CS 319 - Object-Oriented Software Engineering

System Design Report

Iteration 2

Section 02 Group 2c

Space Invaders

Project Group Members

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

Supervisor: Uğur Doğrusöz

Contents

1. Introduction

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

1.1. Purpose of the System

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

1.2. Design Goals

- **Visual Design**

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

- **Gameplay**

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

- **Understandability**

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

- **Cogency**

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

- **Performance**

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

- **Difficulty**

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

- **Satisfaction-Joy**

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

1.2.1 Trade-Offs

Understandability vs Functionality:

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

Memory vs Maintainability:

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

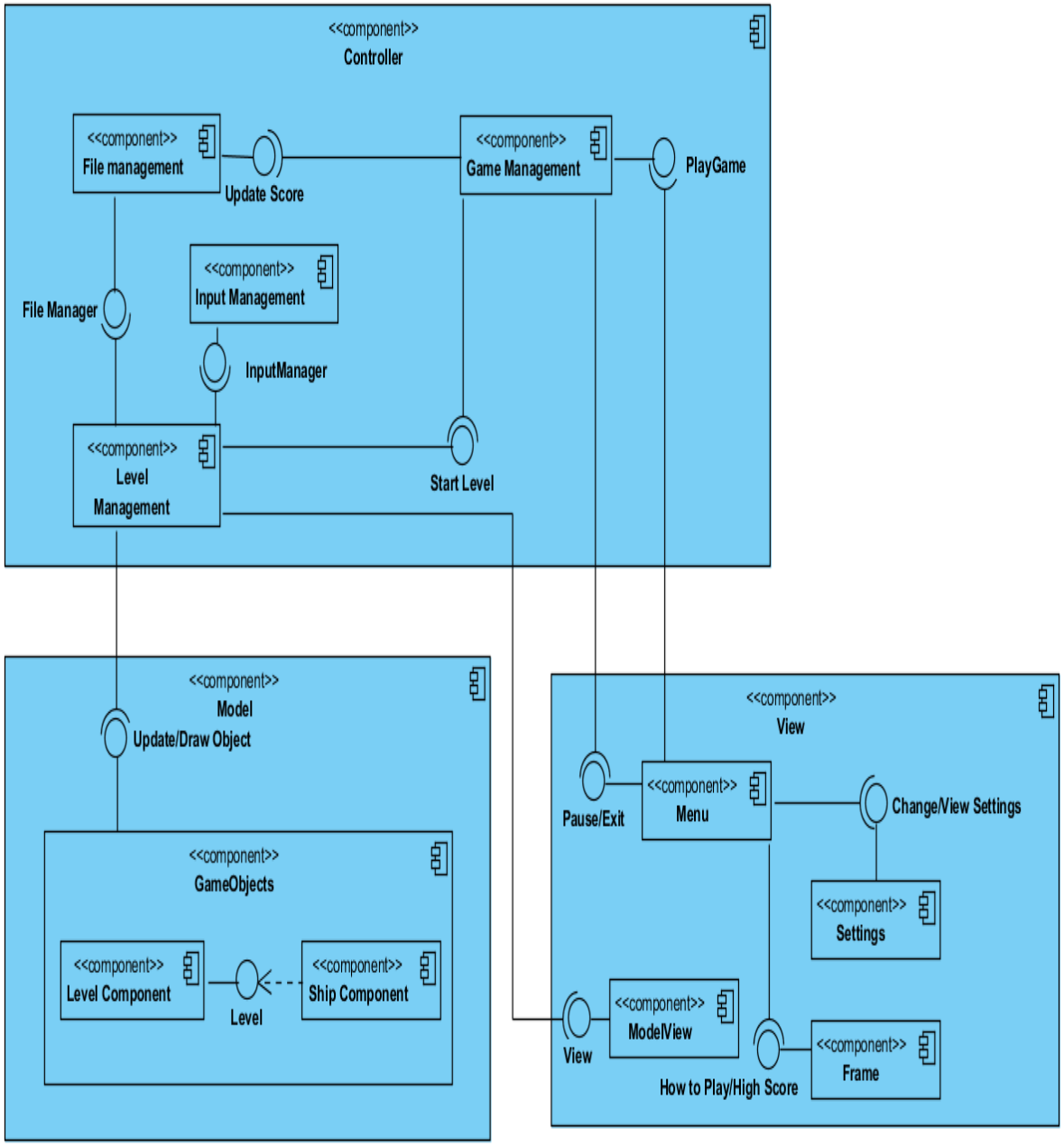
Development Time vs User Experience:

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

development. We might consider using JavaFX for user interface if we have enough tie for implementation period.

