BILKENT UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

CS 319 - Object Oriented Software Engineering

Term Project Iteration 2

FIGHT OR FLIGHT

System Design Report

17/04/2018

Group 1D

Mihri Nur Ceren

Adahan Yalçınkaya

Berk Erzin

Emre Sülün

Table of Contents

1. Introduction

1.1. Purpose of the system

The purpose of Fight or Flight is to provide a fun and exciting experience to the players. The game is going to be dynamic, fast paced and challenging with different kinds of enemies so the player would feel challenged to reach high scores while having unique experiences with fun game-play mechanics.

1.2. Design goals

The main actor of game is the player and main goal should be creating an entertaining and competitive game. From the observations and from the nonfunctional requirements, the following design goals are identified.

1.2.1. Performance

- **Response time:** The response time of the menu and the game mechanics should be less than 5 ms.

1.2.2. Dependability

Since the game is not a safety-critical system; robustness, fault tolerance and security risks are not main concerns of the design.

1.2.3. Cost

- **Development cost:** The design should be implementable in 2 months.

1.2.4. Maintenance

- **Modularity:** Design should allow swapping of similar objects, instances and etc., thus allowing easier modification to software.
- **Portability:** The game will be implemented in Java.

1.2.5. End user

- **Usability:** Our game involves useful functionalities. It makes our project user friendly. For example, there are some basic buttons on the main menu so that any player who knows basic English can play easily. Also, the game does not need any installment.
- **Understandability:** Before starting the game, players who do not know how the game is played are able to access some definitions of the game by using the help button.

1.3. Definitions, acronyms and abbreviations

- **MVC:** Model View Controller
- **Flight Stage:** The running stage of the game where the player runs and avoids obstacles.
- **Fight Stage:** The fight stage of the game where the player fights enemies on a freeze frame

1.4. References

- Bruegge, B, & Dutoit, A 2014, Object-Oriented Software Engineering : Using UML, Patterns, And Java, n.p.: Boston : Prentice Hall, 2014., Bilkent University Library Catalog (BULC), EBSCOhost, viewed 16 February 2018.

1.5. Improvement summary

- Revised design goals.
- Decided upon which design patterns to employ.
- Added deployment diagram.
- Persistent data management improved.
- Added control flow.
- Corrected some formatting errors in the document.

2. High-level software architecture

High-level software architecture is the top level design of the software. It includes subsystem decomposition, architectural style, hardware/software mapping, persistent data management, access control, global control flow and boundary conditions.

2.1. Subsystem decomposition

To reduce complexity and distributing tasks among team members, we divided our system into three subsystems: game logic, controller and user interface. This decomposition is done according to model-view-controller

2.2. Architectural style

We organized the subsystems according to model-view-controller style. Classes in **Game Logic Subsystem** are responsible for game objects regardless of interface. **User Interface Subsystem** classes control the interface and render the screen according to model. **Controller Subsystem** classes establish connection between model and view. User interface classes communicate game logic via controller subsystem.

