BILKENT UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

CS 319 - Object Oriented Software Engineering

Term Project Iteration 1

FIGHT OR FLIGHT

System Design Report

03/03/2018

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

- **Extendibility:** Making updates is a crucial part of the project. New functions, new classes and some new features are

required to increase the quality of game. Due to the system having an object oriented design, adding new classes and modifying existing classes will not disrupt the existing version of the program.

- **Portability:** The game will be implemented in Java.
- **Good documentation:** A well-documented code makes maintenance much easier. Therefore, every class should be documented using Javadoc.
- **Reliability:** Every possible scenario of the game should be tested and debugged, then verified that no stage of the game or the menus cause a crash or a bug. In the case of hardware failures, high scores data should not be lost.

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

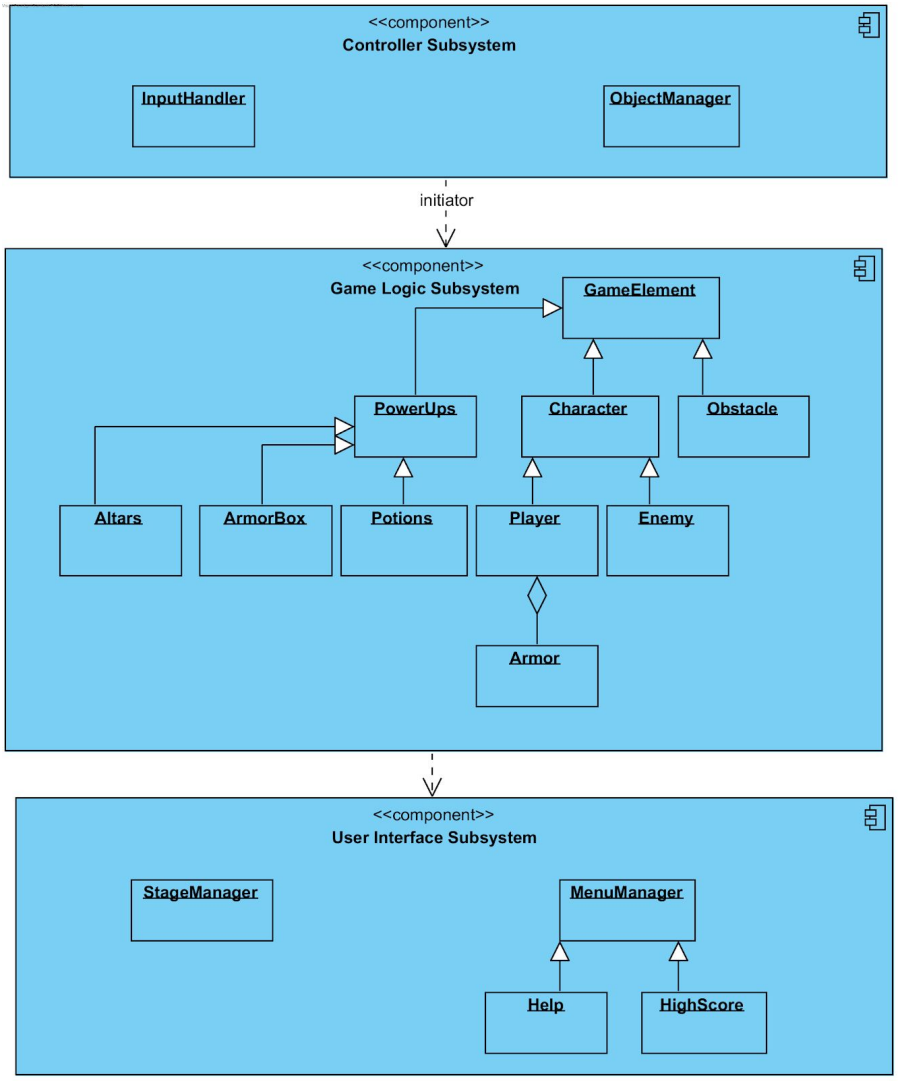
- Bruegge, B, & Dutoit, A 2014, Object-Oriented Software Engineering : Using UML, Patterns, And Java, n.p.: Boston : Prentice Hall, 2014., Bilkent University Library Catalog (BULC), EBSCOhost, viewed 16 February 2018.

