



CS 319 - Object-Oriented Software Engineering
System Design Report

Sea Adventures

Group 2-G

Alper Kağan Kayalı

Büşra Oğuzoğlu

Kasymbek Tashbaev

Salih Zeki Okur

Contents

1. Introduction	6
1.1 Purpose of the system	6
1.2 Design Goals	7
End User Criteria:	7
Maintenance Criteria:	7
Performance Criteria:	8
Trade offs:	8
Ease Of Use and Ease of Learning vs. Functionality:	8
Performance vs. Memory:	8
1.3 Definitions, acronyms, and abbreviations	9
Abbreviations:	9
1.4. References	9
1.5. Overview	9
2. Software Architecture	9
2.1. Overview	9
2.2. Subsystem Decomposition	10
2.3. Architectural Styles	13
2.3.1 Layers	13
2.3.2 Model View Controller	13
2.4. Hardware / Software Mapping	14
2.5. Persistent Data Management	14
2.6. Access Control and Security	15
2.7. Boundary Conditions	15
3. Subsystem Services	15
3.1. Design Patterns	16
Façade Design Pattern:	16

3.2. User Interface Subsystem Interface:	16
MainMenu Class	17
Constructors:	17
Methods:	18
ScreenManager Class	18
Attributes:	18
Constructor:	18
Methods:	19
Menu class	19
Attributes:	19
Constructor:	20
Methods:	20
PauseMenu class	20
Constructor:	21
Methods:	21
3.3 General Game Management Subsystem Interface	22
GameEngine Class:	23
Attributes:	23
Constructor:	23
Methods:	23
InputManager class	24
SoundEngine Class	25
Attributes:	25
Constructors:	25
Methods:	25
SettingManager Class:	26
Attributes:	26
Constructor:	27

Methods:	27
CollusionManager Class:	27
Attributes:	27
Constructor:	28
Methods:	28
3.4 Game Objects Subsystem Interface	29
Map Class:	31
Attributes:	31
Constructor:	32
Methods:	32
GameObject Class	34
Attributes:	34
Constructor:	34
Methods:	35
Bullet class	36
Attributes:	36
Constructors:	36
PowerUp Class	36
Attributes:	36
Constructors:	37
Methods:	37
Submarine Class	37
Enemy Class:	41
Attributes:	41
Constructor:	42
Methods:	42
Small Enemy Class:	42
Attributes:	42

Constructor:	42
Methods:	42
Big Enemy Class:	42
Attributes:	42
Constructor:	42
Methods:	43
Boss Manager Class:	43
Attributes:	43
Constructor:	43
Methods:	43
ObjectRandomLocationManager Class:	44
Attributes:	44
Constructor:	44
Methods:	44
CooldownManager Class:	45
Attributes:	45
Constructor:	45
Methods:	45
SkillManager Class:	46
Attributes:	46
Constructor:	46
Methods:	46
Health Class:	46
Attributes:	46
Constructor:	47
Methods:	47
Energy Class:	47
Attributes:	47

Constructor:	47
Methods:	47
Skill Class:	48
Attributes:	48
Constructor:	48
Methods:	49
3.5 Detailed System Design	50

1. Introduction

1.1 Purpose of the system

Sea Adventures is a horizontal scrolling shooter game based on the classic game space shooter. The goal of the project is to implement the game using OOP structure using JavaFX. Since it will be horizontal unlike classic space shooter game and using different graphical objects such as the submarine, it will create a different gaming experience for the player. It will also have bosses at the end of each level to make the game more challenging and fun.

1.2 Design Goals

We identified our important design decisions in analysis report therefore, we will continue doing our design based on our non-functional and functional requirements that we specified in our analysis part. Important design goals will be explained in detail in the following section:

End User Criteria:

Easy to use: Sea adventures will be a game that can be played by children and adults. Therefore gaming system should be easy to understand and control. Interface will be user-friendly so it will be easy to navigate through menus and options.

Entertaining: Game should be easy to understand however it should be difficult enough for entertainment. Since it will have different elements compared to classic space shooters game it will increase entertainment.

Maintenance Criteria:

Extendibility: We are going to make our object design by taking important criteria such as sustainability and extendibility into consideration. Therefore our system will be suitable for adding new functionalities such as new enemies, new difficulty levels, new submarines or new power-ups.

Portability: We will implement our project in Java which includes a system called JVM. Since we implement this technology, our system will be portable. Therefore, a lot of users that use different platforms will be able to reach our game.

Reliability: For our system to be reliable it should be bug-free and should not crash easily. In order to reach this outcome, we will do lots of testing simultaneously with the development.

Modifiability: Functionalities in our system should be easy to modify and improve for general sustainability of the system.

Efficiency: We will focus on increasing games speed rather than decreasing the memory usage of the game. We will optimize the memory usage of game, given that high fps for games running speed is guaranteed. Because the running speed of the game is one of the most important things for the user to enjoy the game.

Performance Criteria:

Trade offs:

Ease Of Use and Ease of Learning vs. Functionality:

Since the system we are designing is a shooter game, users should learn how to use the system with ease so that they can spend more time on playing the game. Hence the priority of the design of our system will be ease of use rather than functionality. To ensure that, our design will provide only the necessary features for the user to enjoy a shooter game.

Performance vs. Memory:

The system that we are building needs to consider the performance since it is an interactive application with the user. Of course, it will consume more memory than the systems who are slower than our system. However, the requirements of the game consider performance more than memory. Although there are many improvements in the memory

which are considered being done in the project, the design is concerned about the performance, not the memory.

1.3 Definitions, acronyms, and abbreviations

Abbreviations:

MVC: [2] Model View Controller

JDK: [1] Java Development Kit

JVM: [1] Java Virtual Machine

1.4. References

[1] [http://en.wikipedia.org/wiki/Java_\(programming_language\)](http://en.wikipedia.org/wiki/Java_(programming_language))

[2] Object-Oriented Software Engineering, Using UML, Patterns, and Java, 3rd Edition, *by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2010, ISBN-10: 0136066836.*

1.5. Overview

In this section, we showed the aim of our system, which is basically entertaining the user as much as possible. In addition, we defined our design goals in this section in order to achieve our purpose. Our design goals are defined based on to provide the extendability, portability, reliability, modifiability and efficiency on the maintenance side. On the end-user side, we defined our design goals as easy-to-use and entertaining. In order to achieve our goals, we made some trade-offs. We sacrificed from functionality in order to make the system more simple and more understandable. In addition, we gave up from the memory in order to increase the performance such as smooth animation, smooth effects etc.

2. Software Architecture

2.1. Overview

In this section, we will divide our system into some subsystems. We divide our system because we want to reduce the coupling between different subsystems. However, we want to increase the cohesion of the subsystem components. Furthermore, we want to decompose our system in order to apply MVC(Model-View-Controller) architectural design into our system.

2.2. Subsystem Decomposition

In this section, our system is divided into relatively independent parts in order to show how the system is organized. The decisions we make in the subsystems will automatically affect the features in our system such as extendibility, portability, reliability, modifiability and efficiency. Therefore, the decisions we make are very crucial in order to satisfy non-functional requirements and create a high-quality game.

In figure 1, the system is divided into three subsystems which are called User Interface, Game Management and Game entities subsystems. They are called at specific cases, however, they work as one whole system together. As the Figure-2 shows, Game Management and Game Entities are slackly connected. In addition, there is only one connection between Game Management and User Interface through Game Manager class. Thus, any change in the User Interface or any action will only affect the Game Manager class while other classes will be intact.

The detailed design shows the type of each class. For example, the classes have the name with “Manager” or “Engine” shows that they deal with the controls by sending requests and retrieving responses.

