Bilkent University

Department of Computer Engineering

CS-319 Project

Defenders: Cinematic Edition

Design Report

Group 3E

Daniyal Khalil, Leyla Ismayilova, Leyla Hashimli, Emil Alizada, Zulfugar Verdiyev

Instructor: Eray Tuzun
Teaching Assistant(s): Baris Ardic

Design Report

Nov 10, 2019

Contents

Design Report

Defenders

1. Introduction

1.1. Purpose of the System

Defenders is a renewed version of Arcade game Defender 1981. The main purpose of the game is similar to the previous version. The player must attack enemies and protect his/her character from their attacks. Additionally, this game brings extra features such as store and multi-theme option. The main aim of adding these features is addressing the expectations of today's games. Since alien attacks are no longer a trending theme in the game industry. Moreover, these options increase the interest of the user and the competition between them.

Defenders game is planned to be a graphically quality game to ease the learning period and get the user more involved in the game. Moreover, this game is convenient and tactical in terms of strategic game playing and requires quick reflexes.

1.2. Design Goals

While designing the game, it is important to identify the priorities and set well-defined goals. The main goal of this version is to entertain the user and bring back the old reputation by making it to-day. Therefore, we plan to design our game to meet the needs of today's users and be modifiable enough for future use. The following sections include a detailed description of design goals.

1.2.1 End user criteria

Usability: To make the game usable we followed intuitive design for our menu system that most games use. For the menu, we use simple English and avoid using sophisticated words or terms related to themes. Also, the similarity of the menu with

other games eases the work of the player and directly sends him/her to the game itself. User does not need to choose a player or theme for the first time since the game supplies the player with default ones. By these features, when the user plays the game for the first time, he/she does not need to learn anything. Moreover, the game structure is preserved to ease the work of the user. The game uses cursor keys for moving the character and character is limited to only going upward, downward, left and right. Using common control keys for playing the game eases the use and the user does not need to remember them.

1.2.2 Performance Criteria

Defenders game requires quick reflexes. Having minimal response time is very crucial for the user, since the game is very interactive and any delay can result in the death of the character. Moreover, we use 30FPS for the animation of the player and enemy characters.

1.2.3 Maintenance Criteria

Extendibility: This game aimed to meet the standards of today's game industry and the expectations of the users. However, user expectations change constantly. Therefore, the design of this game enables the designers to change it in the future based on new needs or user feedback. To illustrate, this version of the game has only one level, however, in the future game can be extended to have multiple levels to increase the interest.

Modifiability: Defenders game is designed adjustable for future modifications. Because of the multilayered design, it is easy to have alterations. The dependencies between subsystems are minimized to increase modifiability. Through this, we can easily modify some parts of the game without changing the other. For example, today our themes are famous and well-known, however, in the future, this can change. Due to our design, modifying the themes will not affect the game itself.

