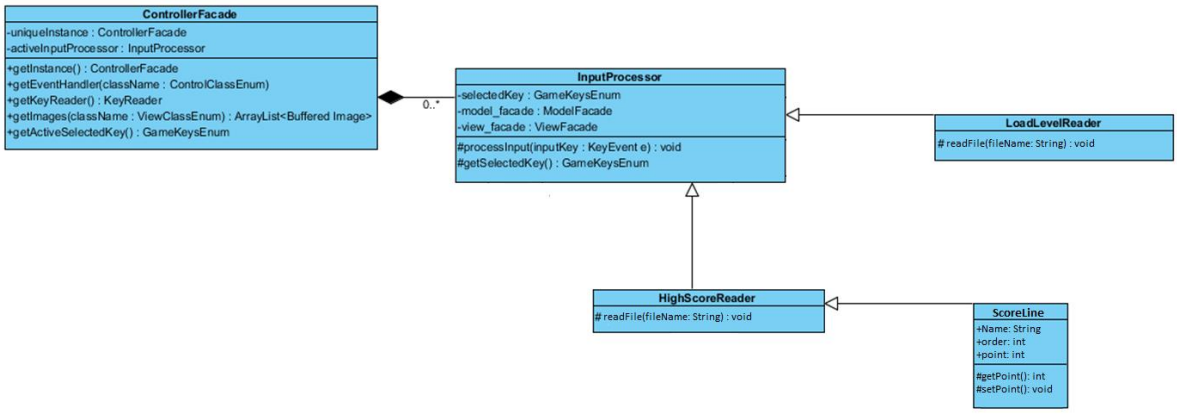



Figure 34: Strategy pattern used for InputProcessor class

### 5.1.3. Observer Pattern

In game, weapon upgrades are obtained randomly through the stations. The enemy spaceships' and players spaceship's locations and the weapon upgrades must be observed by certain classes to keep track of them. So observer pattern is the right option for this system. The observer pattern will be applied for the spaceships and the weapon upgrades.

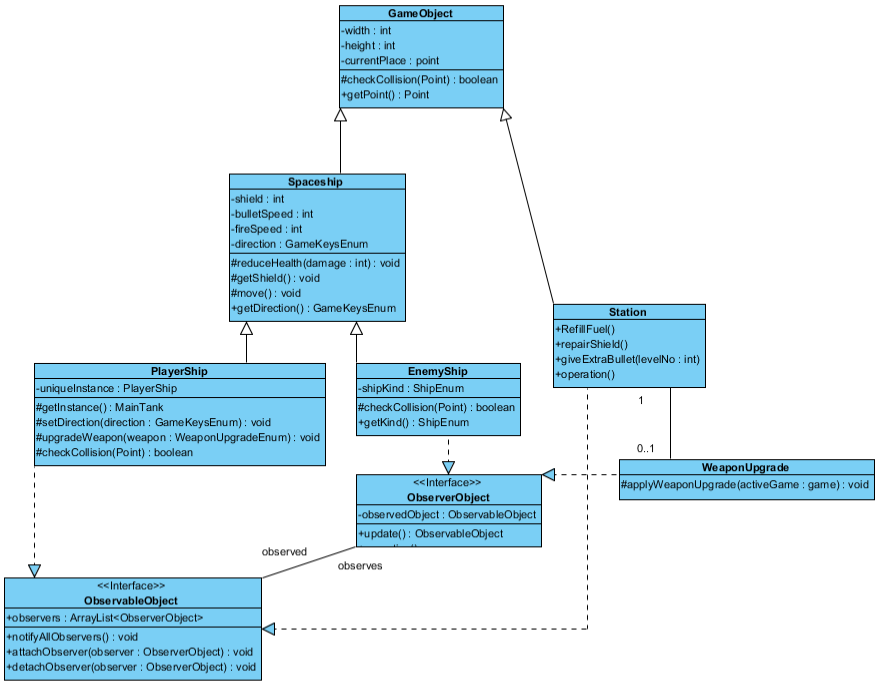



Figure 35: Observer pattern used for WeaponUpgrade, PlayerShip and EnemyShip classes

### 5.1.4. Pattern Applied Class Diagram

Logic behind the patterns have been explained. We will apply the patterns to the class diagram and make a new one which will serve as the final version.