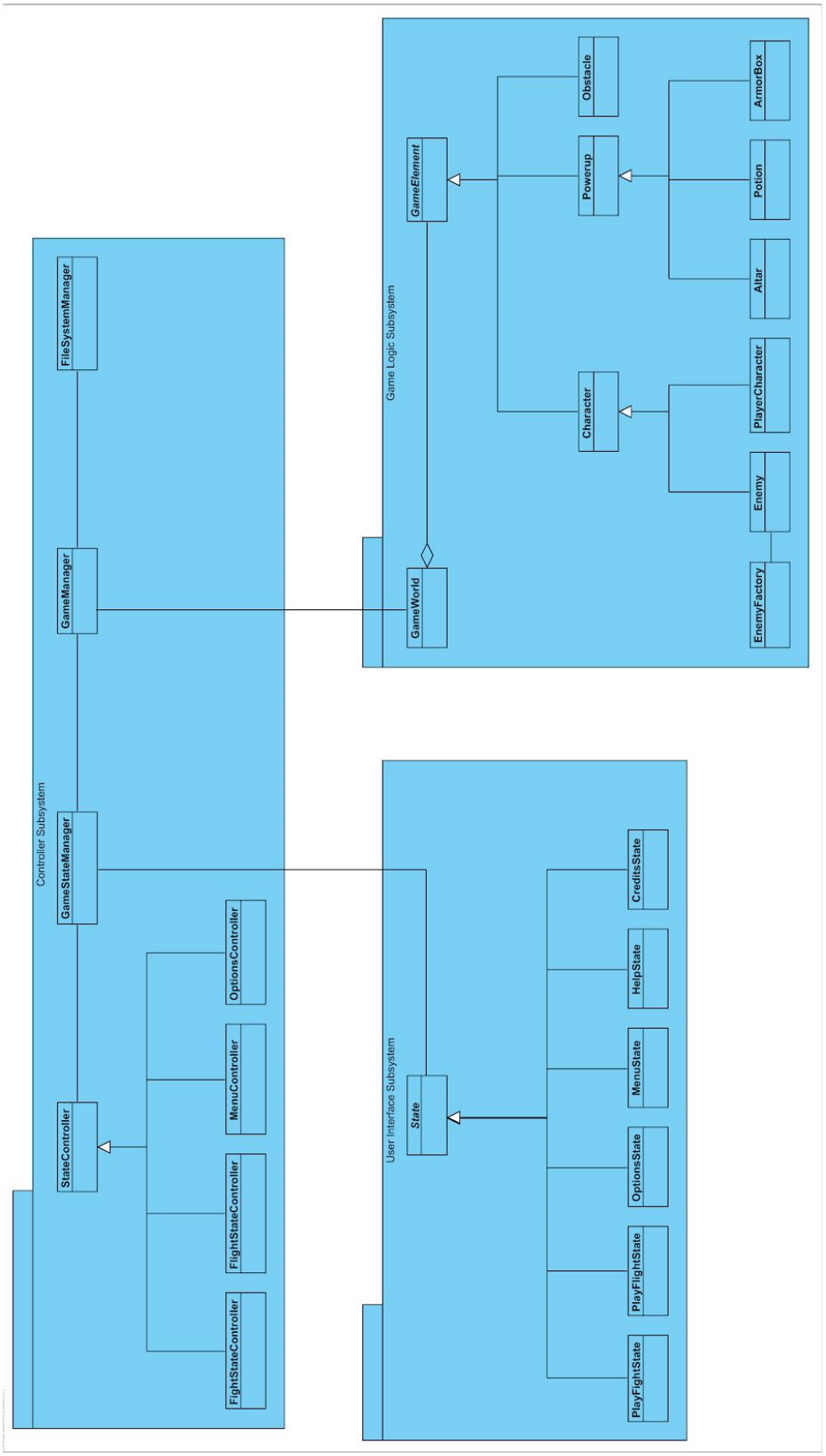


Figure 2.2.1 The system is divided into three subsystems

2.3. Hardware/software mapping

Software configuration: The game will be implemented in Java. We will use libGDX game development framework because it provides useful helper classes for input handling, asset managing and user interface.

Hardware configuration: Players will select menu options by left clicking on them. The character's movement and shooting is controlled with the keyboard. Therefore a keyboard and a mouse is needed as hardware components. A computer with a Java SE Runtime Environment 8 is also needed to run the game. The only data to be kept are the high scores. Therefore a database system is not needed. Local text files are going to be used.

Since the system works on a single computer, the deployment diagram consist of one node. Three subsystems are located in this node as follows.

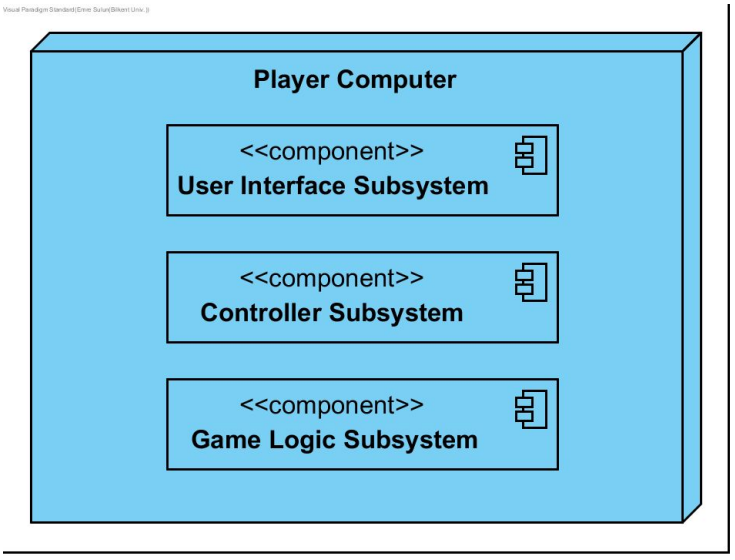


Figure 2.3.1 Deployment diagram

2.4. Persistent data management

The software will offer a simple save/load system along with a highscore table. Save/Load system will allow users to save their progress in a game instance while high score table will record the highest scores attained through all game instances. This means our software will keep two different types of data, though both can be kept in simple *.txt* files.

The software will employ a filesystem instead of a database. Main reason for this decision is because any saved files will be kept and modified only by the local copy of the software itself. In such a case filesystem provides a more efficient answer than a database as it allows for faster read and write options.

2.5. Access control and security

The game does not require users to login. Any user who installed the game can play and see the highscores. Therefore, the design does not include access control, authentication, and encryption functions.

2.6. Global control flow

The control flow mechanism of the game is event-driven control. In the main method, a Game object is created which has a loop, it constantly executes the update method for expecting external events and updating the model objects according to the data that is provided and the other loop constantly executes the render method which draws the objects.

Event-driven control is preferred over procedure-driven control because event-driven control is more suitable and introduces less difficulties for an object oriented design.

2.7. Boundary conditions

- **Start-up and shutdown:** Game starts when user double clicks the game icon. To exit game, user can press *Esc*, then click *Quit* button or press *Alt+F4*.
- **Exception handling:** If any software exception or hardware failure happen, game should store high score without loss. To achieve this goal, high score should not be saved at the end of game, but continuously be saved while playing.
- **Data corruption:** There are two ways this condition can happen. Either through the corruption of save files or corruption highscore file. In both cases, software should give appropriate error messages before deleting the data file in question. In case of highscore file, it should also create a new highscore file.

3. Low level design

Low level design consists of detailed descriptions of all classes in the subsystems, also their member variables, methods and associations between them.

3.1. Game logic subsystem

Game logic is the “Model” in MVC design. It contains the entity objects of the system. GameElement is the abstract class of all elements. We use **Facade pattern** on GameElement. It hides the complexities of the different types of elements and provides an interface to be used by GameManager.