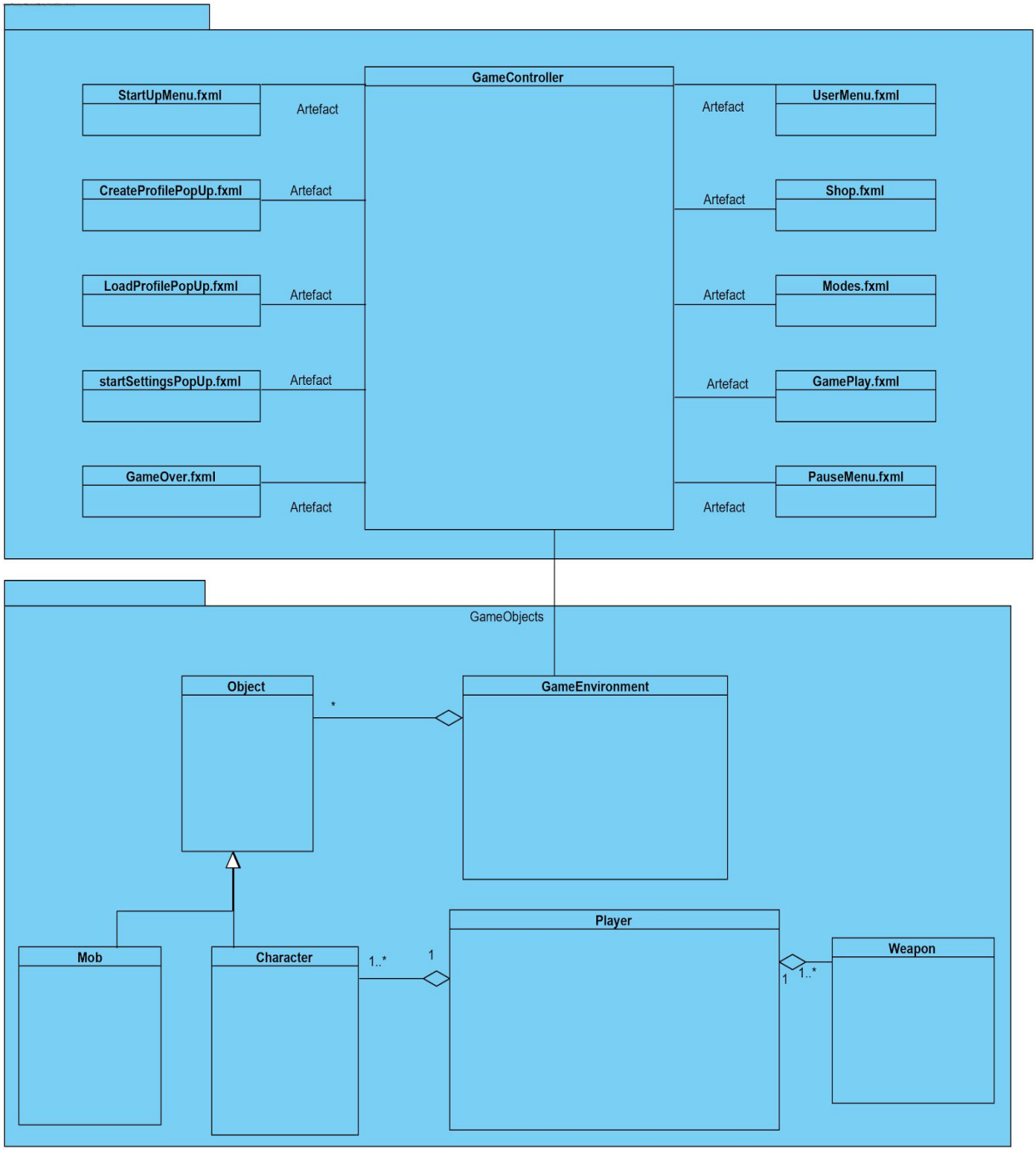
Portability: To enhance the target of the users, we designed our game to run on any computer. Considering the importance of portability, the game is implemented on Java. Thus, this game can be played on any machine that has Java Virtual Machine installed.

2. System Architecture

2.1. Subsystem Decomposition

We have decomposed our system into 3 main subsystems. They are Game Control, Game Play and, Storage and Objects. Our main purpose is to reduce the coupling between each subsystem, where considering that Defenders is plainly simple game, decomposition should not be very complicated. Decomposing our subsystems in this particular way, will allow us to modify each part without necessarily changing the others.

In our design, we used MVC (Model-View-Controller) architectural style with 3 subsystems, but here it will have only 2 subsystems because the view would be generated using FXML sheets and Java Scene Builder. Our Game Control subsystem is basically a set of interactions that user can have. Since we are using JavaFX, we are importing relevant packages for user to be able to easily communicate with the game interface. Thus avoiding any irrelevant listeners and interfaces. For Game Play subsystem, we are mostly using fxml sheet instead of classes given the documentation of JavaFX.


GameController
StartUpMenu.fxml
CreateProfilePopUp.fxml
LoadProfilePopUp.fxml
startSettingsPopUp.fxml
GameOver.fxml
UserMenu.fxml
Shop.fxml
Modes.fxml
GamePlay.fxml
PauseMenu.fxml
Artefact
GameObjects
Object
GameEnvironment
Mob
Character
Player
Weapon
1..*
1

2.2. Hardware/Software Mapping

We are implementing our game in Java programming language and use JavaFX libraries. So, execution of Defenders game is going to require Java Runtime Development as software. Since we are implementing JavaFX libraries using IntelliJ IDEA as an IDE, each of us has supplied Java Development Kit 8, to match the SDK version that is used in our IDE. Moreover, the version is compatible with JavaFX libraries.

As a hardware requirement, the Defenders game requires a keyboard. It will supply the connection between the user and the game. Keyboard inputs will regulate menu selections, and also movements and fires in the game play.