Figure 36: Pattern applied class diagram

### 5.2.1 Model Classes

**ModelFacade.java:**

ModelFacade is responsible of making the changes due to the modifications on the model. Its attributes are the currentEnemies which stores the coordinates of the enemy spaceships in an ArrayList, activeBullet keeps an ArrayList of the coordinates of the bullets as well and the asteroids attribute stores the locations of all the asteroids in the map. The other attributes are playerGame and activeShip. Addition to these attributes, ModelFacade class has the methods such as getObjects(), updatePlaces(), setSettings(newSettings : Settings), initializeData(loader : GameDataLoader), setPlayerName(name : String), checkisOver(), isHighScore(), initializeNewGame(), initializeOtherLevel(), getSettings(state : int), applyWeaponUpgrade().

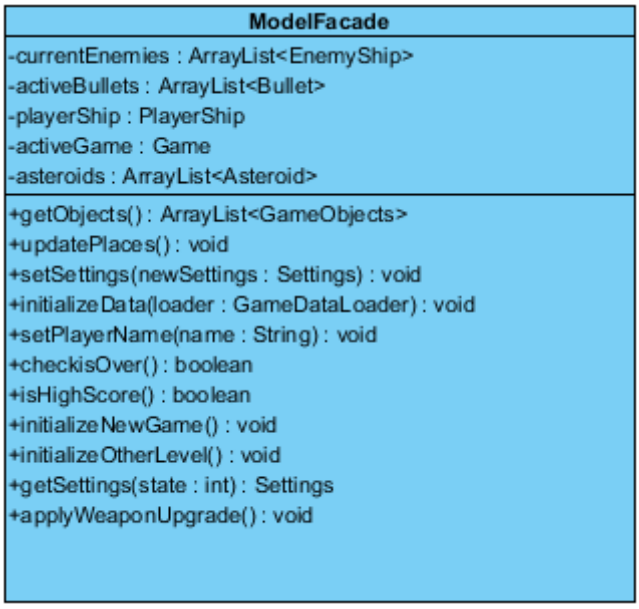



Figure 37: ModelFacade class interface

**Game.java**

ModelFacade class consists of Game class which has the attributes of levels, currentSettings, defaultSettings, currentLevel and score. In addition the Game class has the protected methods such

as setSettings(newSettings : Settings), getdefaultSettings(), getLevel(), setPlayerName(name : String), isHighScore(), getScore(), getPlayerName(), levelPassed(), initializeNewGame().

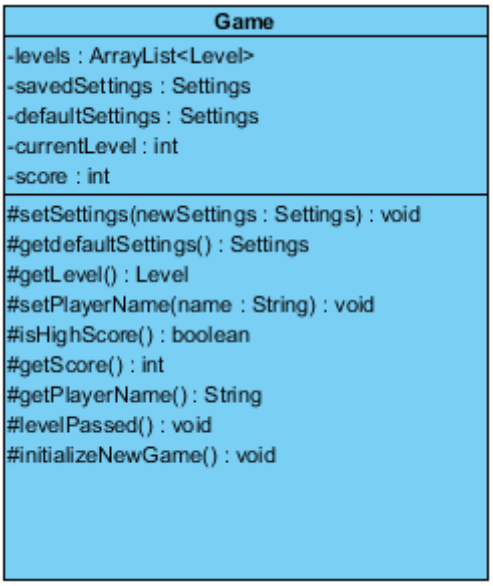



Figure 38: Game class interface

**Settings.java**

Game class consists of Setting calss which has the properties isMusicOn and IsSfxOn. It also provides protected methods setMusic(changeMusic), getIsMusicOn(), setSfx(), getIsSfxOn().

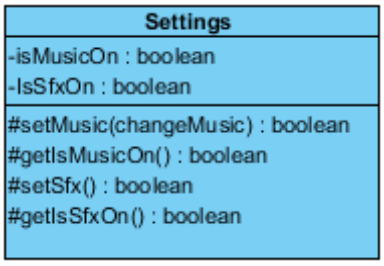



Figure 39: Settings class interface

**Level.java**

**Implements :** Serializable

Level.java class includes the feature of all levels. Its attributes are levelNumber, shipStart, enemyList, asteroidList and it has protected methods of getEnemies(), getAsteroids(), getPlayerLocation().