We tried to meet our goals while designing the subsystems. Hence, we aimed to design an efficient system with slackly coupling and high cohesion. This shows that the system will be flexible to the changes.

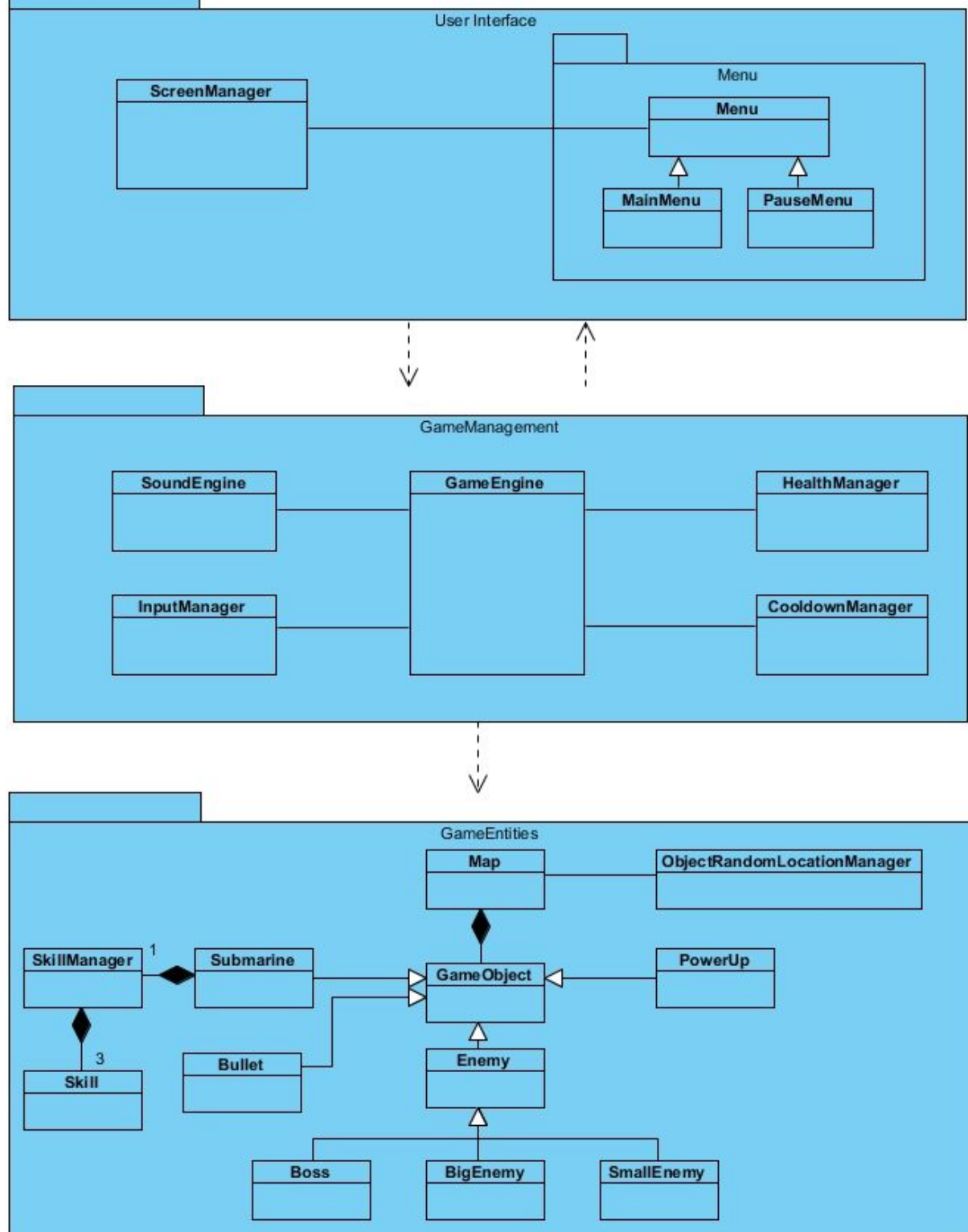


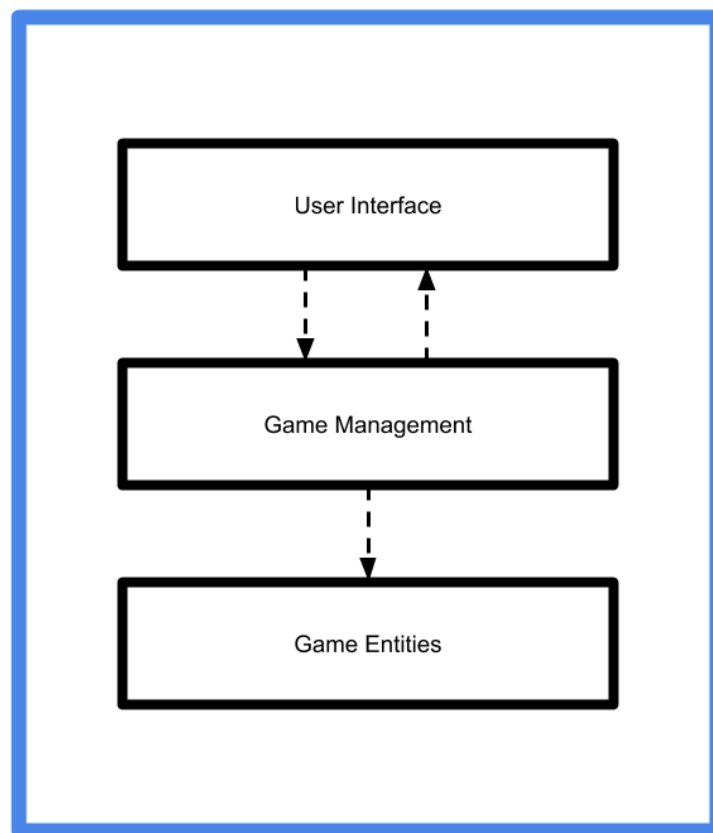
Figure – 1 (Basic Subsystem Decomposition)

2.3. Architectural Styles

2.3.1 Layers

Our game system consists of three different layers. They are generally management of User Interface, the general sequence of the game and game objects. Although they are different layers, user interface, game management and game entities layers work as a whole system.

Game management layer can interact with the user interface through the game engine and screen manager, menu classes. The game engine will also be in interaction with the Map class that is responsible for general management of Game Entities layer.



Apart from that, user interface layer is responsible for showing the interaction to the user and changing the visuals according to feedback as well as reporting user the options and changes through the whole gaming experience. Game management layer is responsible for handling

game's general sequence and mechanics such as cooldown times, inputs, sound effects, managing health and general play of the game.

With the help of game entities layer, actual gameplay will be constructed. Game entities such as enemy and submarine are responsible for interacting other game entities like bullets and power-ups as well as keeping the attributes attack speed, level, bullet damage, energy, experience and others that contain data about gameplay and will be changed through the gameplay by player's interactions with the game.

2.3.2 Model View Controller

Our three different layers User Interface, Game Management and Game Entities are representations of the model view controller architectural style. This style is for dividing the control and model of the game from the view and other specifications about the game. In our design, game entities are mostly the model of the game while user interface classes such as screen manager and menus compose the view part. Other classes which are used for game management such as game engine, sound engine and health manager are responsible for controlling general game mechanics therefore, they compose the controller part of the system.

2.4. Hardware / Software Mapping

Sea Adventures will be implemented using JavaFX therefore, it will be able to operate on any basic computer with a Java Runtime Environment and an operating system. As hardware aspect, a basic keyboard and mouse is needed to interact with the game. Since our high score list in the game will not have many data, we will not use a complex database for it. Image files will be able to access through the game therefore, the game will not need any internet connection either. Input overhead of Sea Adventures will be very small therefore any average computer will be sufficient enough to play the game.