For convenience, since our game will require an extra storage for user details and statistics, we will use text files to keep the data. Thus, we do not need any extra segment for internet connection to database or JSON files.

2.3. Persistent Data Management

The Defenders game does not require any online connection or database to operate storage. We are using text files for any storage of information (such as scores and profile information). The game stores old profiles with their preferences of theme and avatar, their highest score. As the old profile is selected to play the game, the data related to this profile will be modified if it is needed.

In addition, we are going to use wav and mp3 formats for our in-game sound effects. Mode, avatar data will be stored as PNG.

2.4. Access Control and Security

Since the game does not require any internet connection to operate, any security forces is expected. After installing game, the user does not need any access from any portals.So the implementation does not involve any access control system or security

guard. Data storage of the game will be unique for each installation, any change in one computer will not be modified in any other installations.

2.5. Boundary Conditions

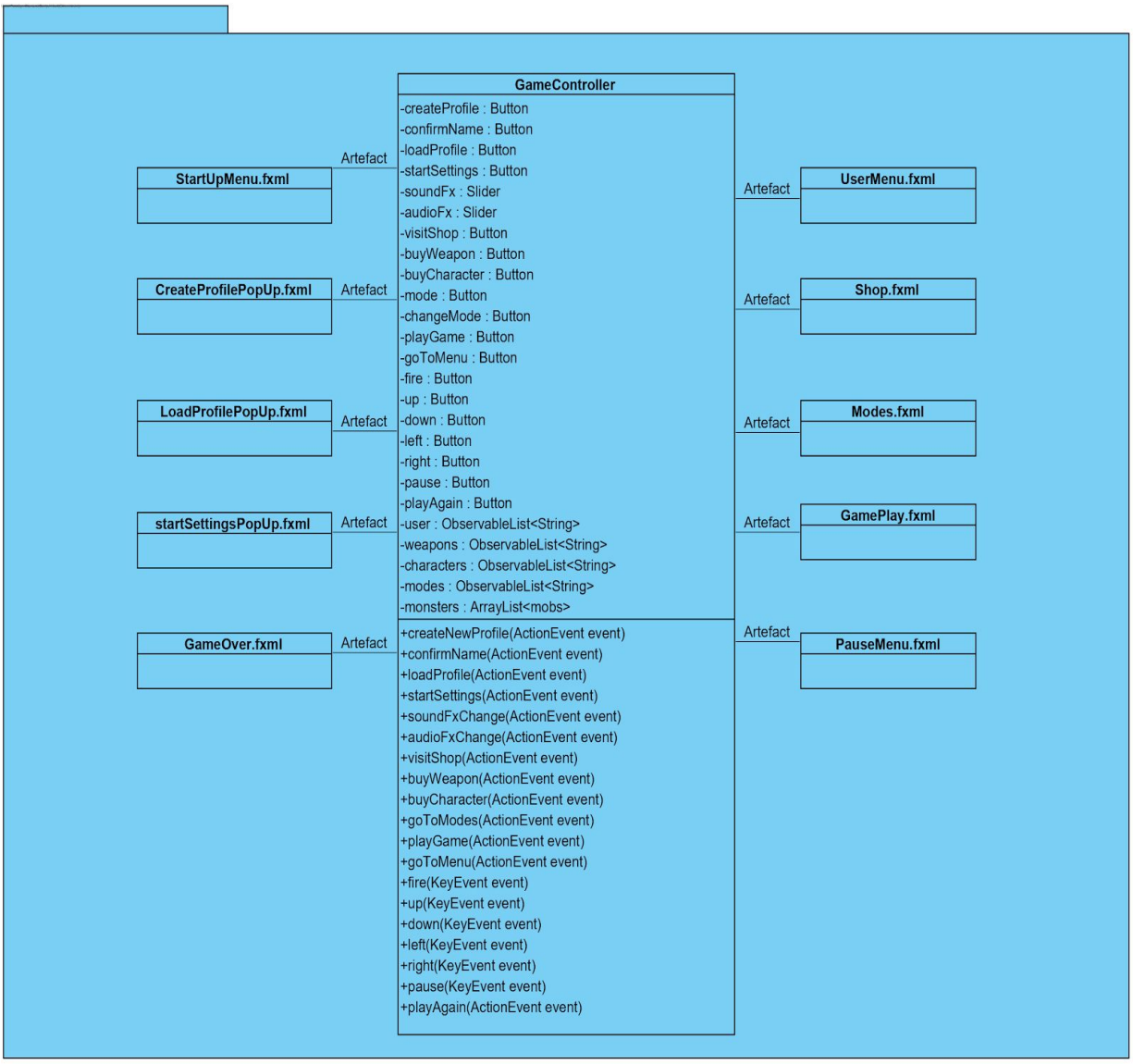
Defenders require installation since we will have executable .exe. With that we boost up the portability of the program since the user does not need to install and JDK or SDK. When the system is initialized all the necessary variables, constants and tasks will be executed at once.