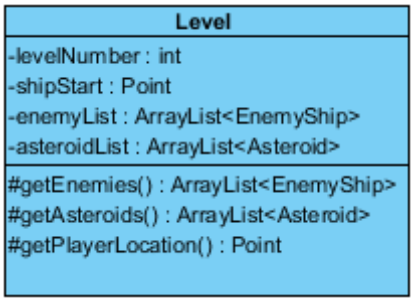



Figure 40: Level class interface

**GameObject.java**

GameObject class responsible of representing the objects in the game such as bullets, spaceships, stations etc. It has the attributes of width, height, currentPlace. In addition it has the protected method checkCollision(Point) which checks if collision occurred between objects and the public method getPoint() which returns the location of the object.

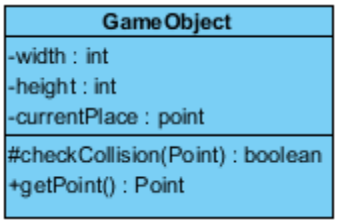



Figure 41: Level class interface

**Spaceship.java**

**Extends:** GameObject.java

Spaceship is the child class of GameObject class and models the properties of Spaceship object. It has attributes such as shield, bulletSpeed, direction. It also has the protected methods reduceHealth(damage : int), getShield(), move() and the public method getDirection(). The getDirection() method is public because inorder to draw the object the View class should be able to access its locaton.

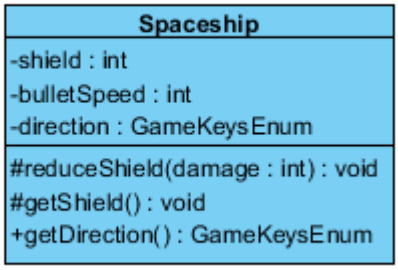



Figure 42: Spaceship class interface

**Bullet.java**

**Extends:** GameObject.java

Bullet is the child class of GameObject and models the properties of Bullet object. It has attributes such as isCollide, damage. It also has protected method BulletCollide() and the public method getIsCollide() because we need View class to have an access to it.

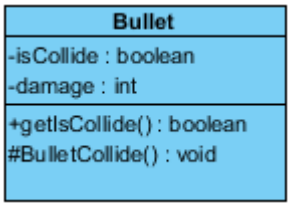


Figure 43: Bullet class interface

**PlayerShip.java**

**Extends:** Spaceship.java

**Implements:** ObservableObject.java

PlayerShip is a child class of Spaceship class and is responsible of representing the spaceship that the user is controlling. It has the properties uniqueInstance, weaponType and fuel and the protected methods getInstance(), setDirection(direction : GameKeysEnum), upgradeWeapon(weapon : WeaponUpgradeEnum), checkCollision(Point). This class also implements the ObservableObject class since it is observable by enemyships.

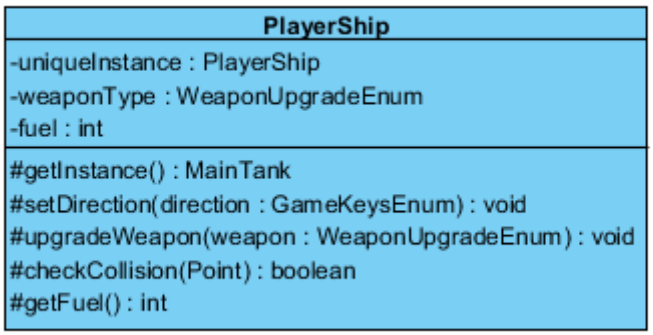



Figure 44: PlayerShip class interface

**EnemyShip.java**

**Extends:** Spaceship.java

**Implements:** Serializable.java, ObserverObject.java

EnemyShip is a child class of Spaceship class and is responsible of representing the enemy spaceships in the game. Its attribute is shipKind and it has the protected method checkCollision(Point) and the public method getKind(). It also implements ObserverObject class since it shoul observe the player's object.

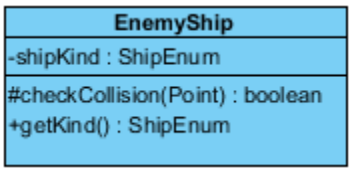



Figure 45: EnemyShip class interface

**Station.java**

**Extends:** GameObject.java

**Implements:** ObservableObject.java

Station class is a child class of the GameObject class and it is responsible of representing the non-moving object on the screen. It has the private property ship and public methods RefillFuel(), repairShield(), giveExtraBullet(levelNo : int).