2.5. Persistent Data Management

Since we will not have a lot of data to save and to be accessed while the game will be played by the player, we will not use a complex database system. The game's maps, submarine and enemy's images, sounds and high score list will be stored in suitable formats in the hard disk drive. Therefore, necessary files will be loaded before the game starts running. This gives us the ability to update the visuals and sound effects very efficiently however if files got corrupted, game's user interface will not work properly.

2.6. Access Control and Security

Sea adventures will not have player login system therefore, there will not be any database for storing details for users. For this reason, there will not be any controls or restrictions about accessing the game. Everyone who has the game file can access and play the game. Their data will not be saved as a player so if other player wants to access the game from the same system, the second player should start over or continue where the first player left. On the other hand, sea adventures game will not require any network connection therefore, there will be no security issues.

2.7. Boundary Conditions

Sea Adventures will have the executable .jar file so it does not require installing before entering the game. From main menu, the game can be terminated whenever wanted by pressing the related button. While in the game if player loses all health, game will be over and player will return to the main menu, can start the game again from beginning. If player defeats the final boss, game will be over and player can return to the main menu and check if the score is in high scores list.

3. Subsystem Services

The detailed information about the interfaces of our subsystems will be given in this section.

3.1. Design Patterns

Façade Design Pattern:

Since our system is divided into some subsystems and they communicate with each other, we need to develop a façade class. What this design does is it helps a subsystem to communicate with other subsystems while we still can manage its control. In addition, this design pattern helps us with extensibility, maintainability and reusability because we can change the specific subsystem through this façade class.

In our design, we created façade pattern for all three subsystems. In User Interface subsystem we use ScreenManager class as our façade class which communicates with the other components in User Interface subsystem based on the request of General Game Management subsystem. In General Game Management we use GameEngine as our façade class which communicates with the other components in General Game Management subsystem based on the request of both User Interface and Game Objects subsystems. The same rule is applied in Game Entities subsystem as Map being façade class and the subsystem being communicated with Game Management subsystem.

3.2. User Interface Subsystem Interface:

User Interface subsystem is very helpful in order to create our graphical components. In addition, it helps us to manage the transition between panels which are constructed based on

the selection option in the main menu. The MainMenu class is the interface we use in order to refer User Interface subsystem since it is the first class interacts with the user.

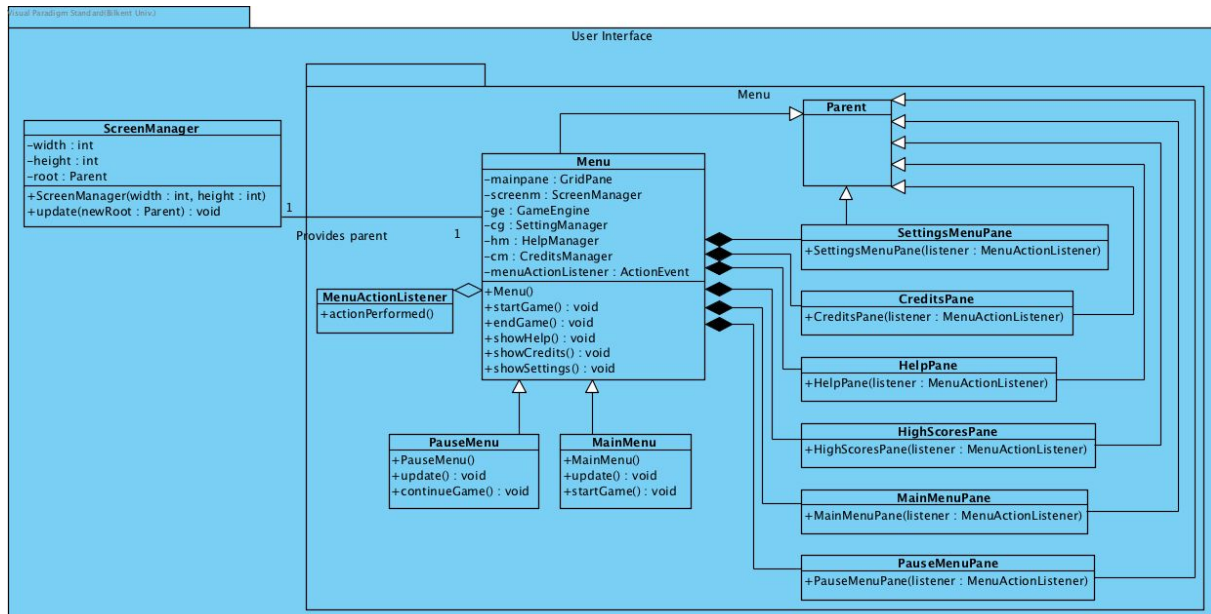
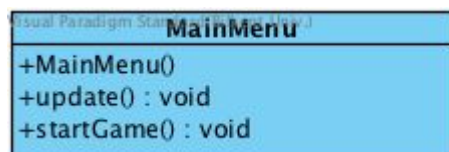


Figure-1(User Interface Subsystem)

MainMenu Class



Constructors:

public MainMenu(): Initializes mainpane, screen manager, menu action listener, game engine and setting, help, and credits managers.

Methods:

public void update(): This method will help us with updating the current situation and current pane. If any other option on the main menu is selected, it will update the game engine and main pane accordingly, nothing will change otherwise.

public void startGame(): This method will start the game when the player wants to start playing the game which will override the startGame() method of the Menu class.

ScreenManager Class

ScreenManager
<div><div>-width : int</div><div>-height : int</div><div>-root : Parent</div></div>
<div><div>+ScreenManager(width : int, height : int)</div><div>+update(newRoot : Parent) : void</div></div>

Attributes:

private int width: The value set for the width of the main screen which is called root.

private int height: The value set for the height of the main screen which is called root.

private Parent root: This is the main frame that will be used for showing all visual context of the program

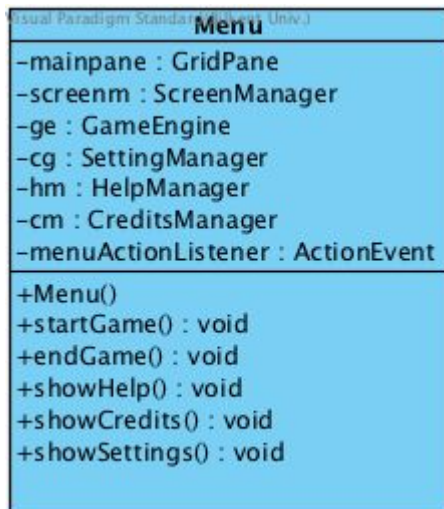
Constructor:

ScreenManager(int width, int height): Takes width and height value in order to construct the screen.

Methods:

public void update(Parent newRoot): This method will update the screen manager with a new screen if there is a change in the screen.

Menu class



Attributes:

private GridPane mainpane: This is the pane that we display all visual content of the game whether it is on the main menu or change setting or some other section.

private ScreenManager screenm: this object will help us to change the content on the screen. It will be correlated with mainpane.

private GameEngine ge: This object will help us to deal with the main mechanics of the game such as the location of the objects, the collisions etc.

private SettingManager cg: This module will help us to get the current settings, modify the settings and display the current settings.

private HelpManager hm: This module will help us to get the help menu and display it.

private CreditsManager cm: This module will help us to get the credits menu and display it.

private ActionEvent MenuActionListener: This object will help us what to do when there is a user input such as pressing a key or pressing mouse etc.

Constructor:

Menu(): This will create an instance of the Menu.

Methods:

public void startGame(): This is the method that starts the application.