There are different kinds of enemies and all of them have different creation logics. We use **Factory pattern** to hide that complexity and to provide a common interface.

Although all enemies have an ability to attack, they can attack in different ways. For example, some of them attack from a closer distance while some attack from long distance by firing HolyLight. We use **Strategy pattern** to support all those kinds of attacks.

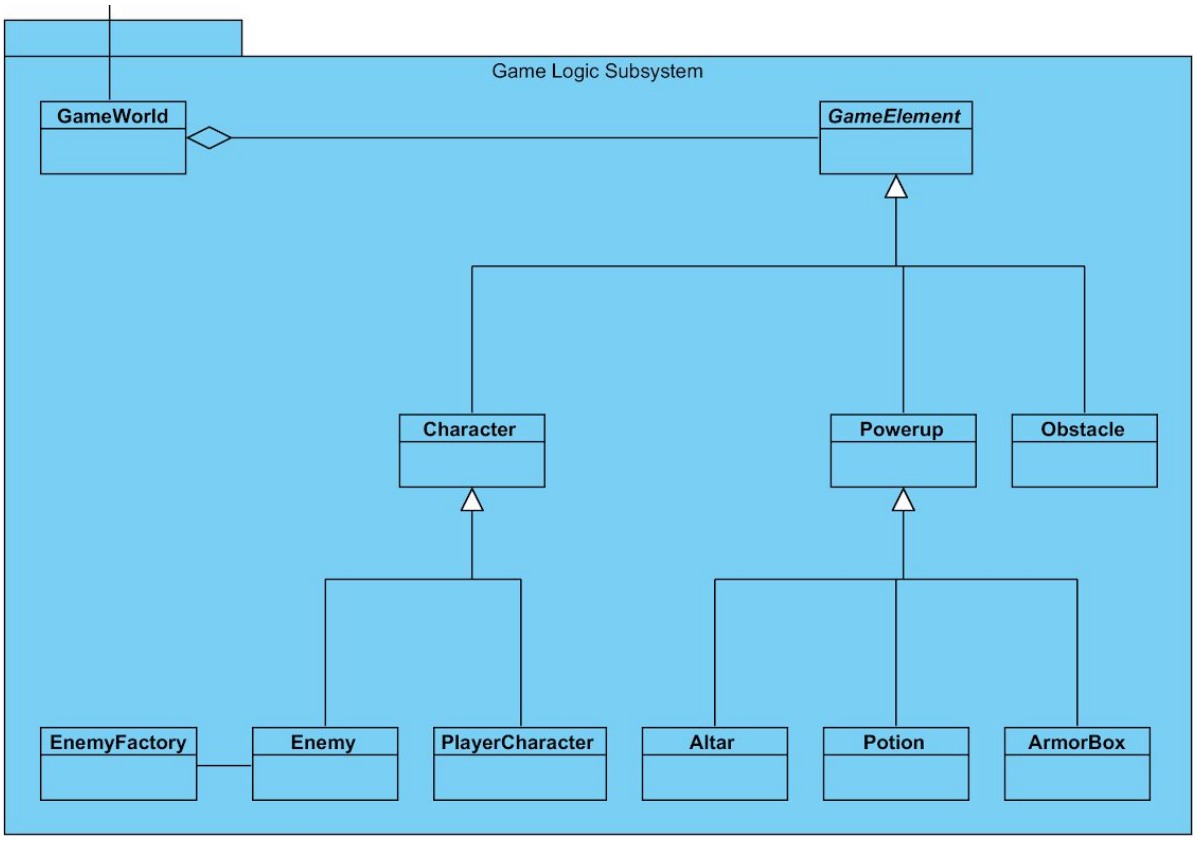


Figure 3.1.1 Game Logic Subsystem

3.2. Controller subsystem

Controller classes are responsible for taking input from user and updates model objects accordingly. GameStateManager, GameManager and FileSystemManager are designed according to **Singleton pattern** because only one instances of those classes should be used to prevent confusions.

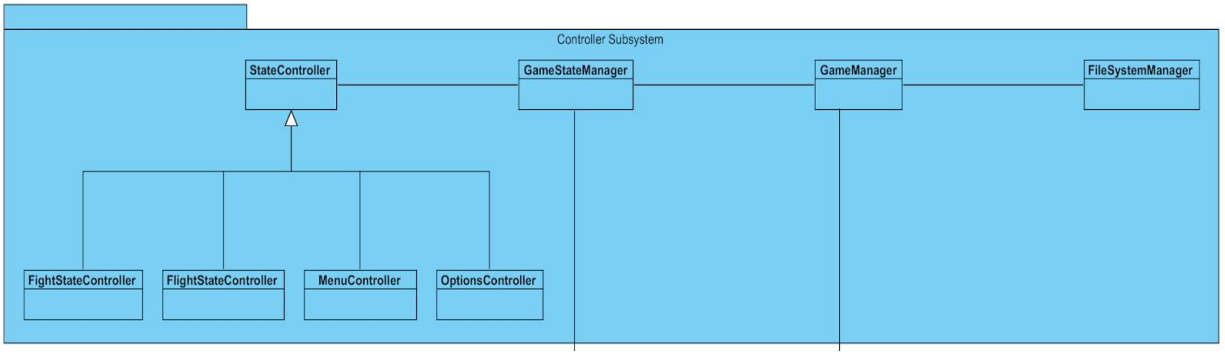


Figure 3.2.1 Controller subsystem

3.3. User interface subsystem

This is the view part of MVC design. We design screen changes according to **State pattern.** All screens implement an abstract class called State and states are stored in a stack. States transition to other states depending on events or inputs.