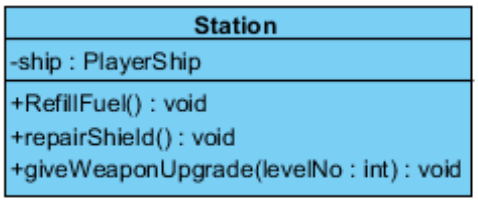



Figure 46: Station class interface

**WeaponUpgrade.java**

**Extends:** GameObject.java

**Implements:** ObserverObject.java

WeaponUpgrade class is a child class of the GameObject class and it is responsible of representing the three different types of upgrade of weapons in the game. It has the protected method applyWeaponUpgrade(activeGame : game). It applies the upgrade to the user's spaceship.


Figure 47: WeaponUpgrade class interface

**LaserUpgrade.java**

**Extends:** WeaponUpgrade.java

LaserUpgrade class is a child class of the WeaponUpgrade class and it is responsible of representing weapon upgrade which enables the player's spaceship to have laser power. It has the protected method applyLaserUpgrade(model_facade : ModelFacade).


Figure 48: LaserUpgrade class interface

**ElectricUpgrade.java**

**Extends:** WeaponUpgrade.java

ElectricUpgrade class is a child class of the WeaponUpgrade class and it is responsible of representing weapon upgrade which enables the player's spaceship to have electric power. It has the protected method applyElectricUpgrade(model_facade : ModelFacade).

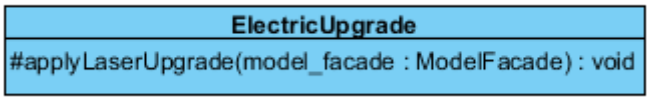



Figure 49: ElectricUpgrade class interface

**RocketUpgrade.java**

**Extends:** WeaponUpgrade.java

ElectricUpgrade class is a child class of the WeaponUpgrade class and it is responsible of representing weapon upgrade which enables the player's spaceship to have rocket power. It has the protected method applyRocketUpgrade(model_facade : ModelFacade).

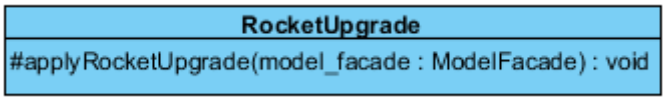



Figure 50: RocketUpgrade class interface

**BackGround.java**

**Extends:** Gameobject.java

Background class is a child class of the GameObject class and it is responsible of representing the background apperance of the game. It has attribute farIcon.


Figure 51: Background class interface

**CloseStar.java**

**Extends:** Gameobject.java

CloseStar class is a child class of the GameObject class and it is responsible of representing the stars with the close apperance. It has the public method move().

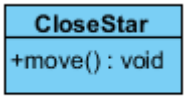



Figure 52: RocketUpgrade class interface

**FarStar.java**

**Extends:** Gameobject.java

FarStar class is a child class of the GameObject class and it is responsible of representing the stars with the further apperance. It has the public method move().


Figure 53: FarStar class interface

**Asteroid.java**

**Implements:** Serializable.java

Asteroid class represents the moving obstacles in the game. It has the attribute health and the protected method checkCollision().

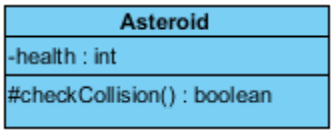



Figure 54: Asteroid class interface

**ViewFacade.java**

ViewFacade is responsible for making communication between panels in View classes. This class has cardPanel, activePanel, systemPanels, uniqueInstance, activeSoundSystem attributes. Methods that ViewFacade has, are initializeUniqueInstance(), updateActivePanel(), changeActivePanel(className: ViewClassEnum), getInstance(), playSound().

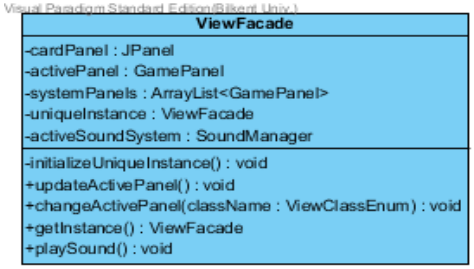



Figure 55: ViewFacade class interface

**MainMenuPanel.java**

**Extends:** GamePanel.java

MainMenuPanel is the panel that responsible for choosing a panel from a menu. This class has paintComponent() method as a protected method. User will be able to choose a panel from menu screen.