2. Software architecture

Software architecture is the top level design of the game. It includes subsystems, design patterns, hardware/software mapping, persistent data management and access control.

2.1. Subsystem decomposition

To reduce complexity and distributing tasks among team members, we divided our system to three subsystems: game logic, controller and user interface. This decomposition is inspired from well-known Model View Controller design pattern.



2.2. Design Patterns

Our system consists of three subsystems similar to MVC design pattern. Classes in Game logic subsystem are responsible for game objects regardless of interface. User interface classes control the interface and render the screen according to model. Controller classes establish connection between model and view.

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.

2.4. Persistent data management

The only data of the game is high scores table. Since this is a very simple table, using a database is redundant and increases the size of application. We will use a single *.txt* file which contains the user names and their highscores.

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. Boundary conditions

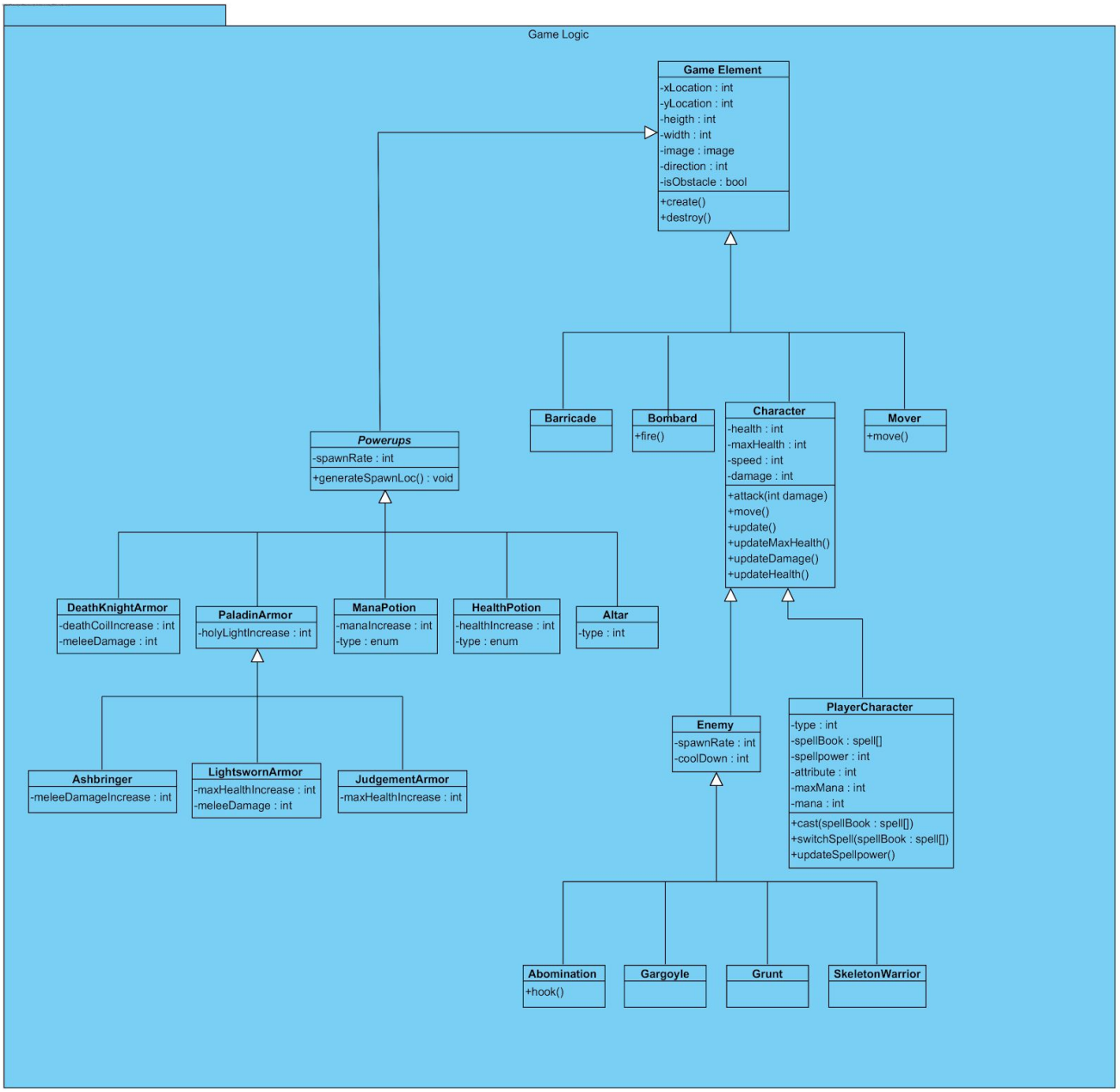
- **Start-up and shutdown:** Game starts when user double clicks the game icon. To exit game, user can press *Esc*, then click *Quit* button or press *Alt+F4*.
- **Exception handling:** If any software exception or hardware failure happen, game should store high score without loss. To achieve this goal, high score should not be saved at the end of game, but continuously be saved while playing.