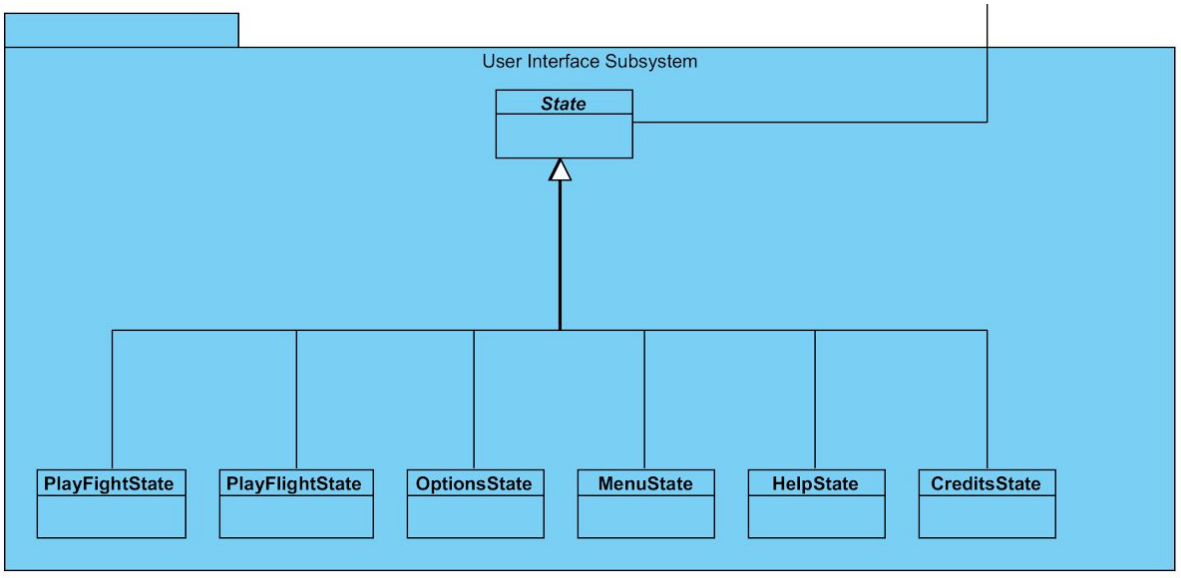


Figure 3.3.1 User Interface Subsystem

3.4. Final class diagram

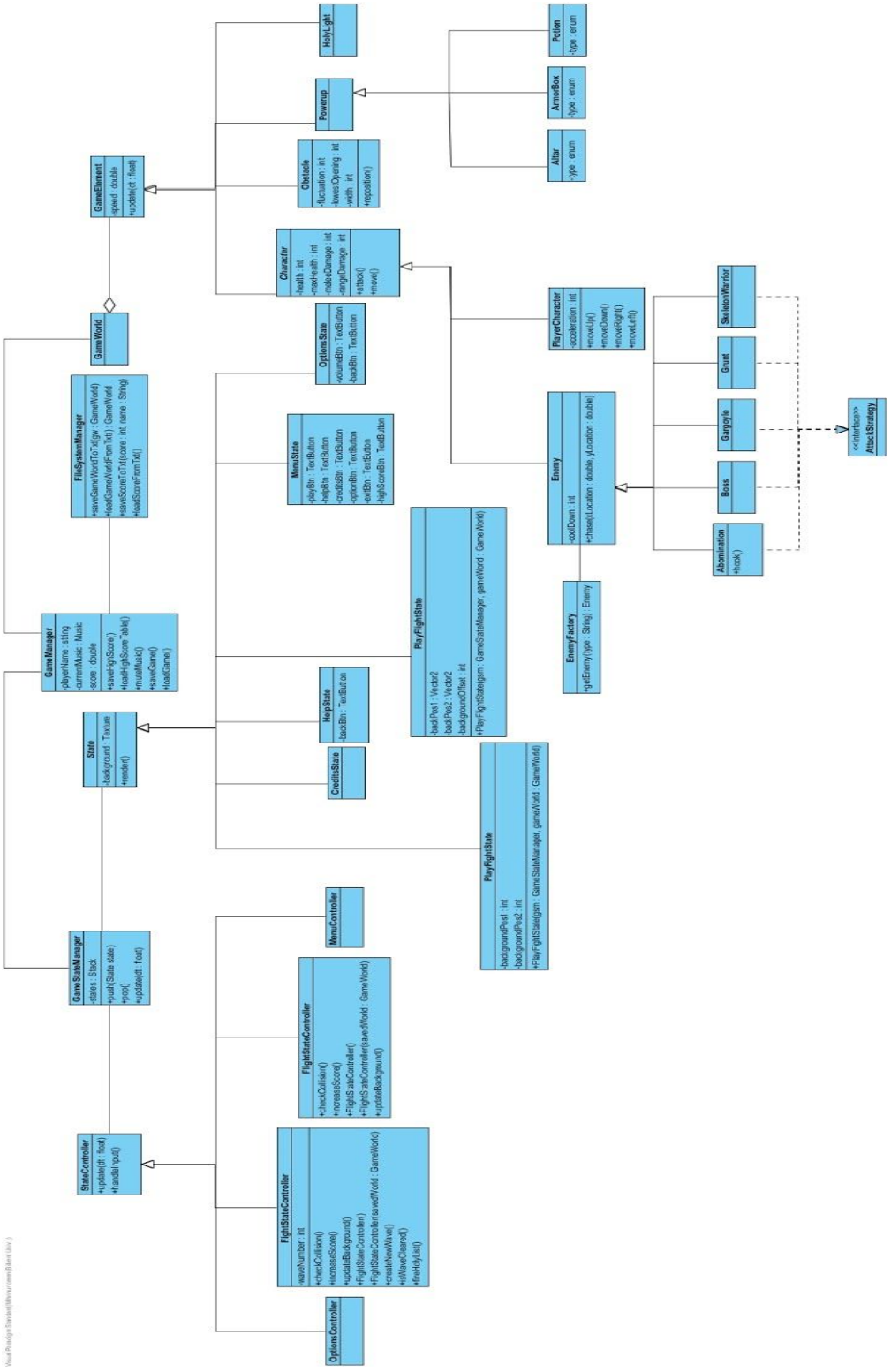