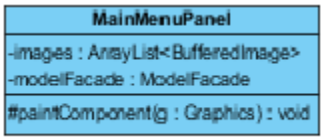



Figure 56: MainMenuPanel class interface

**NextLevelPanel.java**

**Extends:** GamePanel.java

NextLevelPanel is the panel that responsible for to call next level game. This class has paintComponent() method as a protected method.

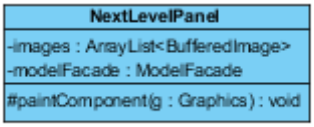



Figure 57: NextLevelPanel class interface

**GameScreenPanel.java**

**Extends:** GamePanel.java

GameScreenPanel is the panel that responsible for providing user to play game. This class has paintComponent() method as a protected method.

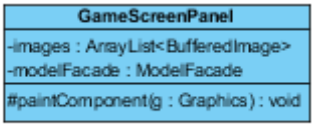



Figure 58: GameScreenPanel class interface

**LoadLevelPanel.java**

**Extends:** GamePanel.java

LoadLevelPanel is the panel that responsible for providing user to choose a game s/he saved before. This class has paintComponent() method as a protected method.

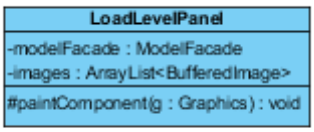



Figure 59: LoadLevelPanel class interface

**CreditsPanel.java**

**Extends:** GamePanel.java

CreditsPanel is the panel that responsible to show information about game to user. This class has paintComponent() method as a protected method.

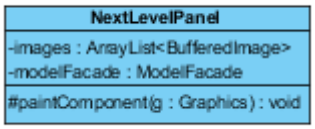



Figure 60: NextLevelPanel class interface

**HighScorePanel.java**

**Extends:** GamePanel.java

HighScorePanel is the panel that responsible to show highest ten score that user made before. This class has paintComponent() method as a protected method.

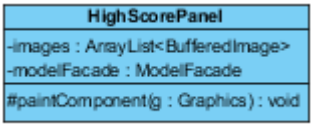



Figure 61: HighScorePanel class interface

**PauseMenuPanel.java**

**Extends:** GamePanel.java

PauseMenuPanel is the panel that responsible to show setting options when user pause the game. This class has paintComponent() method as a protected method.

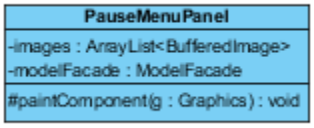



Figure 62: PauseMenuPanel class interface

**GameOverPanel.java**

**Extends:** GamePanel.java

GameOverPanel is the panel that responsible to make user be able to save his/her score if it is in top ten and to give a password to be able to play game in current level. This class has paintComponent() method as a protected method.

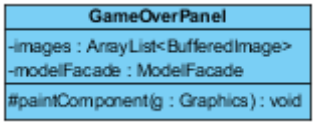



Figure 63: GameOverPanel class interface

**SoundManager.java**

SoundManager is the panel that responsible to play music and sound effects while application is running. This class has music and effectSound attributes. Methods that SoundManager has are playMusic(), playEffectSound().

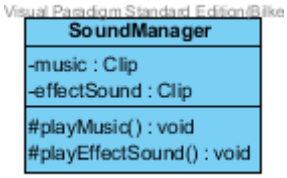



Figure 64: SoundManager class interface

### 5.2.3 Controller Classes

**ControllerFacade.java:**

Our design has Facade Pattern so that we have ControllerFacade class which provides the other classes with the ability to communicate with Controller classes. Variables of the ControllerFacade are activeInputProcessor and uniqueInstance. And methods of the ControllerFacede class are getInstance(), getEventHandler(className : ControlClassEnum), getImages(className :ViewClassEnum), getKeyReader(), saveChangedData(), getActiveSelectedKey(), processInput() which are defined as public methods to enable other classes to call these methods.

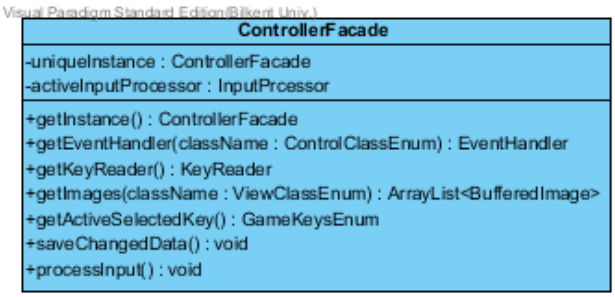



Figure 65: ControllerFacade class interface

**EventReader.java:**

**Implements:** ActionListener.java

Our design has EventReader class which reads timing and transactions in the game. Properties of this class are model_facade and view_facade as a Timer object and provides a method actionPerformed () which is indicated as protected.