2. System Architecture

2.1. Subsystem Decomposition



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

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

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

- Player
- Enemies
- Bullet
- Enemy Bullet

• Barriers

• Menu

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

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

2.2. Hardware/Software Mapping

Space Invaders software will implement on PC and programmed in Java. The object of the program is to handle all necessary game logic, to make the game function as planned. A model of the game, programmed in Java, was used to evaluate the needed functionality. From the model it was derived that functions to initialize the hardware were needed as well as one main game loop. From the loop all the different functions handling the game logic itself were called upon. Functions such as moving and spawning sprites, collision detection and polling keyboard events. The structure of the software was designed to be single

threaded. The program is running with a main loop to handle the game logic functions. In addition to this it reads the input from the keyboard after receiving an interrupt. The received information from the keyboard only describes if a key has been pressed or released. Therefore, extra functionality was added to the function handling the interrupt from the keyboard. The player should be able to move and shoot simultaneously

2.3. Persistent Data Management

Space Invaders keeps its data in an ordinary file. This data includes game instances, objects, pictures or music. So, for Space Invaders game we will not use database or any specific data storage. Additionally, other important data members like user setting preferences, sound settings, and highest score will be kept in a modifiable text files, therefore, user can change or update the values of these files during game time

2.4. Access Control & Security

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

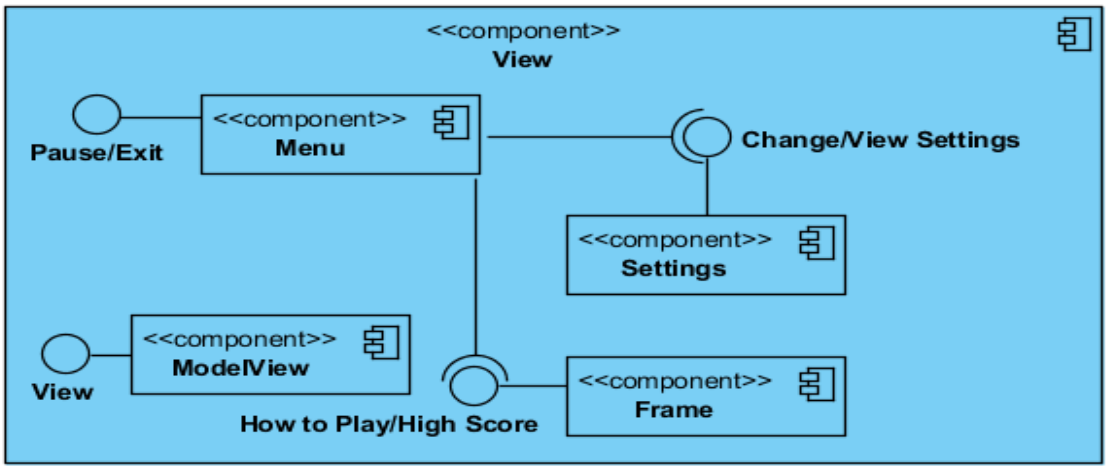
2.5. Boundary Conditions

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

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

3. Subsystem Service

3.1. View Subsystem



View Subsystem is providing user interface for “Space Invaders” game. Four major components of this subsystem are:

1. **Menu Component**
2. **ModelView Component**
3. **Frame Component**

4. **Settings Component**

Before starting the game, **Menu Component** provides a menu to the user. Menu component has a calss for Pause Menu and it contains Exit option. Furthermore, Menu Component can invoke and instantiate other components such as frame component and settings component. Menu component is the interface that provides different functionalities for the interface.