Figure 3.4.1 Final Class Diagram *(closer diagrams are given below)*

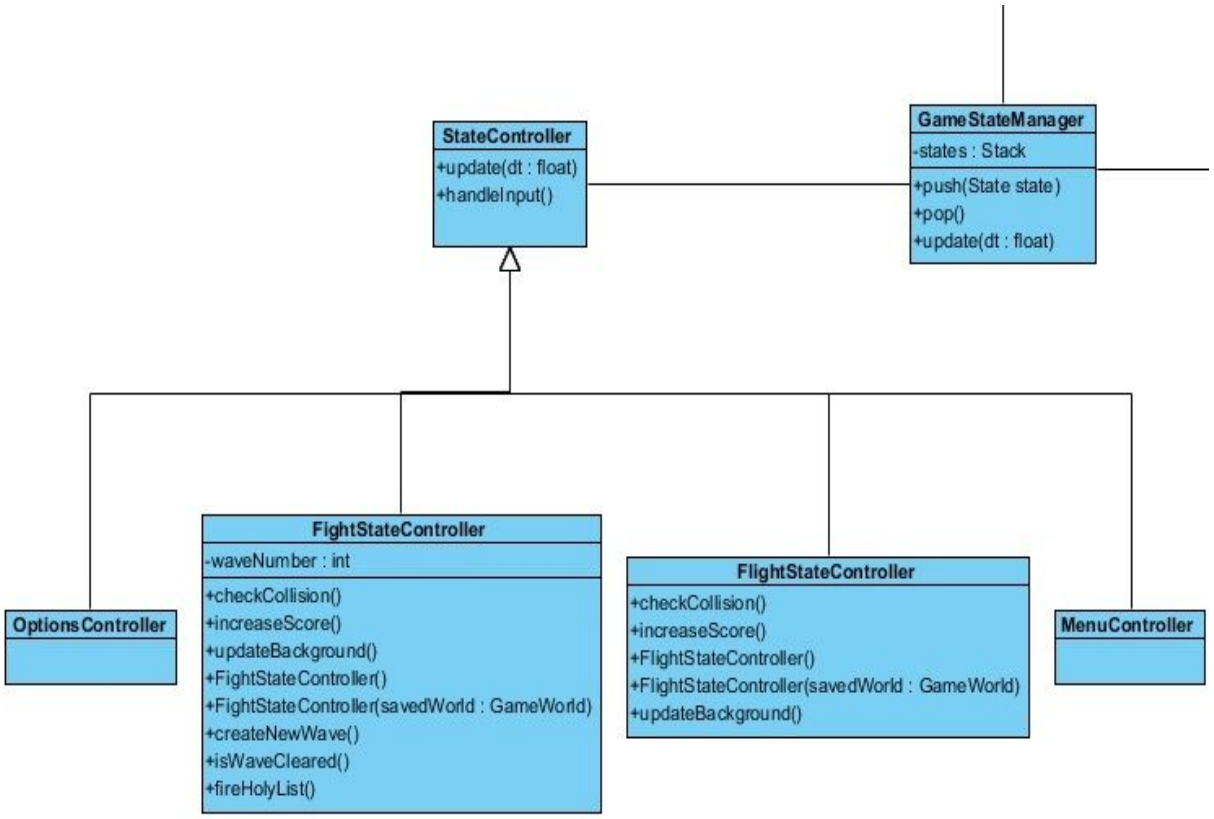


Figure 3.4.2 Final Class Diagram (part 1)