3. Low level design

Low level design consists of all classes, their member variable, methods and associations between them.

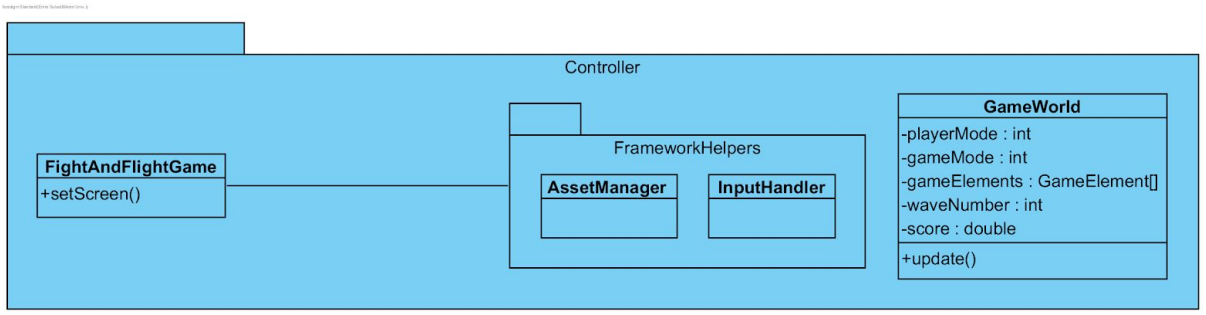
3.1. Game logic subsystem

Game logic is the “Model” in our Model View Controller design. It contains the entity objects of the system. GameElement is the abstract class of the system. It contains attributes like coordinates, width, height, image since all of our game elements have these common attributes. GameElement breaks down to 5 subclasses, 3 for obstacles in the Flight stage, 1 for the moving and fighting characters of the game, which are enemies and the playerCharacter, and 1 for the stationary objects which are the powerups that increase the attributes of the playerCharacter.


Game Logic
Game Element
-xLocation : int
-yLocation : int
-heigth : int
-width : int
-image : image
-direction : int
-isObstacle : bool
+create()
+destroy()
Barricade
Bombard
+fire()
Character
-health : int
-maxHealth : int
-speed : int
-damage : int
+attack(int damage)
+move()
+update()
+updateMaxHealth()
+updateDamage()
+updateHealth()
Mover
+move()
Powerups
-spawnRate : int
+generateSpawnLoc() : void
DeathKnightArmor
-deathCoilIncrease : int
-meleeDamage : int
PaladinArmor
-holyLightIncrease : int
ManaPotion
-manaIncrease : int
-type : enum
HealthPotion
-healthIncrease : int
-type : enum
Altar
-type : int
Ashbringer
-meleeDamageIncrease : int
LightswornArmor
-maxHealthIncrease : int
-meleeDamage : int
JudgementArmor
-maxHealthIncrease : int
Enemy
-spawnRate : int
-coolDown : int
PlayerCharacter
-type : int
-spellBook : spell[]
-spellpower : int
-attribute : int
-maxMana : int
-mana : int
+cast(spellBook : spell[])
+switchSpell(spellBook : spell[])
+updateSpellpower()
Abomination
+hook()
Gargoyle
Grunt
SkeletonWarrior