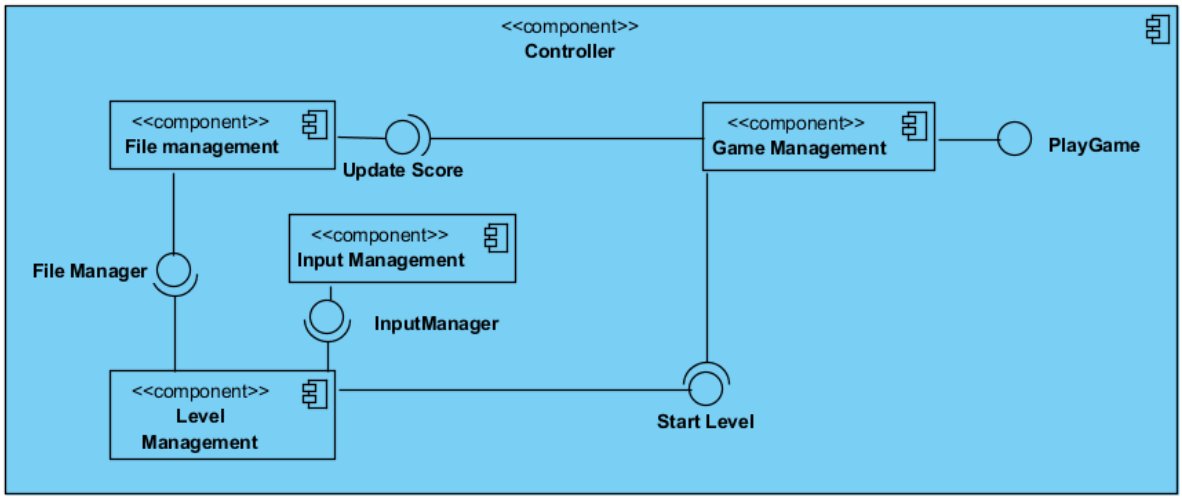
ModelView Component is providing interface for model views and it gets update according to changes in model subsystem from Controller subsystem

A new game can be instantiated only through the **Menu Component.** Menu Component can be invoked by the signals from Controller Subsystem. Those signals are “pause” and “exit game”. Menu Component has a functionality that starts a new game through Menu Class which invokes Game Management subsystem by calling its methods that starts a new game. Façade Design is applied to reduce coupling and increment coherence.

Frame Component provides user interfaces for How to Play and High score table. Frame component can only be invoked by Menu Component.

Settings Component provides settings for the user. The purpose of the settings component is keeping the user settings and changing the settings if the changes are applied. Settings component can only be invoked by Menu Component.

3.2. Controller Subsystem



Controller Subsystem is responsible for controlling the game. This subsystem consists 4 major components:

1. **File Management**
2. **Game Management**
3. **Level Management**
4. **Input Management**

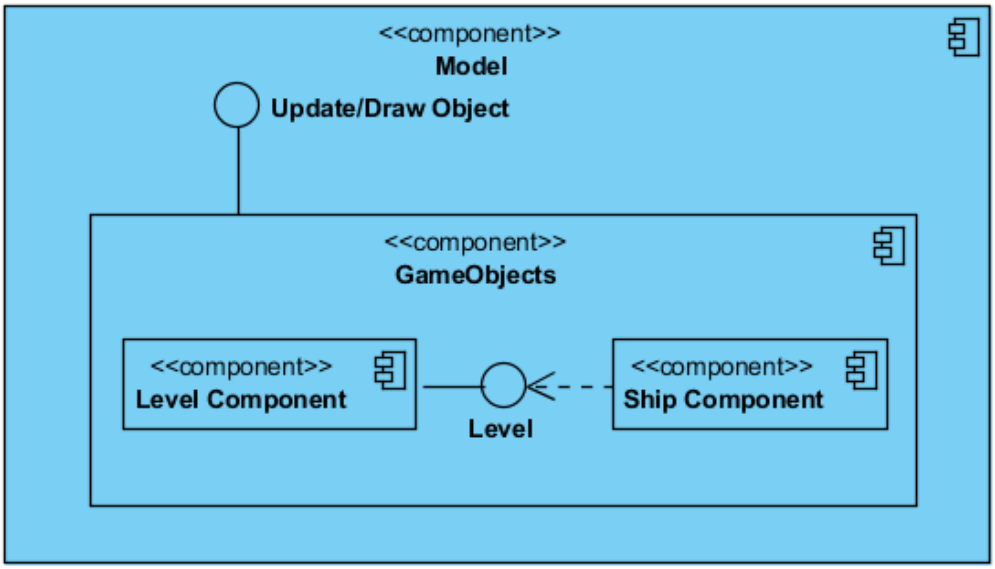
File Management component is responsible for saving the high scores of the players. It is invoked by Level management subsystem at the beginning of the game to get level objects and at the end of the game by Game Management subsystem to update highest score.