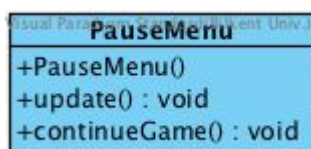
public void endGame(): This is the method that ends the application.

public void showHelp(): This method will work with update method in order to show the help section

public void showCredits(): This method will work with update method in order to show the credits section.

public void showSettings(): This method will work with update method in order to show the settings section

PauseMenu class



Constructor:

PauseMenu(): Initializes the PauseMenu object.

Methods:

public void update(): This method will update the current situation and current pane just like MainMenu. However, in here the player won't be able to quit the game unlike MainMenu.

public void continueGame(): This method will work exactly same as the startGame() method in the MainMenu class. However, in continueGame() the current condition of the player in the game is saved and the player will continue on the save position unlike startGame() method. In addition, this will override the startGame() method of the Menu class.

3.3 General Game Management Subsystem Interface

In this interface, we designed the components which will be helpful for our game management. The subsystem will consist of 5 components. The InputManager, SoundEngine, and SettingManager will directly depend on the user, which means that they can be changed directly by the player. However, the CollisionManager and GameEngine components will be system components and will act with each other. the interface will be explained in the section given below.

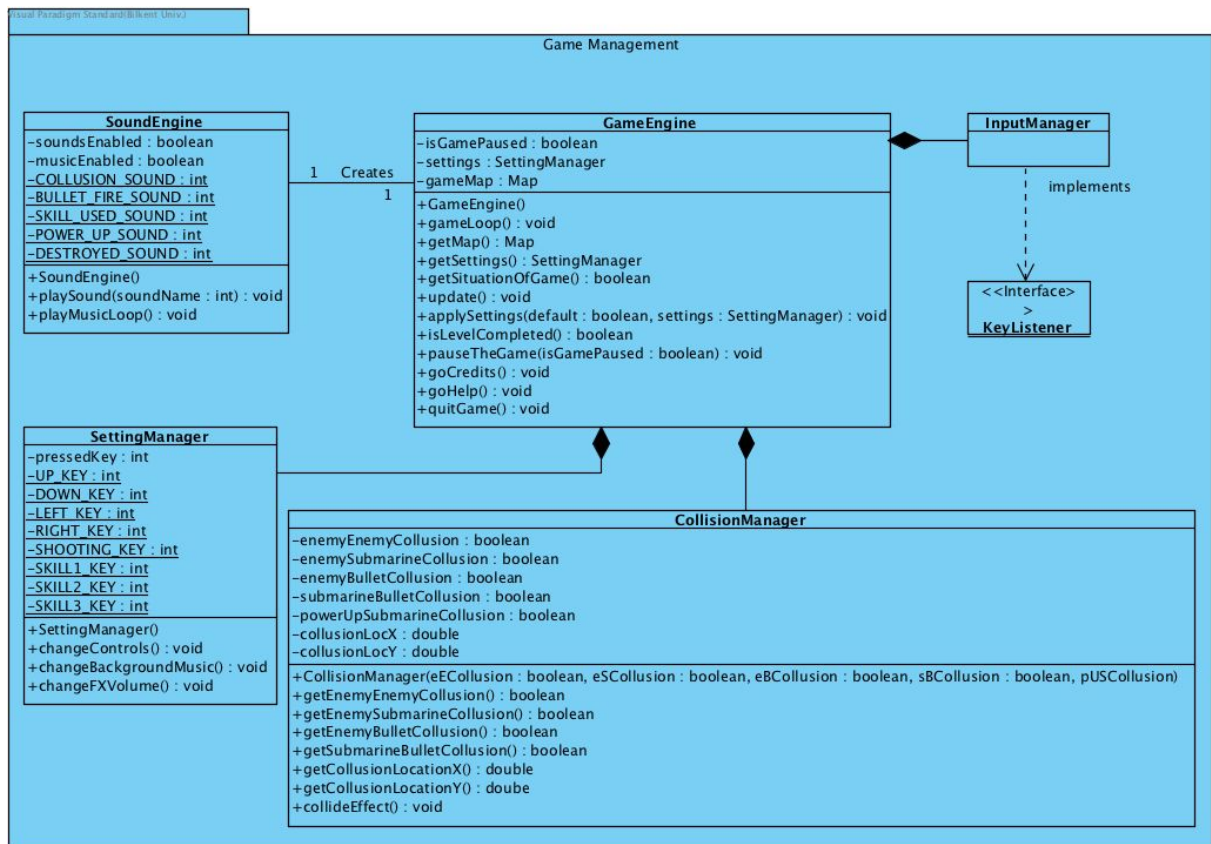


Figure-2(GameManagement Subsystem Interface)

GameEngine Class:

Visual Paradigm Standard (Bilkent Univ.)	GameEngine
-isGamePaused : boolean -settings : SettingManager -gameMap : Map	
+GameEngine() +gameLoop() : void +getMap() : Map +getSettings() : SettingManager +getSituationOfGame() : boolean +update() : void +applySettings(default : boolean, settings : SettingManager) : void +isLevelCompleted() : boolean +pauseTheGame(isGamePaused : boolean) : void +goCredits() : void +goHelp() : void +quitGame() : void	

Attributes:

private boolean isGamePaused: This variable will be true or false whether the game is paused or not.

private SettingManager settings: This object will help us to modify the settings of the game.

private Map gameMap: This object will be our backbone since the whole game will be executed on a map.

Constructor:

GameEngine(): This basically constructs a GameEngine object. Since there will be only one Game Engine in the game, there is no parameter needed in order to differentiate it from other objects.

Methods:

public void gameLoop(): This will basically create the infinite loop of the game.

public Map getMap(): This will return the gameMap variable in order to manipulate it.

public SettingManager getSettings(): This will return the settings variable in order to manipulate it.

public boolean getSituationOfGame(): This method will return true or false whether the game is paused or not.

public void update(): This method will help us to update the game engine when the game is paused, or a setting is changed, or there is a change on the map.

public void applySettings(boolean default, SettingManager settings): This method will help us to apply the new settings on the game engine.

public boolean isLevelCompleted(): This will return true or false based on the completion of a level.

public void pauseTheGame(boolean isGamePaused): This method will help us to pause the game if the pause variable becomes true.

public void goCredits(): This method will help us to go to the credits if the game is paused.

public void goHelp(): This method will help us to go to the help if the game is paused.

public void quitGame(): This method will help us to quit the game if anything unexpected happens while saving the stats of the player.

InputManager class



- This class is implemented in order to take the inputs from the player. Since the player will play the game through the keyboard(moving the submarine, shooting, using skills etc.), a key listener as an action listener can be used. This is already an implemented interface in Java Library, therefore, we will use that library.

SoundEngine Class

SoundEngine
<ul style="list-style-type: none">-soundsEnabled : boolean-musicEnabled : boolean-COLLUSION_SOUND : int-BULLET_FIRE_SOUND : int-SKILL_USED_SOUND : int-POWER_UP_SOUND : int-DESTROYED_SOUND : int
<ul style="list-style-type: none">+SoundEngine()+playSound(soundName : int) : void+playMusicLoop() : void

Menu and Map classes have an object of this class and they call soundsEnabled and musicEnabled methods. SoundEngine class will implement Runnable.

Attributes:

private boolean soundsEnabled: soundsEnabled will hold the boolean value of, sounds are either enabled or disabled, in settings.

private boolean musicEnabled: musicEnabled will hold the boolean value of, music is either enabled or disabled, in settings.

Constant int attributes in SoundEngine class are to decide which sounds will be played when the playSound method is called.

Constructors:

public SoundEngine(): Creates an object of SoundEngine class, soundEnabled and musicEnabled are set to true.