Termination will occur using exit button. Our exit button is different from the game screen's built-in exit button, which additionally will ensure the safety of the exit. In-game pause will not serve as the termination point.

When we terminate the program multiple unexpected bugs may occur. If the user terminates the game using screen's own exit button, the user's last activity may not be updated. Since we are using text file it is also possible that terminate the program without properly adjusting its contents. For that we plan on preparing additional notifications to the user to safely exit the program.

3. Subsystem Services

3.1. Game Control Subsystem



The User Interface for the game appears to be simpler than other games due to the game being very straightforward and not containing much complexity other than collision will be dealt within the game management systems. The UI consists of only one Controller class, the GameController which controls all the actions within separate FXML sheets, which are different windows at different stages of the Game.

GameController Class

Game Controller Class, is the main user interface controllers for the Defenders. All the separate frames the game will contain are all made in FXML sheets using Java Scene builder. The Game Controller class loads the FXML sheet at proper actions and times within the game for the User Interface. It has separate operations for each ActionEvent or KeyEvent that will take place within the game.

ATTRIBUTES

private Button createProfie: The createProfile button from the StartUp Menu.

private Button confirmName: The confirm button in the PopUp for entering username after create Profile.

private Button loadProfile: The loadProfile button from the StartUp Menu.

private Button startSetttings: The settings button in the StartUp Menu.

private Slider soundFx: The Slider for adjusting Sound effects in the Settings.

private Slider audioFx: The Slider for adjusting Audio in the Settings.

private Button visitShop: The Button for visiting shop in the User Menu.

private Button buyWeapon: The Button for buying weapons in the Shop.

private Button buyCharacter: The Button for buying characters in the Shop.

private Button mode: The Button for Modes in the UserMenu.

private Button changeMode: The Button to change the Game mode.

private Button playGame: The Button to start a new game in the UserMenu.

private Button goToMenu: The Button to go back to UserMenu from the Shop and Modes.

private Button fire: The Button to fire the projectile from the User.

private Button up: The Button for the user to move up.

private Button down: The Button for the user to move down.

private Button left: The Button for the user to move left.

private Button right: The Button for the user to move right.

private Button pause: The Button for pausing the game while it is Playing.

private ObservableList<String> users: The Observable List for List View to show the users list in the loadProfilePopUp.

private ObservableList<String> weapons: The Observable List for List View to show the weapons in the Shop.

private ObservableList<String> characters: The Observable List for List View to show the characters in the shop.

private ObservableList<String> modes: The Observable List for List View to show the modes in the mode.

private ArrayList<mobs> monsters: To keep track of the monsters that can be called in gamePlay.

METHODS

public void createNewProfile(ActionEvent event): Launches the createNewProfilePopup after the Create Profile Button is pressed.

public void confirmName(ActionEvent event): Confirms the entered name in the Popup.

public void loadProfile(ActionEvent event): Launches the loadProfilePopUp after the Load Profile Button is pressed.

public void startSettings(ActionEvent event): Launches the startSettings Window in after Settings is pressed.

public void soundFxChange(ActionEvent event): Changes the slider in settings for sound.

public void audioFxChange(ActionEvent event): Changes the slider in settings for audio.

public void visitShop(ActionEvent event): Lauches the Shop after shop button is pressed in the usermenu.

public void buyWeapon(ActionEvent event): Buys weapons after in the shop.

public void buyCharacter(ActionEvent event): Buys character after in the shop.

public void goToModes(ActionEvent event): Launches Modes Window after Modes is pressed.