Game Management component's purpose is getting the play game input from the View subsystem and it starts a new game. Game management is only responsible for starting a new game, when the level is completed, Menu Component has been invoked during the game is paused or exited. Invoking File Management when the new highest score has been achieved by updating the score.

Level Management component is responsible for controlling the model and tis components. It has **game loop,** level management components and models are updated constantly. Also, it updates the ModelView component of View subsystem. This component also checks the collisions because this component has collision checks. It frequently checks that whether there is a collision or not. Level management update the model objects according to Collision Manager.

Input Management handles the input from the user. When the user gives a specific input, level management has been notified and do a specific function accordingly to input.

3.3. Model Subsystem



Model Subsystem represents the model in the game. It consists of the Game Objects which includes several different types. Model has a main component which is called GameObjects and it represents the different type of game objects and entities. All game entities can only be updated and drawn by the method which is called from Controller Subsystem. Only Management component's level management component can modify the model component's instances. Model class has one component which consists of two subcomponents:

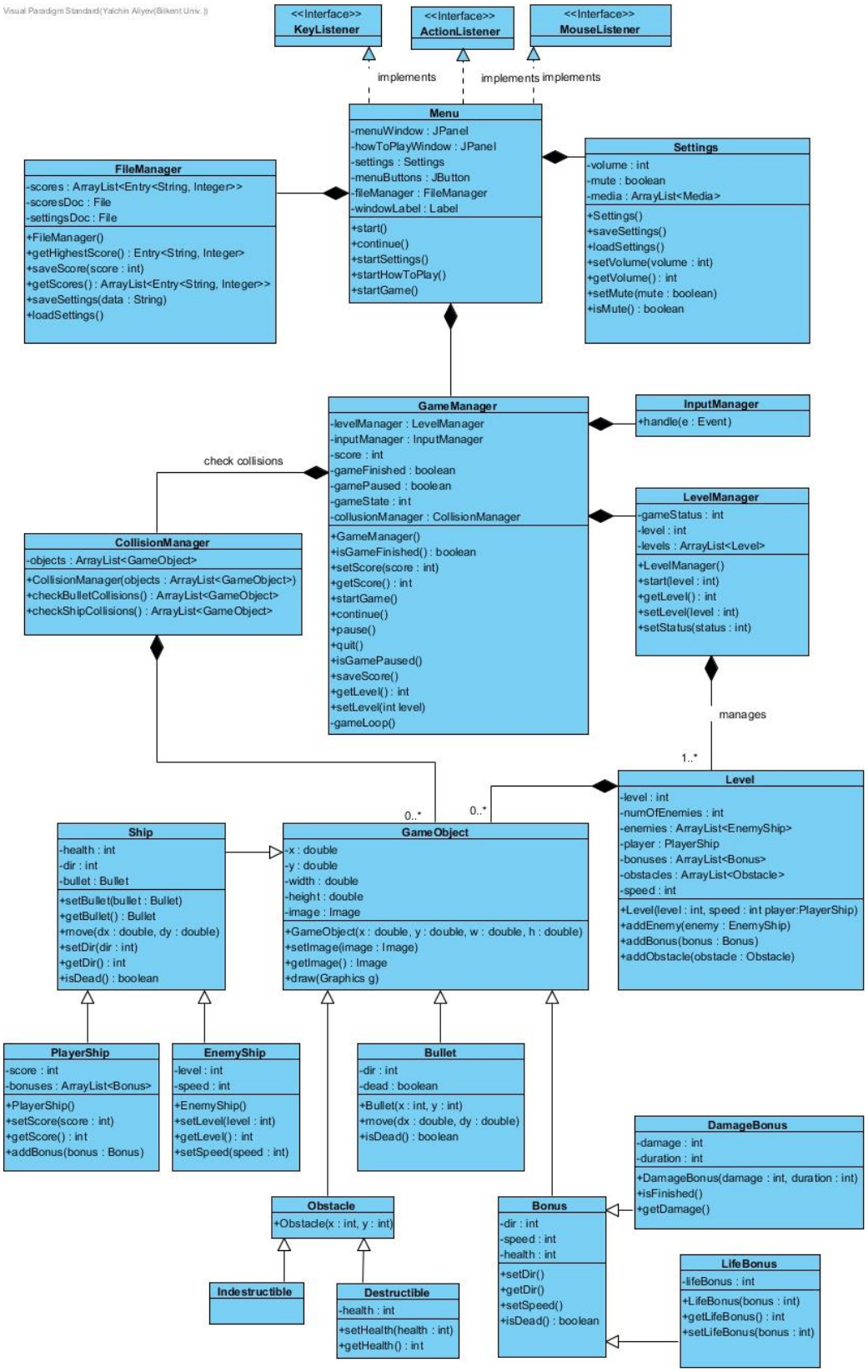
Game Entities: it represents the model objects