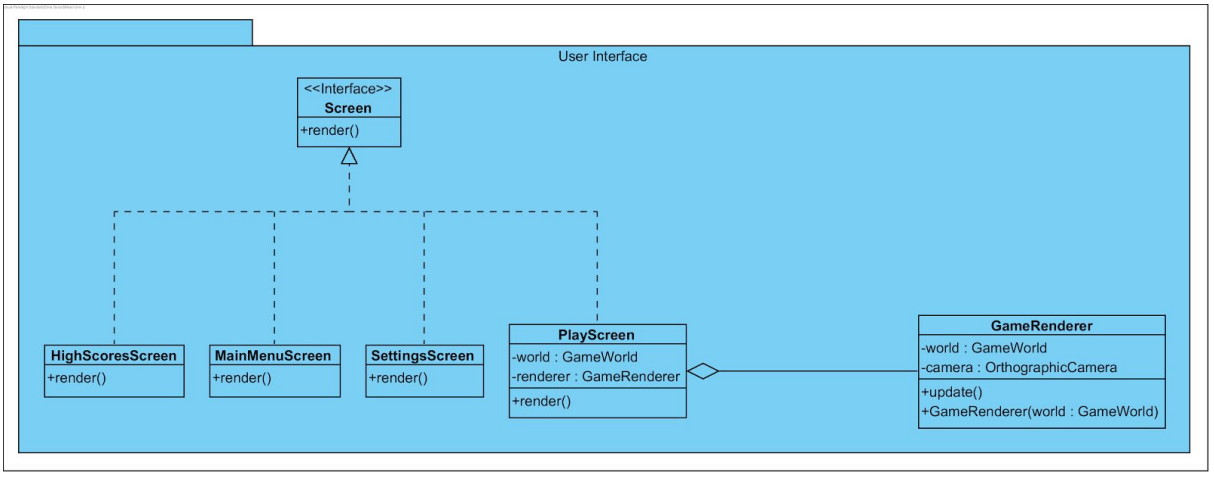
3.2. Controller subsystem

Controller classes take input from user and updates model objects accordingly. Helper classes of the framework are essential for this package.