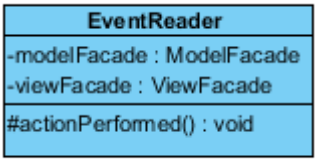



Figure 66: EventHandler class interface

**MainMenuPanelReader.java:**

**Extends:** InputProcessor.java

Our design has MainMenPanelReader class that processes the directives of the user while user is on the MenuPanel.


*Figure 67 : MainMenuPanelReader class interface*

**CreditsPanelReader.java:**

**Extends:** InputProcessor.java

Our design has CreditsPanelReader class which processes the directives of the user while user is on the CreditsPanel. User can only go back to the MenuPanel.


*Figure 68:* CreditsPanelReader *class interface*

**HighScorePanelReader.java:**

**Extends:** InputProcessor.java

Our design has HighScorePanelReader class that processes the directives of the user while user is displaying the HighScorePanel. User can only go back to the MenuPanel.


*Figure* – HighScorePanelReader *class interface*

**LoadLevelPanelReader.java:**

**Extends:** InputProcessor.java

Our design has LoadLevelPanelReader class which processes the directives of the user while user is displaying the PasswordPanel which is for loading a saved game. User can either go back to the MenuPanel or submit his/her password for a saved game.


*Figure 69:* LoadLevelPanelReader *class interface*

**KeyReader.java**

**Implements:** KeyListener.java

Our design has KeyReader class which reads the inputs. It has an input and an activeKeyEvent variable. KeyReader class provides one method, it is getKeyEvent() which is indicated as protected.

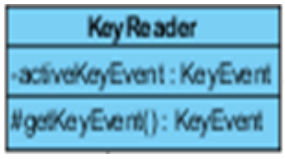



Figure 70 : KeyReader class interface

**InputProcessor.java**

Our design has InputProcessor class which processes the input according to the current panel and data from screen. InputProcessor has three variables which are model_facade controller_facade and scoresReader.

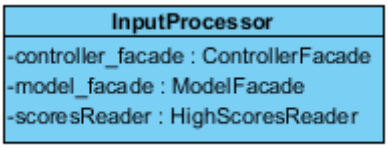



Figure 71: InputProcessor class interface

**HighScoreReader.java:**

**Extends:** InputProcessor.java

Our design has HighScoreReader class which reads the "highscore.txt" file to show top ten high scores to the user while user is displaying the HighScorePanel.

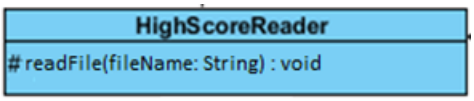



Figure 72: HighScoreReader class interface

**ScoreLine.java:**

Our design has ScoreLine class which is object of each line that is reading from "highscore.txt" file by HighScoreReader.java.

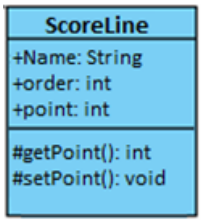



*Figure 73:* ScoreLine *class interface*

**LoadLevelReader.java:**

**Extends:** InputProcessor.java

Our design has LoadLevelReader class which reads the “loadlevel.txt” file to catch what user submits while user is displaying the PasswordPanel. User can submit his/her password and start from saved level.


Figure 74: LoadLevelReader class interface

**BufferedImageLoader.java**

Our design has BufferedImageLoader class that reads the all images from the image file. BufferedImageLoader has the private variables which are uniqueInstance, creditsImages, loadLevelImages, nextLevelImages, highScoreImages, pauseMenuImages, gameScreenImages, gameOverImages and it provides protected methods, these methods are loadImages() and getImages().