Level Component: Represents the bullets, shields and includes their operations.

Ship Components: Represents the player's ship, enemy ships and the bonus ships.

4. Low-level Design

4.1. Final Object Design



4.2. Design Decision Patterns

While processing the low – level design, we used couple of design patterns which have their own trade-offs.

4.2.1 Façade Design Pattern

In order to reduce the coupling between classes we decided to use Façade design pattern. This is a structural design pattern. With the usage of Façade design pattern, we have provided couple of interfaces to interact with the existing objects. For example, connection between the View subsystem and the Model subsystem can occur through the GameManager class. This class provides interfaces for Façade design pattern.

4.2.2 Singleton Design Pattern

We decided to use Singleton design pattern in order to increase the maintainability of the game. There should be only one LevelManager and GameManager during the game. Because a level with more than one manager will most likely cause handling issues and errors during the game. Therefore, usage of Singleton design pattern ensured that there will be only one modifier for the level at once.

4.3. Packages

4.3.1 java.util

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

4.3.2 java.io

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

4.3.3 java.swing

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

4.3.4 java.awt

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

4.3.5 java.awt.event

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

4.4. Class Interfaces

These are the detailed and clear explanation of our class interfaces.

4.4.1 Menu Class

- **private JPanel menuWindow:** This attribute will provide the Menu object layout in the Menu, via JPanel
- **private Settings settings:** This attribute will be used to create the panel for the Settings screen
- **private JPanel howToPlayWindow:** This attribute will be used to create the panel for the How to play screen
- **private GameManager gameManager:** This attribute will be used in order to call the gameManager object from GameManager class
- **private FileManager fileManager:** This attribute will be used in order to call the menuFileManager object from FileManager class

Constructors:

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

Methods:

- **public void start()**: starts a new game panel and game manager
- **public void continue():** continues the game beginning from the previous game state
- **public void startSettings():** opens a new panel for settings and puts settings on this panel
- **public void startHowToPlay():** opens a new panel to show how to play screen

4.4.2 Settings Class

Attributes:

- **private int volume**: This attribute will be used to decide the volume of the sound
- **private Boolean mute**: This attribute is to decide whether the sound os muted or not
- **private ArrayList<Media>**: This is a list of medias that will be played during the game

Constructors:

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

Methods:

- **public void saveSettings():** Saves settings data
- **public void loadSettings():** Loads the previously saved settings
- **public void setVolume(volume: int):** Set the volume to the specified value

4.4.3 InputManager Class

Methods:

- **public void handle(e: Event):** This method handles the event according to the event type and source