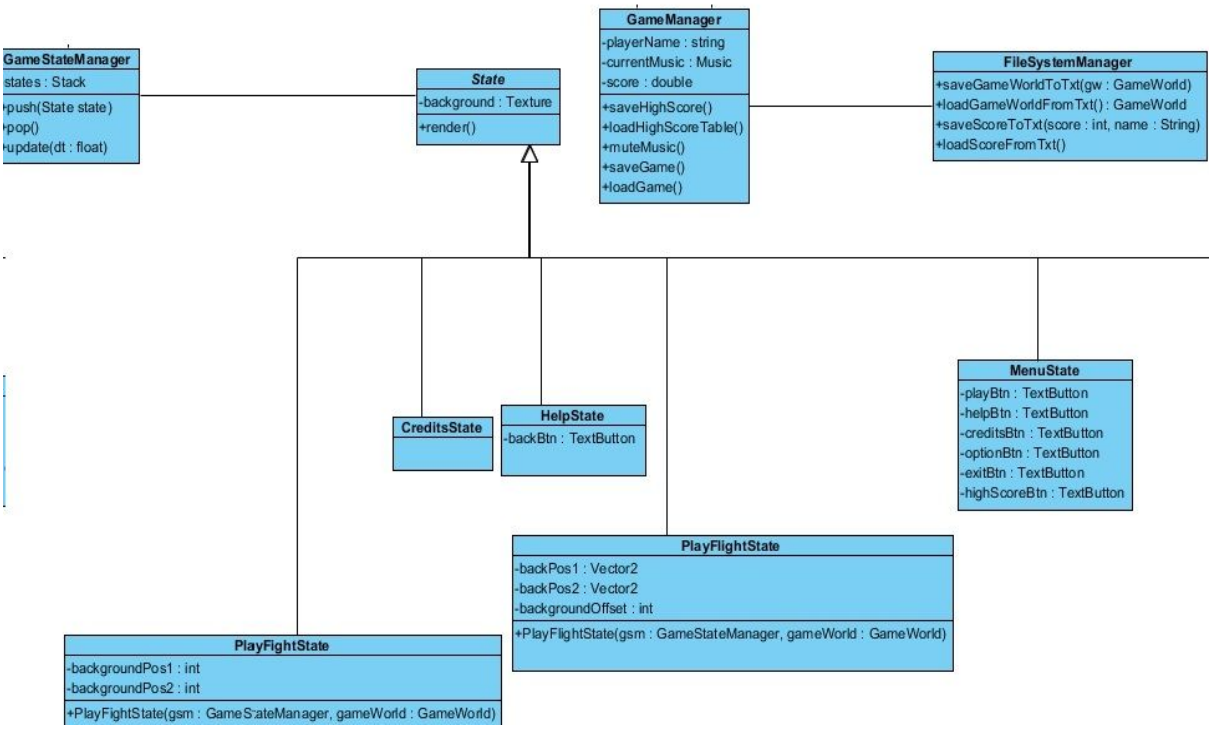


Figure 3.4.3 Final Class Diagram (part 2)

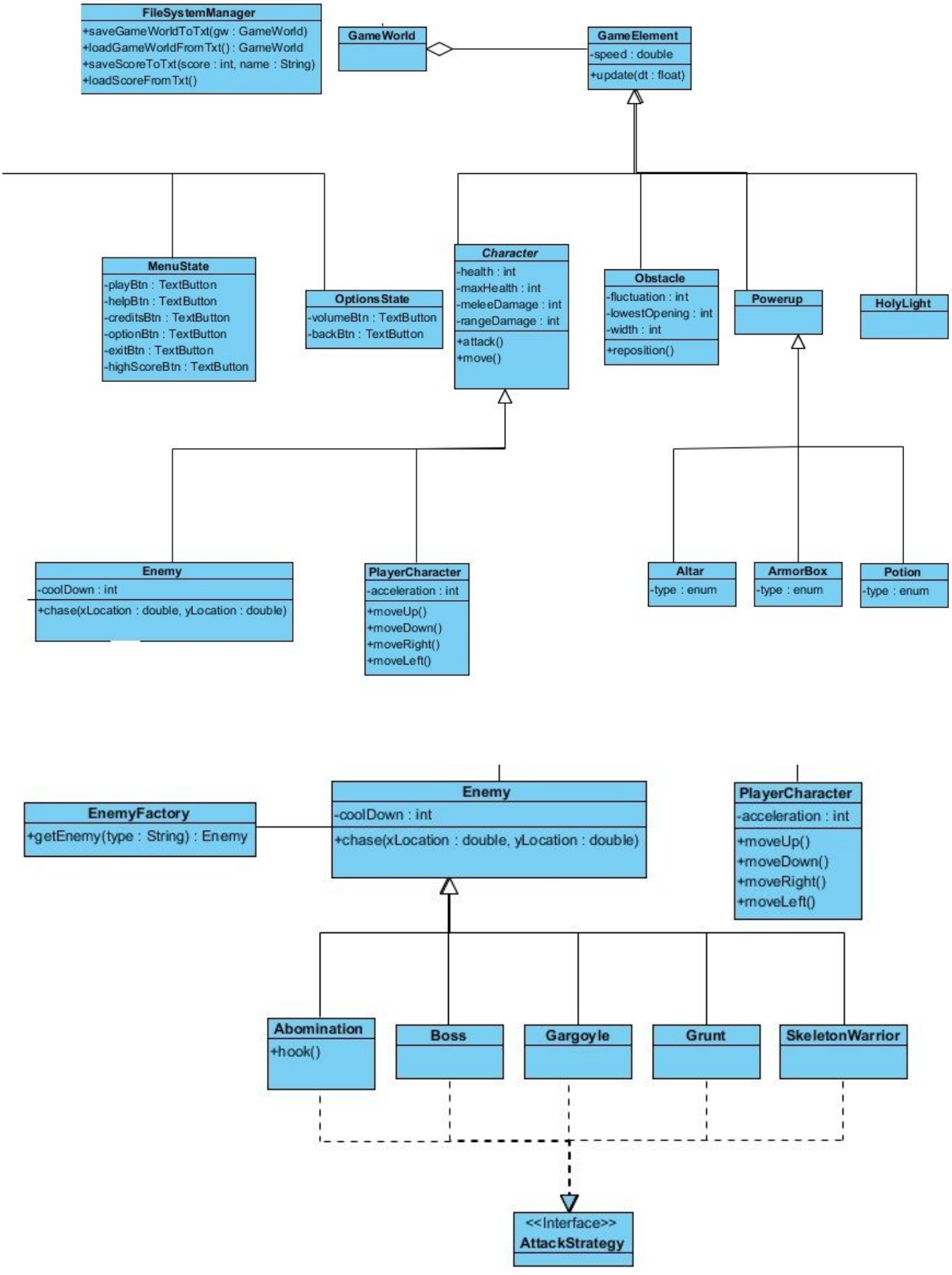
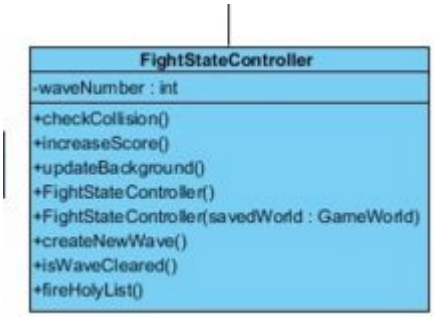