Methods:

public void playSound(int soundName) : playSound method plays a sound according to given sound name.

public void playMusicLoop() : playMusicLoop starts playing the music if musicEnabled is true and stops when it is false.

SettingManager Class:

```

-pressedKey : int
-UP_KEY : int
-DOWN_KEY : int
-LEFT_KEY : int
-RIGHT_KEY : int
-SHOOTING_KEY : int
-SKILL1_KEY : int
-SKILL2_KEY : int
-SKILL3_KEY : int

+SettingManager()
+changeControls() : void
+changeBackgroundMusic() : void
+changeFXVolume() : void

```

Attributes:

private int pressedKey: This attribute will return the key that has been pressed by the player in order to help us to change the controls of the game.

private final int UP KEY: This attribute will return the up key that is assigned for the game.

private final int DOWN_KEY: This attribute will return the down key that is assigned for the game.

private final int LEFT_KEY: This attribute will return the left key that is assigned for the game.

private final int RIGHT_KEY: This attribute will return the right key that is assigned for the game.

private final int SHOOTING_KEY: This attribute will return the shooting key that is assigned for the game.

private final int SKILL1_KEY: This attribute will return the first skill key, which is for activating the mass destruction skill, that is assigned for the game.

private final int SKILL2_KEY: This attribute will return the second skill key, which is invulnerability for 10 seconds, that is assigned for the game.

private final int SKILL3_KEY: This attribute will return the third skill key, which is twice as fast shooting for 10 seconds, that is assigned for the game.

Constructor:

SettingManager(): This constructor will create a manager object for settings. Since there will be only one setting which will be modified throughout the game, any parameters won't be needed.

Methods:

public void changeControlls(): This method will help us to change the controls of the game.

public void changeBackgroundMusic(): This method will help us to change the level of the background music of the game.

public void changeFXVolume(): This method will help us to change the level of the FX volume of the game.

CollisionManager Class:

CollisionManager	
-enemyEnemyCollusion : boolean -enemySubmarineCollusion : boolean -enemyBulletCollusion : boolean -submarineBulletCollusion : boolean -powerUpSubmarineCollusion : boolean -collusionLocX : double -collusionLocY : double	
+CollisionManager(eECollusion : boolean, eSCollusion : boolean, eBCollusion : boolean, sBCollusion : boolean, pUSCollusion)	
+getEnemyEnemyCollusion() : boolean +getEnemySubmarineCollusion() : boolean +getEnemyBulletCollusion() : boolean +getSubmarineBulletCollusion() : boolean +getCollusionLocationX() : double +getCollusionLocationY() : double +collideEffect() : void	

Attributes:

private boolean enemyEnemyCollusion: This attribute will hold true or false whether the object represent an enemy- enemy collision or not.

private boolean enemySubmarineCollision: This attribute will hold true or false whether the object represent an enemy- submarine collision or not.

private boolean enemyBulletCollision: This attribute will hold true or false whether the object represent an enemy- bullet collision or not.

private boolean submarineBulletCollision: This attribute will hold true or false whether the object represent an submarine- bullet collision or not.

private boolean powerUpSubmarineCollision: This attribute will hold true or false whether the object represent an submarine- power up collision or not in order to give the submarine the perks of the power up.

private double collisionLocX: This attribute will hold the horizontal location of the collision.

private double collisionLocY: This attribute will hold the vertical location of the collision.

Constructor:

public CollusionManager(boolean enemyEnemyCollision, boolean enemySubmarineCollision, boolean enemyBulletCollision, boolean submarineBulletCollision, boolean powerUpSubmarineCollusion): This constructor will create a collision object. Since each collision type will have different effects, the boolean attributes must be taken here in order to create the proper collision object.

Methods:

public boolean getEnemyEnemyCollision(): This method will return the value of the enemy-enemy collision attribute in order to understand the collision type.

public boolean getEnemySubmarineCollision(): This method will return the value of the enemy-submarine collision attribute in order to understand the collision type.

public boolean getEnemyBulletCollision(): This method will return the value of the enemy-bullet collision attribute in order to understand the collision type.

public boolean getSubmarineBulletCollision(): This method will return the value of the submarine-bullet collision attribute in order to understand the collision type.

public boolean getSubmarinePowerUpCollision(): This method will return the value of the submarine-power up collision attribute in order to understand the collision type.

public double getCollisionLocationX (): This method will return the horizontal location of the collision.

public double getCollisionLocationY(): This method will return the vertical location of the collision.

public void collideEffect(): This method will basically perform the collision's effects on the regarding objects.

3.4 Game Objects Subsystem Interface

This subsystem is the subsystem that holds game objects of our system. Diagram for the system is shown in the following section.

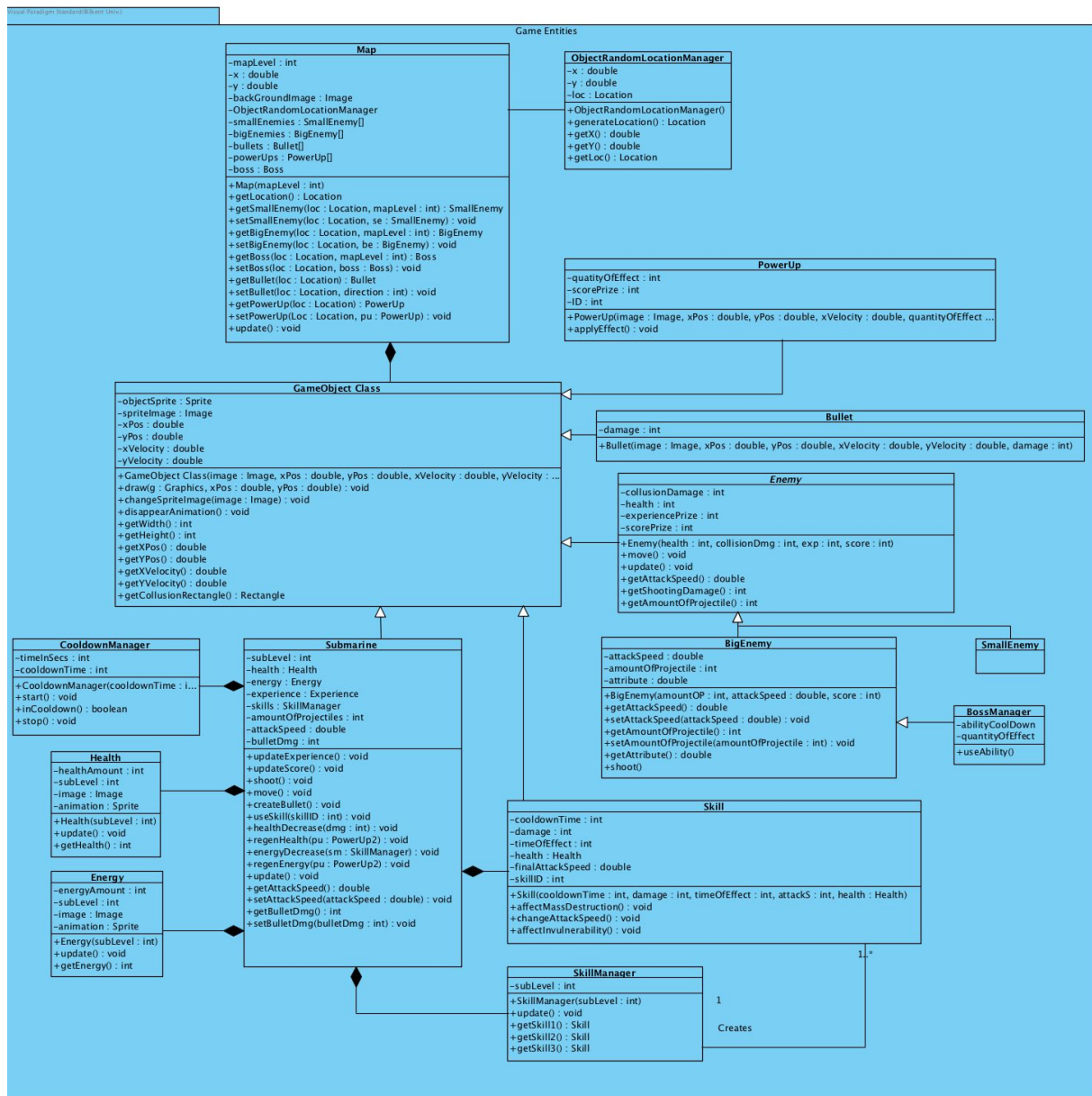


Figure-3 (Game Entities Subsystem)