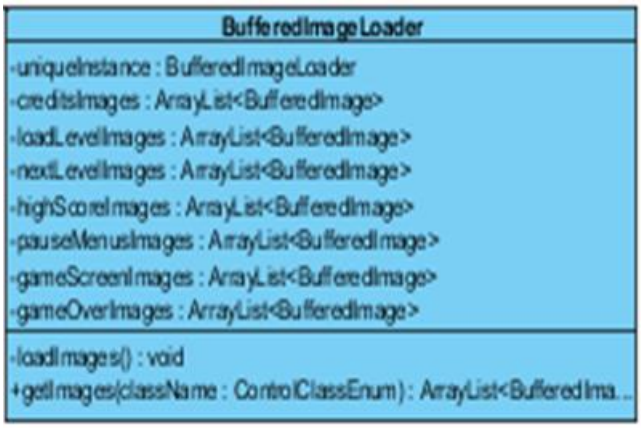



Figure 75: BufferedImageLoader class interface

**GameDataLoader.java**

Our design has GameDataLoader class which holds the necessary variables of the game. GameDataLoader has four variables, these are; levels, defaultSettings, savedSetting and modelFacade. And GameDataLoader class provides a method loadGameData() which is indicated as private.


Figure 76: GameDataLoader class interface

### 5.2.5 Interface Classes:

**ActionListener.java:**

**Extends:** EventHandler.java

Our design has ActionListener interface that processes the actions of the user while user is playing the game. This interface has operation() and actionPerformed(ActionEvent e) methods which are public.


Figure 77: ActionListener interface

**KeyListener.java:**

Our design has KeyListener.java interface which is used to detect key inputs when user presses any button from keyboard. KeyListener is implemented by the KeyReader class from controller part of the design. This interface has three methods; KeyTyped(e: KeyEvent), KeyReleased(e: KeyEvent) and KeyPressed(e: KeyEvent) which are indicated as public.

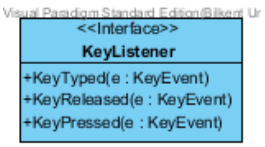



Figure 78: KeyListener interface

**ObservableObject.java:**

ObservableObject is other interface of the design which fits to Observer Pattern. It provides the ability of being observable to an object. PlayerShip class implements this interface. This interface holds observer objects in an ArrayList and provides three methods, these are notifyAllObservers() which is public, attachObserver() and detachObserver() which are indicated as protected.

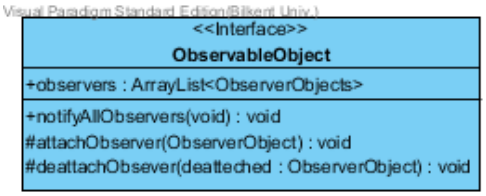



Figure 79: ObservableObject interface

**ObserverObject.java:**

There is also ObserverObject interface in the system which is part of the Observer Pattern. This interface provides the ability to observe an object. EnemyShip class implements this interface. It has observedObject variable which is object of ObservableObject. ObserverObject interface provides a method named update(changedObject : ObservableObject) that is indicated as a public. This method updates the location of the user's spaceship, PlayerShip, through the gameplay.

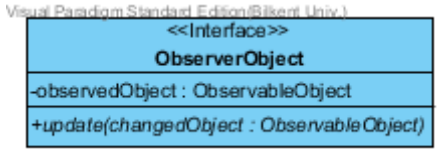



Figure 80: ObserverObject interface

**Serializable.java**

Apart from above interfaces, our design has also an interface which allows saving the classes to a txt file called Serializable.

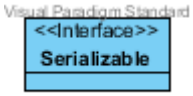



Figure 81: Serializable interface

## 5.3 Specifying Contracts using OCL

1. context Bullet inv: damage = 2
2. context Bullet::bulletCollide()  pre: isCollide==false,  post: isCollide = true;
3. context Bullet::getIsCollide():boolean  post:result = isCollide
4. context GameObject inv: width <= 30
5. context GameObject inv: height <= 30
6. context Settings::setMusicOn(set: boolean)   post: isMusicOn = set
7. context Settings::setSfxOn(set:boolean) post: isSfxOn = set
8. context Game::getScore() post: result = score
9. context Settings::getIsMusicOn():boolean post:result = isMusicOn
10. context Settings::getIsSfxOn():boolean post:result = isSfxOn
11. context Game::setPlayerName(name:string)  post: playerName = name
12. context Game::getPlayerName() post:result = playerName
13. context Game::getScore():int post:result = score
14. context Game::updateLevelScore(shipKind:ShipEnum) post: playerScore=playerScore+ ShipKind
15. context PlayerShip::setDirection(dir:GameKeysEnum) post:direction = dir
16. context PlayerShip::getInstance():  post:result  = self
17. context PlayerShip::upgradeWeapon(weapon : WeaponUpgradeEnum) post: weaponType = weapon
18. context Ship::reduceShield(damage:int) post: shield = @pre.shield - damage
19. context Ship::getShield():int  post: result = shield
20. context Ship::getDirection():GameKeysEnum  post: result = direction
21. context Station::repairShield () post:ship.shield = maxShield
22. context RocketUpgrade::applyPowerUp(model:ModelFacade)