4.4.4. GameManager Class

Attributes:

- **private LevelManager levelManager:** This attribute is the instance of the LevelManager class and provides a connection between the logic of the game and the level design.
- **private InputManager inputManager:** This attribute is the instance of the InputManager class and controls the user inputs during the game.
- **private int score:** This attribute keeps track of the score of the player in the game.
- **private boolean gamePaused:** This attribute keeps track of whether the game is paused or not.
- **private int gameState:** This attribute tracks the level that player is playing.
- **private boolean gameFinished:** This attribute keeps track of whether the game is finished or not.
- **private CollisionManager collisionManager:** Game manager uses this to decided on collisions of game objects.

Constructor:

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

Methods:

- **private boolean isGameFinished():** This method checks whether the game is over or not.
- **private void setScore():** This method updates the score of the player.
- **private void startGame():** This method starts the game from the level 1.
- **private void finishGame():** This method finishes the game and the game state cannot be recovered after this.

- **private boolean isGameFinished():** This method checks whether the game is running or not.
- **private boolean isGamePaused():** This method checks whether the game is paused or not.
- **private void pause():** This method pauses the game.
- **private void continue():** This method continues the game if the game is in a pause state.
- **private void quit():** This method quits the game.
- **private void saveScore():** This method saves the score of the player.
- **private void setLevel(nextLevel: int):** This method starts the next level, if the user finish the level.
- **private void gameLoop():** This is the main game loop of the game that is being run by a thread if the game is in running state. Simplified version of this method is given below.

```
private void gameLoop() {
      while(!isGameFinished && !isGamePaused) {
            update_model();
            update_view();
      }
}
```

4.4.5. LevelManager Class

Attributes:

- **private int gameStatus:** This attribute saves the current state of the game. (In other words, checks whether the game is running or paused etc.)
- **private int level:** This attribute keeps the count of the level that the player is currently playing.

- **private ArrayList<Level> levels:** This list contains all possible levels of the game

Constructors:

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

Methods:

- **private void setLevel(level: int):** This method updates the level of the game.
- **private void start(level : int):** This method starts the new level with the specific level count.
- **private void setStatus(status: int):** This method sets the game status.

4.4.6. Collision Manager Class

Attributes:

- **private ArrayList<GameObject> objects:** This attribute contains all the objects in the game

Constructor:

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

Methods:

- **public void checkBulletCollisions():** This methods checks whether the bullets hit other game objects and return the game objects that they hit. All of the game objects are rectangular, that is why this method checks whether the rectangles around game object intersects.

- **public void checkShipCollisions():** This method checks collision for all ships including player and the enemy ship with each other or bonus objects or obstacles and return list of collided objects. In the same way, this method checks whether the rectangles around game object intersects.

4.4.7 FileManager Class

Attributes:

- **private ArrayList<Entry<String, Integer>> scores:** This is a list of scores
- **private File scoresDoc:** This is the file that scores have been stored.
- **private File settingDoc:** This is the file that settings are stored

Constructor:

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

Methods:

- **public void saveScore(int Score):** This method gets the score from the GameManager and saves it to high score text file.
- **public int getHighestScore():** This method gets the high score from high score text file.
- **public ArrayList<Entry<String, Integer>> getscores():** This method gets scores from the file.
- **public void loadSettings ():** This method gets the player settings from settings text file.
- **public void saveSettings(data: String):** This method gets the changed settings from Settings and saves the new input settings to settings text file.

4.4.8 Level Class

Attributes:

- **private int level:** This attribute will hold the level.
- **private int numOfEnemies:** This attribute will hold the number of remaining enemies in the current level.
- **private ArrayList<EnemyShip> enemies:** This attribute will hold the remaining enemies in the current level.
- **private PlayerShip player:** This will hold a player instance.
- **private int speed:** This attribute will be used to determine the speed of enemies in this level.
- **private ArrayList<Bonus> bonuses:** This will hold all the bonuses for this level.
- **private ArrayList<Obstacle> obstacles:** It will contains obstacles for this level.