+

Launches the Game Window after Play Game is pressed.

void goToMenu(ActionEvent event): Used to go back to User menu from shop or modes after pressing Back.

void fire(ActionEvent event): A projectile is fired after player presses Z.

void up(ActionEvent event): Player moves up after pressing arrow UP.

void down(ActionEvent event): Player moves down after pressing arrow DOWN.

void right(ActionEvent event): Player moves right after Pressing arrow RIGHT

void left(ActionEvent event): Player moves left after Pressing arrow LEFT.

void pause(ActionEvent event): The game pauses after Pressing P.

void playAgain(ActionEvent event): Play Again after the game is over by pressing this.

FXML

StartUpMenu : The starting menu that the game provides.

CreateProfilePopUp: The PopUp that appears for user prompt after Create Profile Button has been pressed

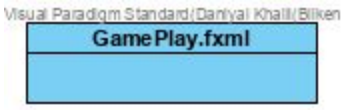
LoadProfilePopUp: The Popup that appears that has a list of profiles for the User to select from.

StartSettingsPopUp: The settings menu that Pops up after Settings is pressed.

UserMenu: The User menu after a user has either created or loaded their profile

Shop: The Shop menu after the user has entered the shop.

Modes: The modes menu after the user has entered the modes.

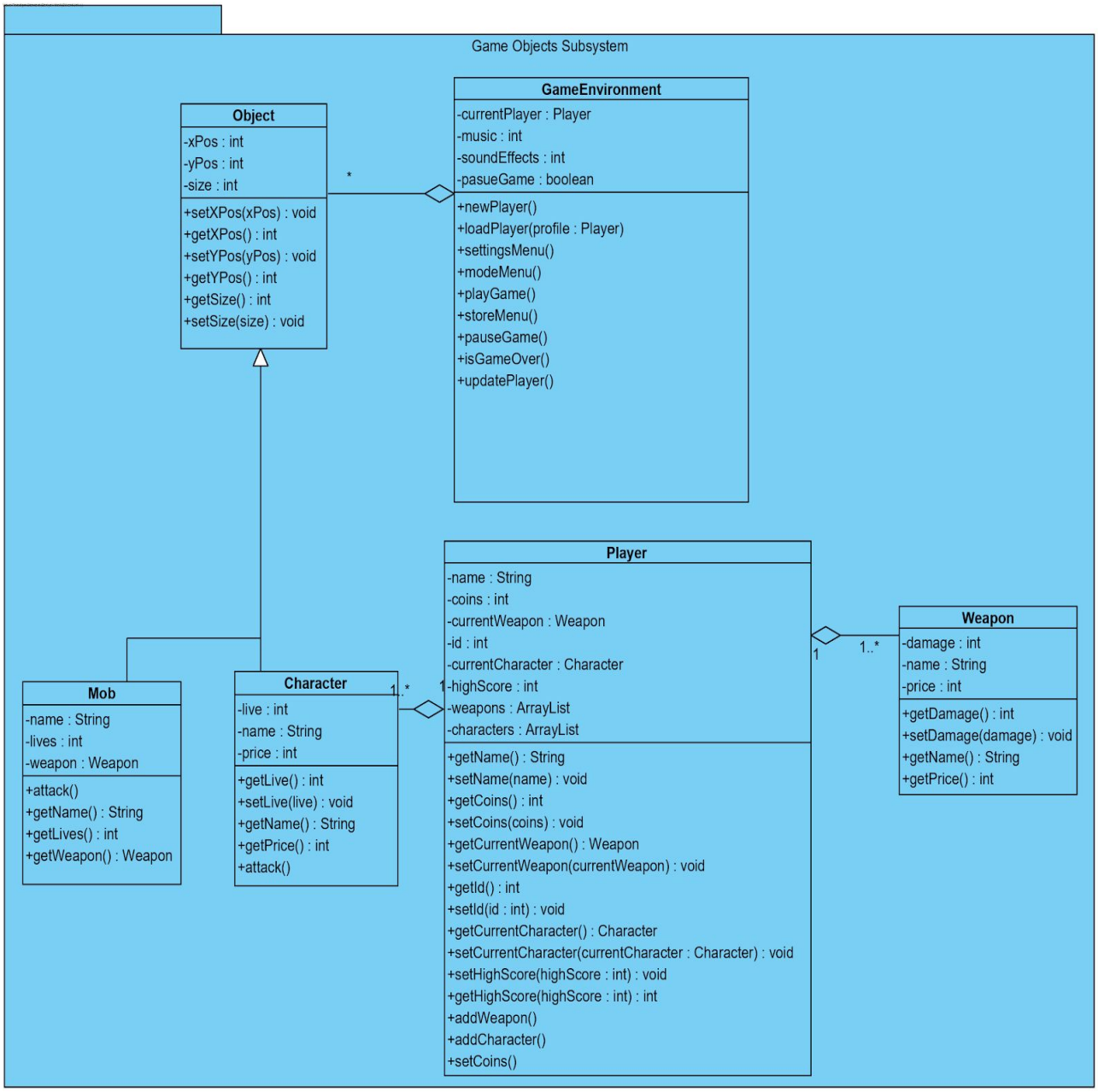


GamePlay: The game play after the user presses Play Game.