    post: model.playership.weaponType = LASER

23. context Level::createPowerUp() pre: point = NULL
24. context EnemyShip::getKind():TankEnum post:result = tankKind
25. context Asteroid:: getHealth():int post:result = health
26. context Level::getPlayerLocation():Point result = shipStart
27. context Game::setSettings(newSettings:Settings) post: savedSettings = newSettings
28. context Game::getDefaultSettings():Settings post:result = defaultSettings
29. context Game::getLevel():Level post: result = currentLevel
30. context Game::getSettings():Settings post:result = savedSettings

## 6. Conclusion

**a) Summary**

Armageddon is a game system which is created with the aim of entertaining people. It challenges the users by enabling them to reach higher levels as they gain more score. It also keeps a highscore table and by this approach it creates a bigger challenge for the users as well. Armeggeddon will be a game to amusement for people and will help them to get over with the stress of the daily life.

The process of creating this game has three stages. First one of them was the analysis stage. Through this process, we defined our problem and described the system's requirements, both functional and non-functional. In addition to the requirments we created use case diagrams, analysis class diagram and dynamic diagrams such as sequence and activity diagrams.

Second part of the process was the design stage. Through this stage, we tried to explain our project and system goals. We divided the system into subsystems and decided which architectural patterns to apply to our system. We decided for our project the MVC and the layer patterns will be the most suitable. After that we created our software-hardware architecture map adressed what design goals we will focus on.

Last stage was the object design and in this stage we decided on applying three different patterns to the system with the aim of improving the implementation process. These patterns are façade, strategy and observer. After that process, we created the class interfaces and used OCL for specifying the contracts.

**b) Lessons Learned**

We learned many things from the different phases of our project. In th analysis process, we learned how describe a problem and how to find solutions to it with the help of the functional

requirements. We learn how to form use-cases and how to create class diagram and dynamic diagrams using the use-case diagrams.

While writing the design report, we saw how useful our analysis report was. It helped us a lot with the details of the design report. First, we started with deciding with which pattern we will use. Our decision was on the MVC pattern and layer pattern. After that, by using the Visual Paradigm we draw our package diagrams regarded to our class diagram. We also used Visual Paradigm to form our Use Case diagrams, Sequence Diagrams, Domain Analysis Diagram and State and Activity Diagrams.

The process of writing our design report teach us lots of things. One of them was how we should always think about our design critically and how we should change its feature according to this critical thinking. Another one was decision making of which pattern we should use with a project that has GUI.

After getting done with the design report, we started our object design. In this stage, we tried to choose the most suitable patterns for our system. Integrating these patterns into our system made it more appropriate for the implemention. According to the patterns we chose we larned ow to form our class interfaces. In additon, we learned how to use UCL to specify the contracts.

**c) Obstacles**

In the analysis procees, we have faced with many obstacles since we did not know how to decide on the functional, non-funcional requirements or how to form use cases. Firstly, we tried to get over these problems. After that we had struggles on how t draw use-case, class and dynamic diagrams. We get over these obstacles by learning the concepts.

In the design stage, the first obstacle we had to deal with was the decision of which pattern we will use. After we made our mind on which one to use, we then had to learn how to implement them in our project. We divided the project between the group members, we sometimes had some

misunderstandings with the part we are responsible of. However we gathered later and tried to clarified the misunderstandings.

The obstacle we had to get over in the object design was the decision making process of which patterns we should use. Again by learning the patterns accurately we dealt with this problem and selected the most appopriate patterns for our system.

**d) Future Work**

We tried to design Armageddon in such a way that it will be very appropriate for future improvement . As an future work, making the game as an online game may make the users enjoy the game much more. Moreover, adding 2 player option would be a remarkable future work.

## 7. References


- Object-Oriented Software Engineering, Using UML, Patterns, and Java, 2nd Edition, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2004, ISBN: 0-13-047110-0.