Map Class:

Visual Paradigm Standard (Bilkent Univ.)	Map
<pre>-mapLevel : int -x : double -y : double -backGroundImage : Image -ObjectRandomLocationManager -smallEnemies : SmallEnemy[] -bigEnemies : BigEnemy[] -bullets : Bullet[] -powerUps : PowerUp[] -boss : Boss</pre>	
<pre>+Map(mapLevel : int) +getLocation() : Location +getSmallEnemy(loc : Location, mapLevel : int) : SmallEnemy +setSmallEnemy(loc : Location, se : SmallEnemy) : void +getBigEnemy(loc : Location, mapLevel : int) : BigEnemy +setBigEnemy(loc : Location, be : BigEnemy) : void +getBoss(loc : Location, mapLevel : int) : Boss +setBoss(loc : Location, boss : Boss) : void +getBullet(loc : Location) : Bullet +setBullet(loc : Location, direction : int) : void +getPowerUp(loc : Location) : PowerUp +setPowerUp(Loc : Location, pu : PowerUp) : void +update() : void</pre>	

Attributes:

private int mapLevel: hold the current level of the map that player is playing. It will be either 1, 2 or 3.

private double x: Horizontal measurement of the map.

private double y: Vertical measurement of the map.

private Image backgroundImage: Image that is used as background scene for the specific map according to mapLevel.

private ObjectRandomLocationManager: holds instance of Random location manager to generate random numbers for objects' position and some other attributes.

private SmallEnemy[] smallEnemies: Array that holds all SmallEnemy objects that are currently on the map.

private BigEnemy[] bigEnemies: Array that holds all BigEnemy objects that are currently on the map.

private Bullet[] Bullets: Array that holds all Bullet objects that are currently on the map.

private PowerUp[] powerUps: Array that holds all PowerUp objects that are currently on the map.

private Boss boss: holds Boss object that is met in the end of the map.

Constructor:

Map(int mapLevel): Constructor that constructs the map according to game level. Image, enemies, boss and other attributes will be constructed according to the level of the map.

Methods:

public Location getLocation(): Returns the location in the map as a location object

public SmallEnemy getSmallEnemy(Location loc, int mapLevel): Returns a small enemy object at specified location and map's level.

public void setSmallEnemy(Location loc, SmallEnemy se): Sets a small enemy at specified location.

public BigEnemy getBigEnemy(Location loc, int mapLevel): Returns a big enemy object at specified location and map's level.

public void setBigEnemy(Location loc, BigEnemy be): Sets a small enemy at specified location.

public Boss getBoss(Location loc, int mapLevel): Returns a boss object at specified location and map's level.

public void setBoss(Location loc, Boss boss): Constructs a boss at specified location .

public Bullet getBullet (Location loc): Returns the bullet object at specified location .

public void setBullet(Location loc, int direction): Constructs a bullet object at specified location and direction.

public PowerUp getPowerUp(Location loc): Returns the power up object according to location.

public void setPowerUp(Location loc, PowerUp pu): Constructs specified power up on the specified location.

public void update(): Updates the condition and all attributes on the map.

GameObject Class

Visual Paradigm Standard (Bilkent Univ.)	GameObject Class
<pre>-objectSprite : Sprite -spriteImage : Image -xPos : double -yPos : double -xVelocity : double -yVelocity : double +GameObject Class(image : Image, xPos : double, yPos : double, xVelocity : double, yVelocity : double) +draw(g : Graphics, xPos : double, yPos : double) : void +changeSpriteImage(image : Image) : void +disappearAnimation() : void +getWidth() : int +getHeight() : int +getXPos() : double +getYPos() : double +getXVelocity() : double +getYVelocity() : double +getCollusionRectangle() : Rectangle</pre>	

Attributes:

private Sprite objectSprite: This variable will take the animation of any object in the game since it will be a subclass of the GameObject class.

private Image spriteImage: This variable will take the image of the animation of the specific game object.

private double xPos: This variable will hold the horizontal position of the game object.

private double yPos: This variable will hold the vertical position of the game object.

private double xVelocity: This variable will hold the horizontal speed of the game object.

private double yVelocity: This variable will hold the vertical speed of the game object.

Constructor:

GameObject(Image image, double xPos, double yPos, double xVelocity, double yVelocity): This will create an object in the game. It will take the image, the positions(horizontal and vertical), and the velocities(horizontal and vertical) of the game as an input.

Methods:

public void draw(Graphics g, double xPos, double yPos): This method will help us to draw the object as a mockup in order to send it into the map to be shown to the player.

public void changeSpriteImage(Image image): This method will help us to change images of the object in different conditions in order to create animations.

public void disappearAnimation(): This method will help us to create a disappear animation for the object.

public int getWidth(): This method will help us to get the width of the object based on the image of the object.

public int getHeight(): This method will help us to get the height of the object based on the image of the object.

public double getXPos(): This method will return the horizontal position of the object.

public double getYPos(): This method will return the vertical position of the object.

public double getXVelocity(): This method will return the horizontal speed of the object.

public double getYVelocity(): This method will return the vertical speed of the object.

public Rectangle getCollisionRectangle(): This method will help us to get the minimum frame which contains the collision of two objects.

Bullet class

Visual Paradigm Standard (Kasymbek Tashbaev (Bilkent Univ.))	Bullet
-damage : int	
+Bullet(Image image, double xPos, double yPos, double xVelocity, double yVelocity, int damage)	

Attributes:

private int damage: the damage that bullet does when it hits the enemy

Constructors:

Bullet(Image image, double xPos, double yPos, double xVelocity, double yVelocity, int damage): constructor of Bullet class creates an instance of Bullet object with specified Image and velocity. Bullets on the game will be created at a specified position and move with a constant velocity.

PowerUp Class

Visual Paradigm Standard (Bilkent Univ.)	PowerUp
-quantityOfEffect : int	
-scorePrize : int	
-ID : int	
+PowerUp(image : Image, xPos : double, yPos : double, xVelocity : double, quantityOfEffect : int, ID)	
+applyEffect() : void	

Attributes:

private int quantityOfEffect: powerup applies its effect by the numeric value held by this attribute.

private int ID: hold the numeric value that indicates the type of powerup. There are two types: first regenerates health, the second regenerates energy.