PauseMenu: The pause menu after the user presses pause while playing the game.

GameOver: The Popup that appears after the game is over.

3.2. Game Object Subsystem



Game Objects Subsystem is the subsystem responsible for maintaining the operational and logical processing of the game.

GameEnvironment

ATTRIBUTES

private Player currentPlayer: It is the Player that our processing class is going to have.

private int music: It is for playing music (set the variable to 1).

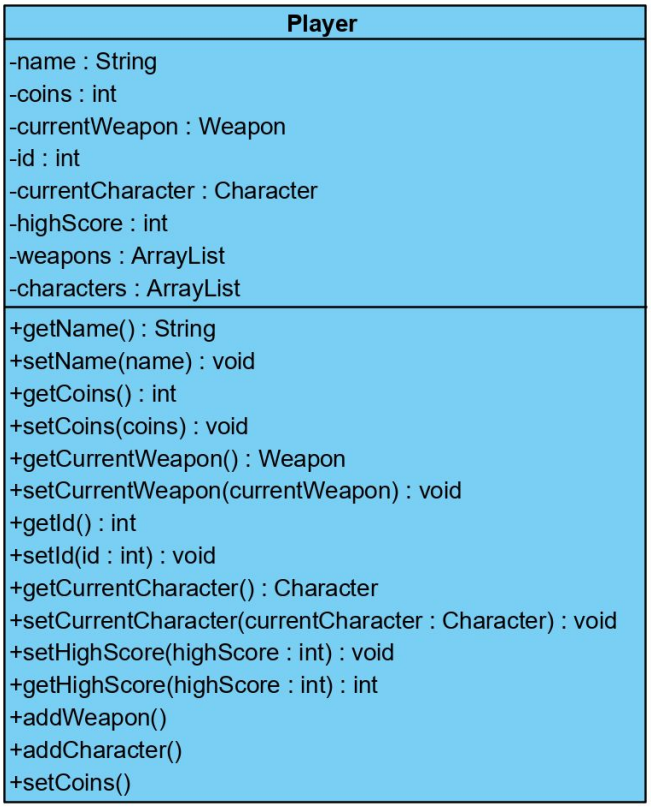
private int soundEffects: Controlling the sound, within a range of 0 and 1.

private boolean pauseGame: True if the game is paused.

METHODS

public void newPlayer(String name): It generates new player.
public void loadPlayer(Player profile): It loads the given player.
public void settingsMenu(): It returns the options for menu.
public void modeMenu(): It returns the modes of menu
public void playGame(): When you call the function it starts the game.
public void storeMenu(): Redirects you to store menu.
public void pauseGame(): Returns true if the game is paused.
public boolean isGameOver(): Returns true if game is over.
public void updatePlayer(): Changes the player.

Player



ATTRIBUTES

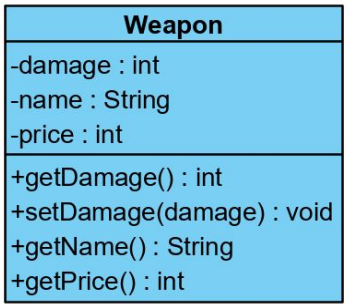
private String name: Name of the player.
private int coins: number of collected coins.
private Weapon currentWeapon: The current weapon that is used.
private Character currentCharacter: The current character that player uses.
private int highscore: Uploads the user's highest score.
private ArrayList<Weapon> weapons: The list of weapons collected by the player.
private ArrayList<Character> characters: The list of characters won by the player (as a trophy).