Figure 3.4.4 Final Class Diagram (part 3)



-

Private int waveNumber: A new wave of enemies is created. Number of enemies increase as waveNumber increases.

Public void checkCollision(): This method is called from the update loop and if a collision is detected between game elements. Gameworld is updated accordingly.

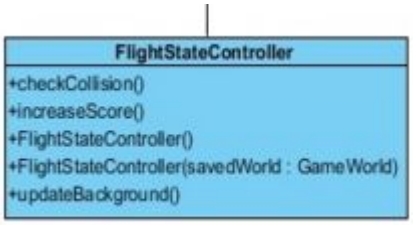
Public void increaseScore(): Score is increased with each enemy being killed.

FightStateController(savedWorld : GameWorld): An additional constructor in case a game is being loaded from a saved instance.

Public void createNewWave(): If isWaveCleared() returns true, this method is executed to create new enemies inside the GameWorld.

Public boolean isWaveCleared(): This method is constantly executed inside the update loop to determine whether all enemies are dead or not.

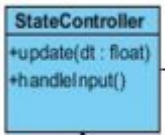
Public void fireHolyLight(): Creates a new HolyLight object inside the GameWorld.



Public void checkCollision(): This method is called from the update loop and if a collision is detected between game elements. Gameworld is updated accordingly.
Public void increaseScore(): Score is determined with distance travelled multiplied by the player character's acceleration.

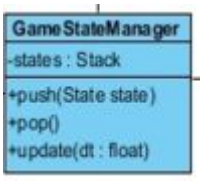
FlightStateController(savedWorld : GameWorld): An additional constructor in case a game is being loaded from a saved instance.

Public void updateBackground(): There are two background images. These two images are constantly being repositioned by this method(their positions are being changed) to create the illusion of a moving background.



Public void update(float dt): Constantly updates the model objects.

Public void handleInput(): This method is constantly being executed inside the update method to handle user inputs such as keyboard presses and mouse clicks and updates the model objects according to the input.



States: A stack holding the current view object(current state) of the game to be rendered.

Public void push(State state): This method pushes a new state on top of the stack. The transition between screens such as Main Menu and Credits screen is handled by this method.

Public void pop(State state): Pops the current state from the top of the stack.

Public void pop(State state): Pops the current state from the top of the stack.

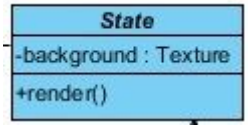
Public void update(float dt): Calls the render method of the state which on the top of the stack.

Public boolean saveGameWorldtoTxt(GameWorld gw): Takes the model object as parameter which is the GameWorld and saves the objects current values inside a text file to be loaded later.

Public boolean loadGameWorldfromTxt(GameWorld gw): Takes the model object as parameter which is the GameWorld and sets the objects current values to new values which are taken from a text file.

Public boolean saveScoretoTxt(int score, String name): Saves the current score to a txt file.

Public boolean loadScoreFromTxt(): Read the score from a txt file.



Attributes:

Public Texture background: Background of the frame.

Methods:

Public void render: This method draws game elements on the screen.

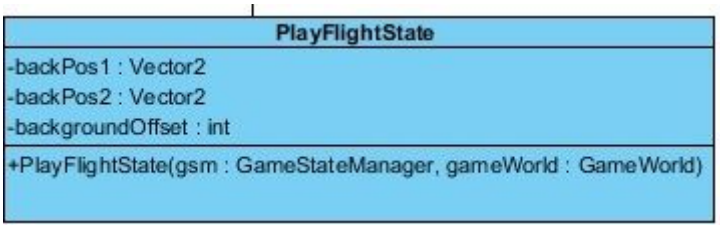
Attributes:

Public int backgroundPos1:We have two background images. In order to relocate, we use two position to combine them.

Public int backgroundPos2:

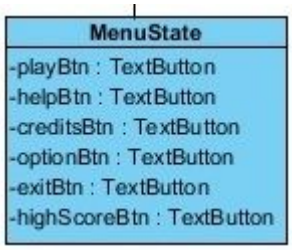
Methods:

PlayFightState(GameStateManager gsm, GameWorld gameWorld): This constructor is used for load operation.