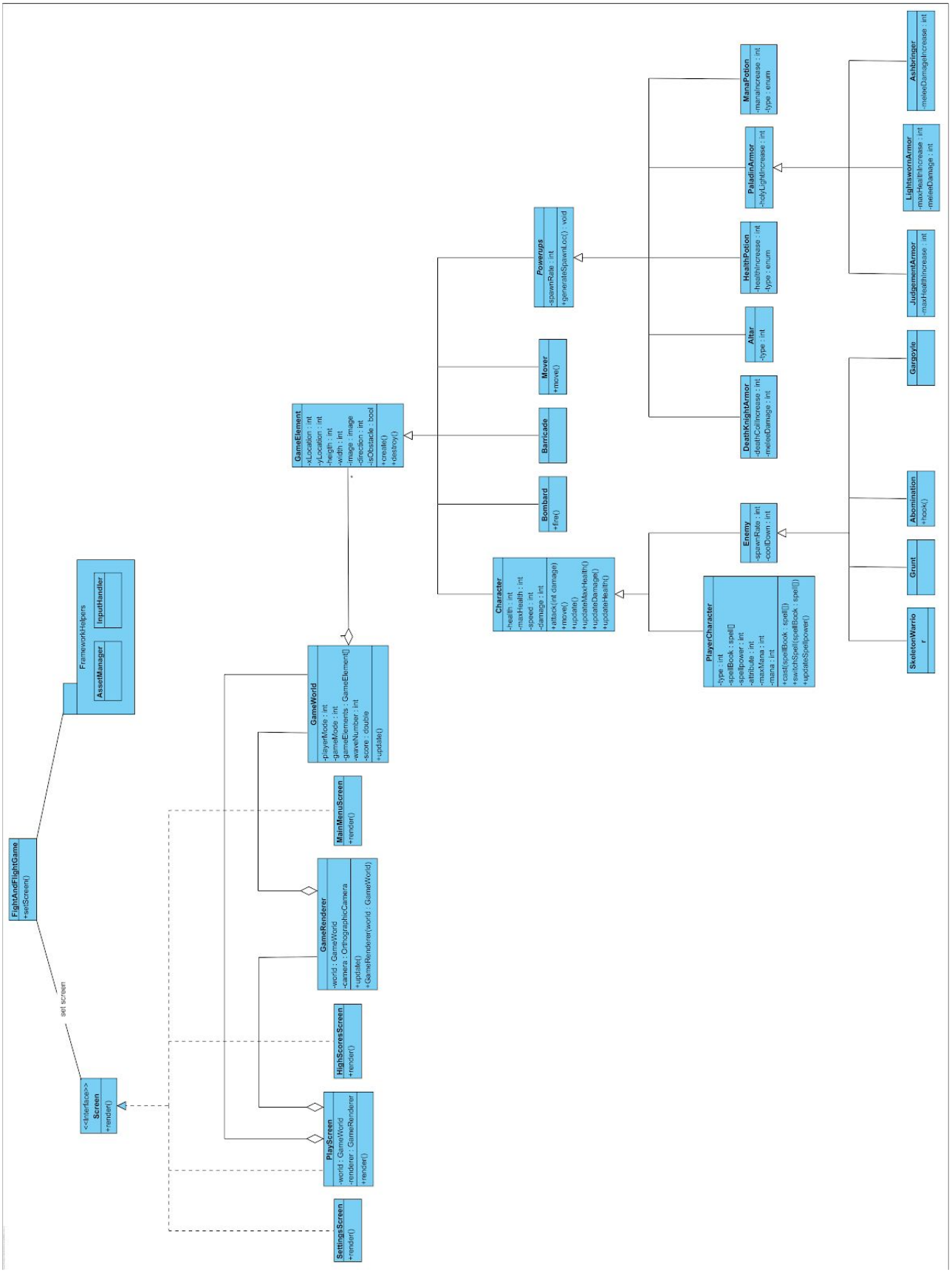


3.3. User interface subsystem

User interface classes are the classes which users interact with. For example, in MainMenuScreen, player can click a button and change screen.

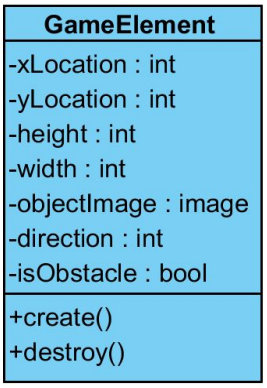


3.4. Final class diagram



Descriptions of each classes are below.

Game Element



GameElement class is the root of all interactive objects inside the scope of a game instance. From enemies to player characters and even obstacles and powerups extends GameElement.

Attributes:

Private int xLocation: x coordinate of an object

Private int yLocation: y coordinate of an object

Private int width: Value defining the horizontal dimension of an object

Private int height: Value defining the vertical dimension of an object

Private Image objectImage: Graphical representation of an object

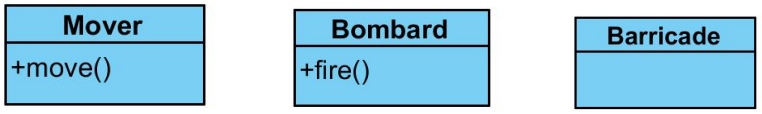
Private int direction: Value defining which direction and object faces. Decides on many factors such as the direction of character sprites and attacks.

Private bool isObstacle: Defines whether an object is an obstacle or not. This decided whether an object can spawn in the running phase or the fighting phase and sets the behaviour of the object.

Public void Create(): Special constructor method created for GameManager use.

Public void Destroy(): Special destructor method created for GameManager use.