METHODS

public String getName(): Returns the name of the player.
public void setName(String name): Sets the name of player.
public int getCoins(): Returns the number of coins collected for a single game.

public void setCoins(int coins): Sets the number of coins gathered.
public Weapon getCurrentWeapon(): Returns the current weapon that is used.
public void setCurrentWeapon(): Sets the weapon as currently used.
public int getId(): Returns the ID.
public void setId(int id): Sets the ID for the user
public Character getCurrentCharacter(): Returns the current character played by the user.
public void setCurrentCharacter(Character character): Sets the character as currently used.
public void setHighScore(int score): Sets the highest score keeps it in the memory.
public int getHighScore(): Returns the highest score.
public void addWeapon(Weapon weapon): Adds the weapon to the collected weapons.
public void addCharacter(Character character): Adds the character to the collected characters.

Weapon



ATTRIBUTES

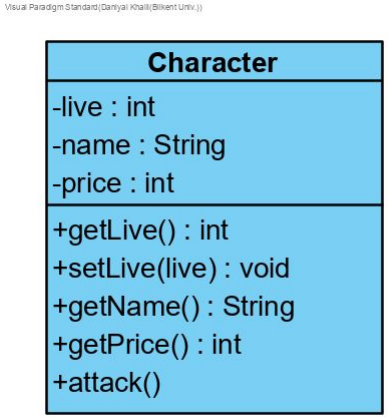
private int damage: Damage level of the weapon.
private String name: Name of the weapon.
private int price: Price of the weapon.

METHODS

public int getDamage(): Returns damage level of the weapon.

public void setDamage(int damage): Sets the damage level for that specific weapon.
public String getName(): Returns the name of the weapon.
public int getPrice(): Returns the price of the weapon.

Character



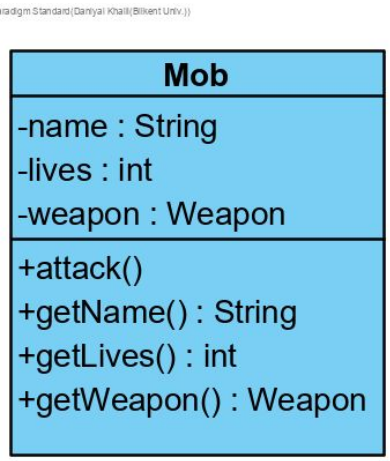
ATTRIBUTES

private int live: Life range of the character.
private String name: Name of the character.
private int price: Price for getting the character.

METHODS

public int getLive(): Returns the life range of the character.
public void setLive(int live): Sets the life range of the character.
public String getName(): Returns the name of that character.
public int getPrice(): Returns the price.
public void attack(): Sets the player in attack mode.

Mob



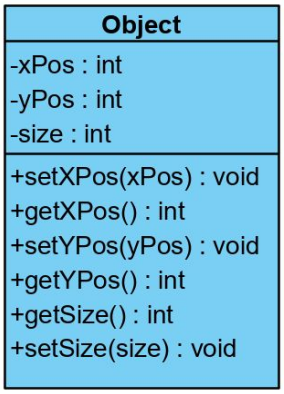
ATTRIBUTES

private String name: The name of the mob.
private int lives: The lives that a mob has.
private Weapon weapon: The weapon that mob uses.

METHODS

public void attack(): Sets the mob in attack mode.
public String getName(): Returns the name of the mob.
public int getLives(): Returns the lives that a give mob has.
public Weapon getWeapon(): Returns the weapon that a mob uses.

Object



ATTRIBUTES

private int xPos: X position of the object.
private int yPos: Y position of the object.
private int size: Size of the object.

METHODS

public void setXPos(int xPos): Sets the X position for the object.
public int getXPos(): Returns the X position of the object.
public void setYPos(int yPos): Sets the Y position for the object
public int getYPos(): Returns the Y position for the object.
public int getSize(): Returns the size of the object.
public void setSize(int size): Sets the size of the object.