Attributes:

Vector2 backpos1: We have two background images. In order to reposite, we use two positions to combine them.



Attributes:

Public TextButton playBtn: Button to initiate play screen.

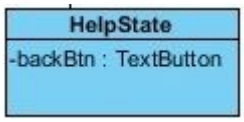
Public TextButton helpBtn: Button to initiate help screen.

Public TextButton creditsBtn: Button to initiate credits screen.

Public TextButton optionBtn: Button to initiate option screen.

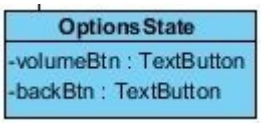
Public TextButton exitBtn: Button to exit.

Public TextButton highScoreBtn: Button to initiate high scores screen.



Attributes:

Public TextButton backBtn: Button to go back.

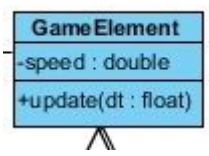


Attributes:

Public TextButton volumeBtn: Button to adjust the volume.

Public TextButton backBtn: Button to go back.

Game Element



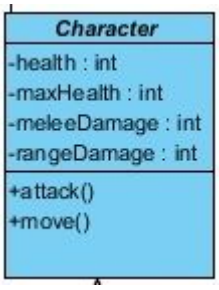
Game Element is the parent class for all dynamic objects that are part of a game instance like enemies, characters, obstacles, projectiles and powerups.

Attributes:

Speed: Speed attribute defines how fast an object can move.

Methods:

Update: Update method is required to reflect the changes that occur to a gameElement like location change.



Attributes:

health: Int value defining the current hit points a single character has. If this value reaches zero then the objects gets destroyed.

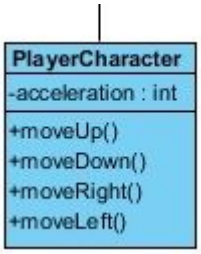
maxHealth: Maximum value of hitpoints a character can have. Can only be increased through upgrades.

meleeDamage: How many hitpoints a character can remove from other characters through a melee attack.

rangedDamage: How many hitpoints a character can remove from other characters through a ranged attack.

move(): This method is executed by gameManager to move a character.

attack(): This method is executed when a character attacks.



PlayerCharacter is the GameElement that the player is able to control. The player can control the players movement and attack.

Attributes:

acceleration: the value that defines how fast, character's speed increases in flights tage.

Methods:

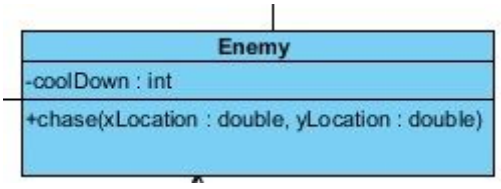
moveUp: This method is executed when player presses W button to move his character towards the top edge of the map.

moveDown: This method is executed when player presses S button to move his character towards the bottom edge of the map.

moveRight: This method is executed when player presses D button to move his character towards the right edge of the map. Only functional in fight stage.

moveLeft: This method is executed when player presses D button to move his character towards the left edge of the map. Only functional in fight stage.

Enemy



coolDown: Attack method belonging to enemy characters take coolDown as an input, only launching a successful attack if coolDown value is zero. On a successful attack this value is

set to a positive integer after which attack method calls upon another function to reduce it over time.

chase(x: double, y: double): This method takes PlayerCharacters x and y as parameter and moves towards that location, chasing the player.

Powerups

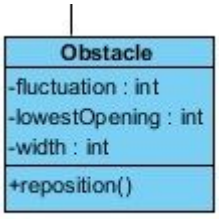
Powerups are immobile gameElements that spawn randomly throughout the map. They provide various bonuses to player when activated.

Altar

When Game manager detects a collision between playerCharacter and Altar, playerCharacters setHealth() and setMana() functions are invoked, taking maxHealth and maxMana as their parameters, and the player reaches max health and mana according to the type of the Altar. Paladins can not use Death Knight altars. Death Knights can not use Paladin Altars.

Type: Determines whether an altar is a paladin Altar or a death Knight altar.

Obstacle



Obstacles are game elements that appear in flight mode. Their main function is to hinder player in the flight mode through variety of means, simplest of which is block his/her path since crashing into an obstacle is an automatic gameover.

Attributes:

Fluctuation: Defines the distance between two obstacles.

lowestOpening: The minimum distance the obstacle can spawn in.

width: How wide the obstacle is.

Methods:

reposition(): Changes the location of an obstacle when executed.

3.5. Object design trade-offs

Creating a completely perfect system is not possible therefore there are various trade-offs.

Development Time vs. User Experience: As a specification of this project, we need to follow deadlines. With time on such short supply we are not able to implement many of the features we have thought of and are very limited in the resources we can use. With more time, we could have implemented a weapon system with each weapon having different game mechanics, a better gear upgrade system, more enemies with special abilities, boss battles that require use of different game mechanics and etc. We could have also produced our own graphical resources like hand drawn sprites and backgrounds if we had more time to spend.

Complexity vs Reliability: Because our first aim is to get a working product, we moved away from many riskier and more complex design choices, further limiting our game in scope.

Efficiency vs. Portability: Portability took a backseat to a more efficient design in order to provide users with a smoother experience. But since we aimed to create a desktop game from the very start, we can afford to make this trade.