PowerUp(Image image, double xPos, double yPos, double xVelocity, int quantityOfEffect, ID): creates a powerup with the given image, quantity of effect at given position. It will move to the left with the given velocity.

public void applyEffect(): when picked up, according to the type of a powerup the corresponding effect will be applied to submarine

This class extends `GameObject` class

```

Visual Paradigm Standa Submarine sfcbasev(Bilkent Univ.)
- subLevel : int
- health : Health
- energy : Energy
- experience : Experience
- skills : SkillManager
- amountOfProjectiles : int
- attackSpeed : double
- bulletDmg : int

+ updateExperience() : void
+ updateScore() : void
+ shoot() : void
+ move() : void
+ createBullet() : void
+ useSkill(skillID : int) : void
+ healthDecrease(dmg : int) : void
+ regenHealth(pu : PowerUp) : void
+ energyDecrease(sm : SkillManager) : void
+ regenEnergy(pu : PowerUp) : void
+ update() : void
+ getAttackSpeed() : double
+ setAttackSpeed(attackSpeed : double) : void
+ getBulletDmg() : int
+ setBulletDmg(bulletDmg : int) : void

```

Attributes:

private int bulletDamage: Bullet damage is saved as an int number as the attribute of Submarine.

private int subLevel: hold a current level of the submarine. It increases each time by one when submarine has enough experience points to reach the next level.

private int amountOfProjectiles: hold the numeric value of bullets that submarine releases when function shoot() is called. Its value increases as the level of submarine increases

private Health health: Health submarine will be accounted for the instance of Health class. When the health of submarine increases or decreases, Health object will change health bar displayed to player accordingly.

private Experience experience: Holds the instance of the Experience class, that. Will be increased when enemies are defeated using update experience() method.

private SkillManager skills: Holds an instance of SkillManager class that will provide the submarine to use Skill objects.

private double attackSpeed: Submarine can shoot ones in some second specified in this attribute

private int bulletDmg: When submarine shoots and the bullet hits the enemy, its health decreases by the value specified in this attribute.

Constructor:

Submarine(int subLevel): Initializes submarine based on its level.

Methods:

public void move(): According to input that user enters the method changes xVelocity and yVelocity.

public void updateScore(): When an enemy is defeated, this method will update game score according to enemy's score prize.

public void updateExperience(): When the enemy is defeated or powerup is picked up, this method will update submarine's experience according to enemy's experience prize.

public void regenHealth(PowerUp pu): When submarine picks up “Repair kit” powerup, this method increases the health of the submarine according to powerup’s quantityOfEffect.

public void createBullet(): When submarine shoots, this method calls Bullet class to create bullets at the position of the submarine.

public void regenEnergy(PowerUp pu): When submarine picks up “Fuel” powerup, this method increases the energy of the submarine according to powerup’s quantityOfEffect.

public void energyDecrease(SkillManager sm): When submarine uses any skill, this method decreases the energy according to skill’s energyCost.

public void useSkill(int skillID): Submarine uses the skill that corresponds to ID given.

public void shoot(): Uses a createBullet method to create bullets according to amountOfProjectile that move to the right.

public void healthDecrease(int dmg): When the submarine is hit by enemy’s bullet, its health is decreased by bullet’s damage.

public void update(): Each time changes submarine’s attributes, if they are affected.

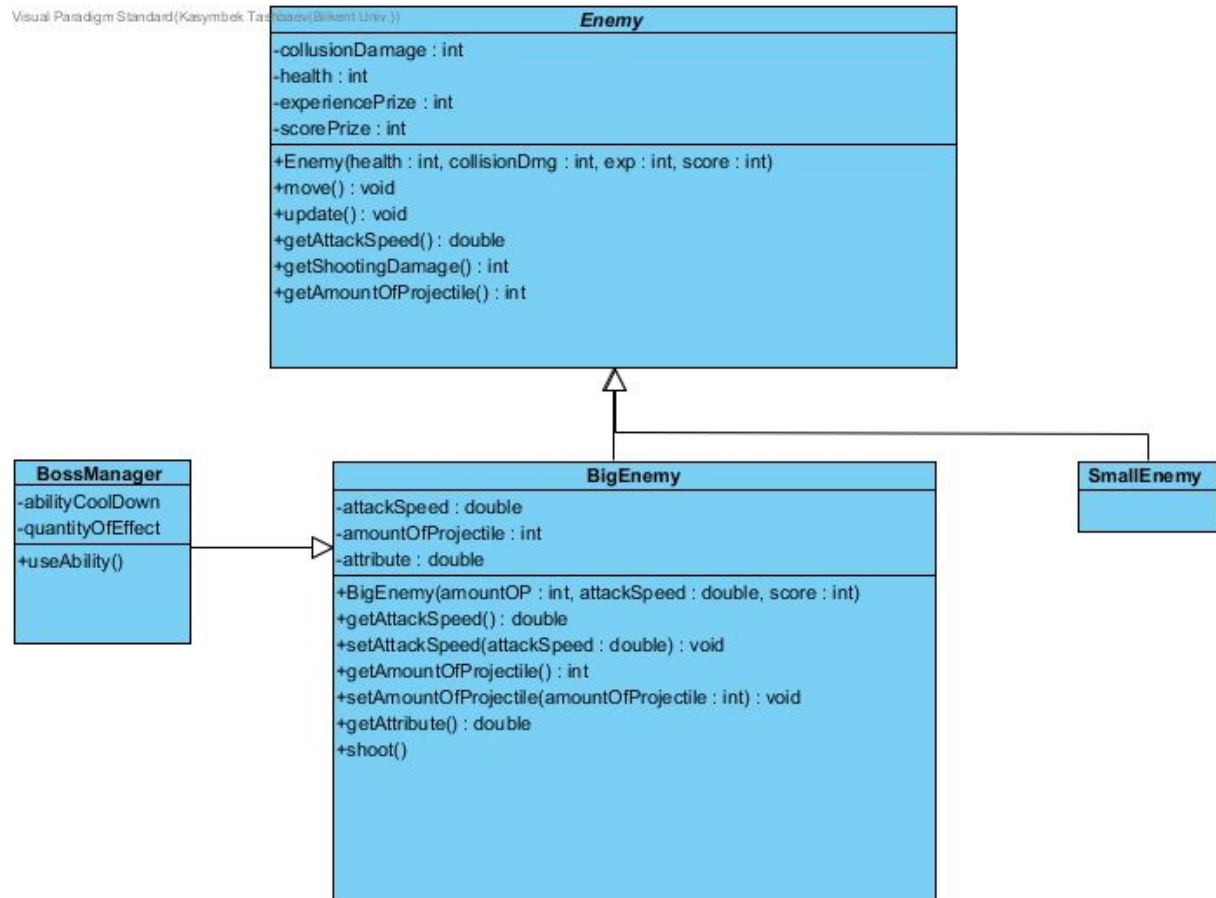
public int getBulletDmg(): Getter method for bulletDmg attribute.

public void setBulletDmg(int bdmg): Setter method for bulletDmg attribute.

public double attackSpeed(): Getter method for attackSpeed attribute.

public set attackSpeed(double as): Setter method for a bulletDmg attribute.

Enemy Class:



This class will be an abstract class that will extends GameObject.

Attributes:

private int collisionDamage: Damage that will be caused by the enemy if a collision happens.

private int health: Health of enemy in an integer.

private int experiencePrize: Experience points that will be gained by the player when enemy defeated. Gaining experience points help submarine to level up.

private int scorePrize: Score points that will be gained by the player when the enemy is defeated.

private double locX: This will hold the horizontal location of the enemy.

private double locY: This will hold the vertical location of the enemy.

Constructor:

Enemy(int collisionDamage, int health, int exp, int score): Enemy constructor that takes collision damage, enemy's health, experience and score points that it will give to the player as parameters and constructs the object.

Methods:

public void move(): Method that controls the movement of the enemy in map.

public void update(): Update method that updates enemy's current stats and condition.

Small Enemy Class:

This class will extend enemy class.

Attributes:

Constructor:

SmallEnemy(double locX, double locY, int mapLevel): This will construct a small enemy with its location. However, since the collision damage of the small enemy will be based on the map level, the map level should be taken as a parameter.

Methods:

Big Enemy Class:

This class will extend enemy class.