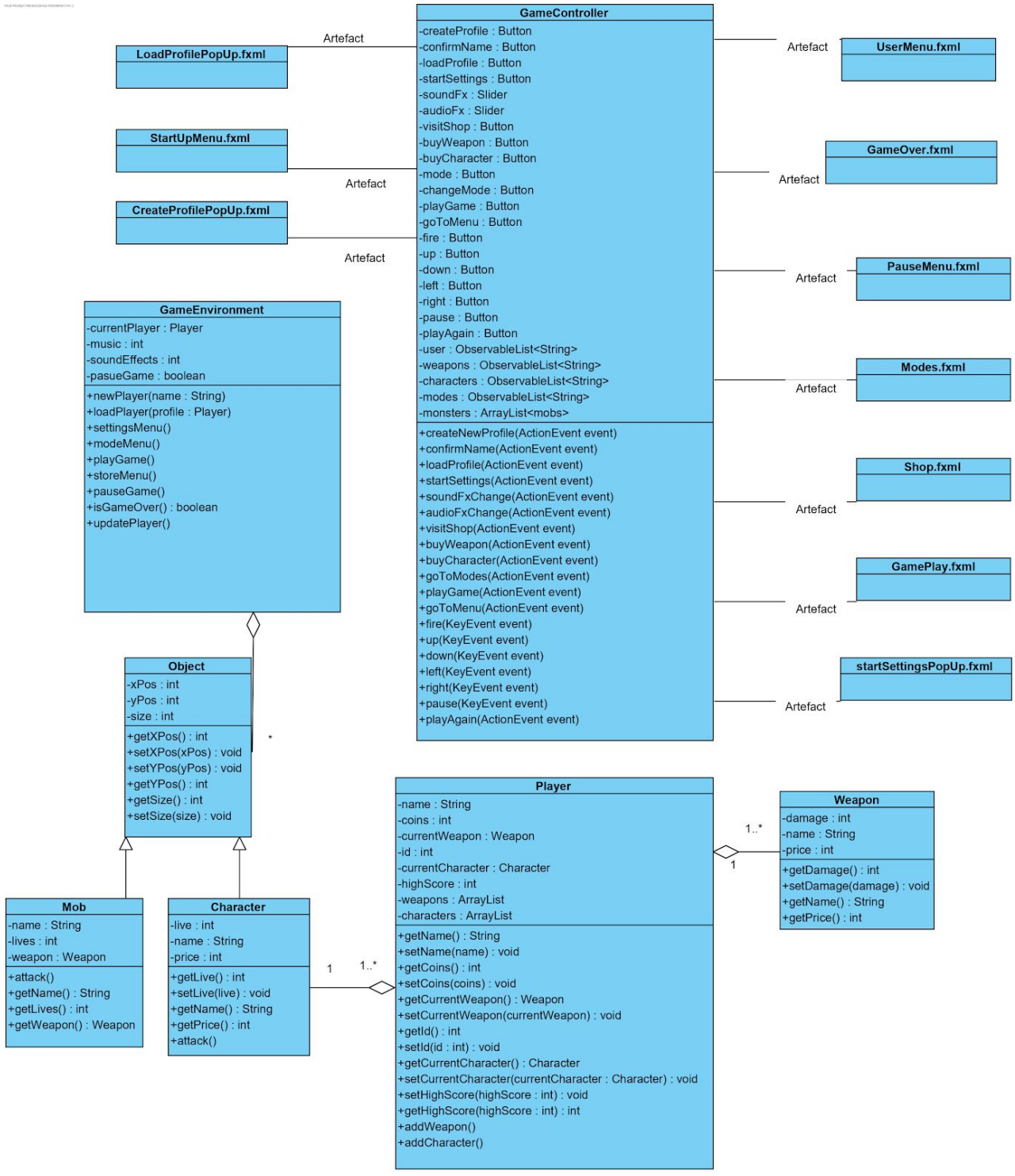
4. Low-Level Design

4.1. Object Design Trade-Offs

Performance vs memory: Our game needs to have minimal response time and be as fast as possible to give the user the best experience. Therefore, we choose performance over memory. Besides, since Defenders is a small game, memory should not be a problem.

Security vs Usability: To make the game more usable and ease the work of the user we did not implement accounts with authentication for each user. Instead, each time a user can write his/her name or choose one of the previously played profiles. Therefore, the user can choose the profile of another person and use it as own. Thus, in terms of security, this game does not assure the user to private access of his/her profile.

4.2. Final Object Design



4.3. Packages

java.util: The utility class used for ArrayList and Scanners:

javafx.collections: The Collection class for maintaining Observable Lists for weapons, profiles and characters.

javafx.fxml: The FXML sheet class for maintaining the FXML and using its functions.

javafx.geometry.Orientation: Used for the orientation of the ListViews in the scenes.

javafx.scene: The class that is used for panes, pop-ups, layouts, ListViews and all the children components of a pane.

javafx.event: The class that implements the actions for all the listeners in the code.

javafx.stage.*: Used for the setting of stages after change in the windows.

5. Glossary & References