Mover-Bombard- Barricade

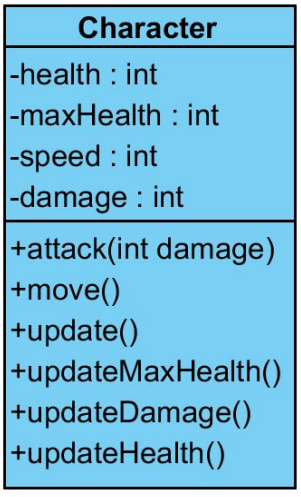


Obstacles are GameElements that only appear during the flight phase of the game. Barricade is the most simple obstacle with no special method of its own. Other two obstacles have their special methods that allow them perform special actions against players during flight phase.

Public void move(): Changes the vertical position of the obstacle.

Public void fire(): Fires a projectile attack.

Character



Attributes:

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

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

Private int speed: Character's move speed. Higher value means character moves faster across the screen.

Private int damage: How many hitpoints a character can remove from other characters when executing an attack. Can be increased through upgrades.

Private void attack(int damage): This method takes the damage integer as input executes an attack.

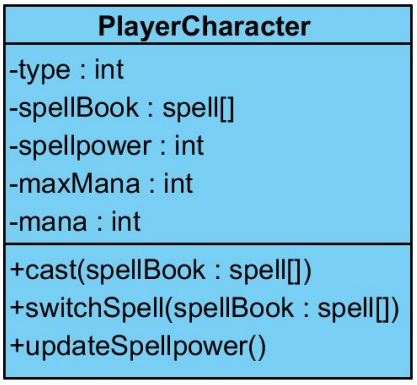
Private void move(): This method is responsible for a character's movement on the screen, depending on the input from player in case of a player character.

Public void updateMaxHealth(maxHealthIncrease: int): This method takes an integer as its parameter and increases the maximum amount of health a character can have.

Public void updateDamage(damageIncrease: int): This method takes an integer as its parameter and increases the amount of damage a character can do.

Public void updateHealth(healthIncrease: int): This method takes an integer as its parameter and increases the current health a character possess.

Player Character



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

Private int type: Type determines which character the player is going to play. It is 1 by default so the character is initially Arathras.

Private spell[ ] spellBook: An array of special abilities character has. The element of the array is the special ability character executes with cast() method.
Private int spellpower: Value defining the effectiveness of a special ability.

Private int maxMana: maxMana determines the maximum amount of mana a PlayerCharacter can have.

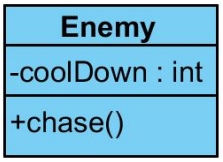
Private int mana: Mana determines the current amount of mana the PlayerCharacter has. It is used for casting spells and it decreases with each used spell. The amount it decreases depends on the type of the spell.

Public void cast(spell[ ] spellBook): Executes the first ability on the spellBook array depending on the spellpower value.

Public void switchSpell(spell[ ] spellBook): Shifts the abilities inside the spellBook array, skipping null values.

Public void updateSpellpower(): Updates the spellpower value.

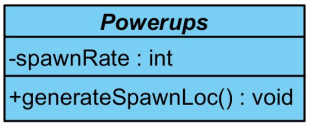
Enemy



Private int 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.

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

Powerups



Powerups are GameElements which have x locations, y locations, heights and widths. They cannot move and they spawn randomly throughout the map. They increase certain attributes of players depending on their types.

Attributes:

Private int spawnRate: Every powerup has a spawn Rate which indicates how often a powerup will spawn on the map.

Public void generateSpawnLoc(): Every powerup would generate its own x and y location. The locations will be generated randomly but the method will have logic and boundaries according to the type of the powerup.

PaladinArmor

PaladinArmor is a box, spawning randomly on the map which carries a paladin armor that changes the appearance and increases the attributes of a playerCharacter only if the playerCharacter type is Paladin.

Attributes:

Private int holyLightIncrease: Every paladin armor increases the ranged damage (Holy Light) of the character.

JudgementArmor

Attributes:

Private int maxHealthIncrease: Every judgement armor increases health by 50.

LightswornArmor

Attributes:

maxHealth: Indicates how much a character's max health will increase.

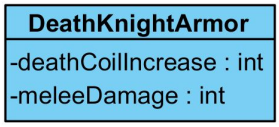
meleeDamage: Indicates how much a character's Melee Damage will increase.