Constructors:

- **Level():** Default constructor of the Level class.
- **Level(level: int, speed: int, player: PlayerShip):** This is the constructor which creates the new level with the level number and the speed for the specified player.

Methods:

- **public boolean addEnemy(EnemyShip):** This method adds enemy ship to the list.

4.4.9 GameObject Class

Attributes:

- **private double x:** This attribute holds the top left point of the object.

- **private double y:** This attribute holds the top point of the object.
- **private Image image:** This attribute holds image file for objects.
- **private double width:** This attribute holds the width of the object.
- **private double height:** This attribute hold the height of the object.

Methods:

- **public void updateLoc():** This method changes the x and y values of the object.
- **public void resize():** This method changes the width and height of the object.
- **public void setImage(Image):** Updates the object's image that will be drawn on the screen.
- **public void draw(Graphic g):** Draw the Image on the screen.

4.4.10 Ship Class

Attributes:

- **private int health:** This attribute holds the current health of the ship.
- **private int dir:** This attribute determines the direction of the bullet and image(front of ship).
- **private Bullet bullet:** This is the bullet that ship will shoot.

Methods:

- **public void setBullet(Bullet):** This method set a bullet object.
- **public boolean isDead():** This method determines whether the ship is dead or not.
- **public void move(dx: double, dy: double):** This method requests the movement in the space.

4.4.11 PlayerShip Class

Attributes:

- **private int score:** This attribute holds the score of the player, which is incremented by killing bots and finishing levels.
- **private List<Bonus> bonuses:** This will contain bonuses that is collected by the player.

Constructors:

- **player():** The constructor of the player class to create a player instance.

Methods:

- **public void setScore():** This method updates the score of player when a level finishes or player kills an enemy.
- **public void addBonus(Bonus):** This method adds bonus to the list if it is intersected in the space.

4.4.12 EnemyShip Class

Attributes:

- **private int level:** This determines the level, that is how strong the enemy is.
- **private int speed:** This is the speed of the enemy.

4.4.13 Bullet Class

Attributes:

- **private int dir**: This attribute holds the owner of the bullet.
- **private Boolean dead:** Keeps the state of the bullet.

Constructors:

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

Methods:

- **public void move(dx: double, dy: double):** This method provides the movement logic for the bullet.
- **public boolean isDead():**

4.4.14 Obstacle Classs

Constructors

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

4.4.15 Destructible Class

Attributes:

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

Methods:

- **public boolean isDestroyed():** This attribute determines if the object is destroyed or not.
- **public void setHealth(health: int):** Setting the health of the obstacle.

4.4.16 Bonus Class

Attributes:

- **private int dir:** Determines direction of its movement.
- **private int speed:** Determines the speed of its movement.
- **private int health:** Stores the health of this object.

4.4.17 DamageBonus Class

Attributes:

- **private int damage:** Determines bonus increase in the players damage.
- **private int duration:** The seconds that it will be effective for the player.

4.4.18 LifeBonus Class

Attributes:

- **private int lifeBonus:** Determines increase in the players health.

5. Improvement Summary

We have added additional features for the second iteration of the project, such as bonuses and new level design. We have also improved our high-level and low-level design. Improvements includes addition and subtraction of classes. Here is the list of important changes.

- Our main architectural design pattern stayed as MVC but in this iteration, Controller subsystem contains only the functionalities for getting user input and asking Model to handle it.
- In this report we used lollipop component diagrams to show the dependencies between subsystems. Model subsystem is completely independent from the others. There is two-way association between Controller and View and one-way association from Controller to Model.
- We have also decomposed each subsystem into smaller subsystems depending on functionalities of classes.
- We have added new classes such as Bonus, LifeBonus and DamageBonus
- We have also deleted some classes which seemed unnecessary such as IronWall, Wall, ViewFrame.
- We have improved system decomposition part of this report which discusses subsystems in more detail.
- To achieve a complete object-oriented design, we also used design patterns in each subsystem. These design patterns include Singleton Design Pattern and Façade Design pattern.
- In the low-level design, the class diagram is also updated according to the new features and the system decompositions.

6. Glossary & References

- Java Docs

  Java™ Platform, Standard Edition 7, API Specification,

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