Ashbringer

Attributes:

meleeDamageIncrease: Indicates how much a character's Melee Damage will increase.

DeathKnightArmor



DeathKnightArmor is a box, spawning randomly on the map which carries a death knight armor that changes the appearance and increases the attributes of a playerCharacter only if the playerCharacter type is Death Knight.

Attributes:

deathCoilIncrease: Every death knight armor increases the ranged damage (Death Coil) of the character.

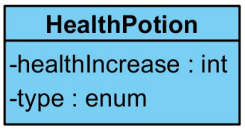
meleeDamageIncrease: Indicates how much a character's Melee Damage will increase.

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.

HealthPotion



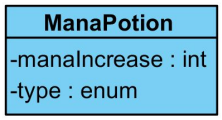
When Game manager detects a collision between HealthPotion and playerCharacter, playerCharacters updateHealth method is invoked, taking HealthPotion objects healthIncrease attribute as its parameter.

Attributes:

type: There are three types of health potions that are ancient health potion, valorous health potion and lesser healing potion. Type determines how much health a HealthPotion would give.

Private int healthIncrease: Every health potion has different healthIncrease.

ManaPotion



When Game manager detects a collision between ManaPotion and playerCharacter, playerCharacters updateMana method is invoked, taking ManaPotion object's ManaIncrease attribute as it's parameter.

Attributes:

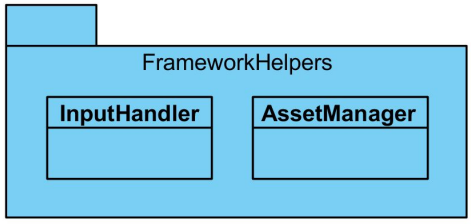
type: There are three types of mana potions.

Private int manaIncrease: Every mana potion has different manaIncrease.

FightAndFlightGame

setScreen(): Changes screen according to user input.

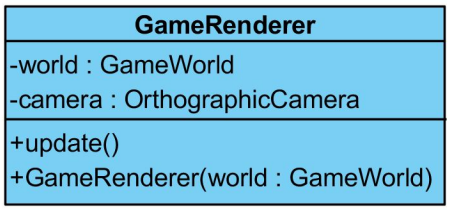
User Interface Subsystem



This package includes classes apart from game logic, these classes are provided from framework.

InputHandler: Controls keyboard and mouse inputs.

AssetManager: Controls images and sounds.

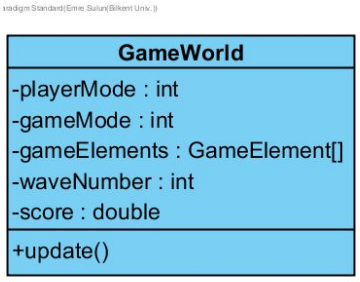


GameRenderer class is responsible for rendering screen according to GameWorld (models).

GameWorld world: The world to be rendered.

OrthographicCamera camera: 2D camera provided from framework

update(): Renders game continuously.



GameWorld class includes all models. Initially starts with the Flight mode. Keeps score of the player, wave number and enemy number in the Fight mode. Detects interaction and collisions between objects, detects attribute changes in objects like coordinates, health etc. and updates them.

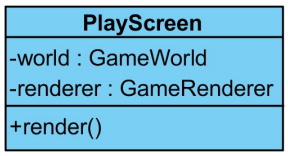
gameMode: If zero, game is at "Flight" mode, else "Fight" mode.

playerMode: If zero, game is single player else multiplayer.

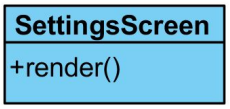
waveNumber: Number of enemy attacks i.e. level of the game

score: Current score of the game

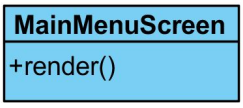
update(): Updates model details like coordinates, health etc.



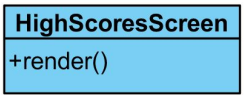
render(): Calls update() methods of GameWorld and GameRenderer classes.



SettingsScreen class is responsible for rendering the settings view.



MainMenuScreen class is responsible for rendering the menu items.



HighScoresScreen class is responsible for rendering the score list.

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.