Attributes:

private int attackSpeed: This attribute will hold the attack speed of the big enemy as bullet shooting pace.

private int attackDamage: This attribute will hold the damage of each projectile released by a big enemy.

private int amountOfProjectile: Holds the number of projectiles submarine releases each time it shoots.

Constructor:

BigEnemy(double locX, double locY): This will construct a big enemy with its location. However, since the collision damage and the bullet damage of the big enemy will be based on the map level, the map level should be taken as a parameter.

Methods:

private void shoot(): This method be used in order to make a big enemy to attack the submarine.

public int getAttackDamage(): Getter method for attackDamage attribute.

public void setAttackDamage(int bdmg): Setter method for attackDamage attribute.

public double attackSpeed(): Getter method for attackSpeed attribute.

public set attackSpeed(double as): Setter method for a bulletDmg attribute.

public double attackSpeed(): Getter method for attackSpeed attribute.

public void setAttackSpeed(double as): Setter method for a bulletDmg attribute.

Boss Manager Class:

This class will extend BigEnemy class.

Attributes:

private int abilityCooldown: Boss in addition to the shooting has a special ability that is applied each time when abilityCooldown is passed. This value is very important because the final boss will be different than regular bosses at each level. Therefore, the special ability that it will use will be different than the regular bosses.

private int quantityOfAbility: special ability applies its effect according to the numeric value held by this attribute.

Constructor:

BossManager(double locX, double locY, int mapLevel): This will construct the boss with its location. However, since the collision damage, the bullet damage and the skills of the boss will be based on the map level, the map level should be taken as a parameter.

Methods:

public void useAbility(): This method will be used in order to make the boss use its special ability against the submarine.

ObjectRandomLocationManager Class:

ObjectRandomLocationManager
-x : double -y : double -loc : Location
+ObjectRandomLocationManager() +generateLocation() : Location +getX() : double +getY() : double +getLoc() : Location

Attributes:

private double x: This will hold the horizontal position of an object.

private double y: This will hold the vertical position of an object.

private Location loc: This variable will hold the both position given above as a Location object for the sake of implementation.

Constructor:

ObjectRandomLocationManager(): Since the manager will only generate Location for the objects, it won't take any parameter inside.

Methods:

public Location generateLocation(): This method will create random horizontal and vertical positions firstly. Then, these positions will be saved to the specific attributes accordingly.

Finally, the Location object will be created and set into the loc attribute. Finally the loc attribute will be returned.

public double getX(): This method will return the value of the attribute x.

public double getY(): This method will return the value of the attribute y.

public Location getLoc(): This method will just return the Location object loc instead of creating it.

CooldownManager Class:

CooldownManager
-timeInSecs : int -cooldownTime : int
+CooldownManager(cooldownTime : int) +start() : void +inCooldown() : boolean +stop() : void

Attributes:

private int timeInSecs: Cooldown time converted to seconds to be able to compare the passed time.

private int cooldownTime: Desired cooldown time.

Constructor:

CooldownManager(int cooldownTime): Constructor for cooldown manager takes cooldownTime as a parameter to be able to compare the passed time with the desired cooldown time.

Methods:

public void start(): This method starts the count for cooldown.

public boolean inCooldown(): This method returns a boolean value to show if the cooldown continues or not.

public void stop(): This method stops the counter for cooldown when it reaches the specific cooldownTime.

SkillManager Class:

SkillManager
-subLevel : int
+SkillManager(subLevel : int)
+update() : void
+getSkill1() : Skill
+getSkill2() : Skill
+getSkill3() : Skill

Attributes:

private int subLevel: This attribute will hold the current level of the submarine.

Constructor:

SkillManager(int subLevel): creates a skill level according to the level of submarine

Methods:

public void update(): This method will update the condition of the skills. For example, if the submarine levels up, this method will be called in order to let submarine acquire the new skill.

public Skill getSkill1(): returns a skill object with properties of first skill.

public Skill getSkill2(): returns a skill object with properties of second skill.

public Skill getSkill3(): returns a skill object with properties of third skill.

Health Class:

Health
-healthAmount : int
-subLevel : int
-image : Image
-animation : Sprite
+Health(subLevel : int)
+update() : void
+getHealth() : int

Attributes:

private int healthAmount: This attribute will hold the amount of health of an object as an integer.

private int subLevel: This will hold the level of the submarine.

private Image image: This will hold the image of a health bar.

private Sprite animation: This will hold the animation of the health bar in order to make the decrease and the increase in the health.

Constructor:

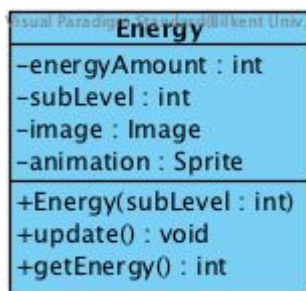
Health(int subLevel): This will create the health of the submarine. Since the health of the submarine will depend on the level of the submarine, it will take the level as a parameter.

Methods:

public void update(): This method will update the health if the submarine will collide with an enemy or if it takes a bullet.

public void getHealth(): This will return the total health of the submarine.

Energy Class:



The image shows a UML class diagram for the 'Energy' class. The class name 'Energy' is at the top. Below it, the attributes are listed: -energyAmount : int, -subLevel : int, -image : Image, and -animation : Sprite. Below the attributes, the methods are listed: +Energy(subLevel : int), +update() : void, and +getEnergy() : int.

Energy
-energyAmount : int
-subLevel : int
-image : Image
-animation : Sprite
+Energy(subLevel : int)
+update() : void
+getEnergy() : int

Attributes:

private int energyAmount: This attribute will hold the amount of energy of the submarine as an integer.

private int subLevel: This will hold the level of the submarine.

private Image image: This will hold the image of the energy bar.

private Sprite animation: This will hold the animation of the energy bar in order to make the decrease and the increase in the energy.

Constructor:

Energy(int subLevel): This will create the energy of the submarine. Since the energy of the submarine will depend on the level of the submarine, it will take the level as a parameter.

Methods:

public void update(): This method will update the energy if the submarine uses any skill.

public void getEnergy(): This will return the total energy of the submarine.

Skill Class:

Usual Paradigm Standard(Bilkent Univ.)		Skill
-cooldownTime : int		
-damage : int		
-timeOfEffect : int		
-health : Health		
-finalAttackSpeed : double		
-skillID : int		
+Skill(cooldownTime : int, damage : int, timeOfEffect : int, attackS : int, health : Health)		
+affectMassDestruction() : void		
+changeAttackSpeed() : void		
+affectInvulnerability() : void		

Attributes:

private int cooldownTime: This attribute hold the cooldown time of the skill, which means the time needs to be passed in order to reuse the skill.

private int damage: This attribute will hold the value of the damage the skill does if it is an attack skill.

private int timeOfEffect: This attribute will hold the time of effect of a skill if it is a skill which is processed for a certain amount of time.

private Health health: This attribute will hold the health of the submarine in order to process the invulnerability skill.

private double finalAttackSpeed: This attribute will hold the value of the final attack speed in order to reload it after fast attack speed is done.

private int skillID: This attribute will hold a unique ID for each skill in order to differentiate one another.

Constructor:

Skill(int cooldownTime, int damage, int timeOfEffect, int attackS, Health health): This will construct a skill based on its effects. It will determine the cooldown time of a skill, if it is attacking skill then the damage it will produce, and relative attributes such as that.

Methods:

private void affectMassDestruction(): This will make the effects of the mass destruction skill to the relevant objects if the skill is mass destruction skill.

private void changeAttackSpeed(): This will make the effects of the changing attack speed skill to the submarine if the skill is changing attack speed skill.

private void affectInvulnerability(): This will make the effects of the invulnerability skill to the submarine if the skill is invulnerability skill.

3.5 Detailed System Design

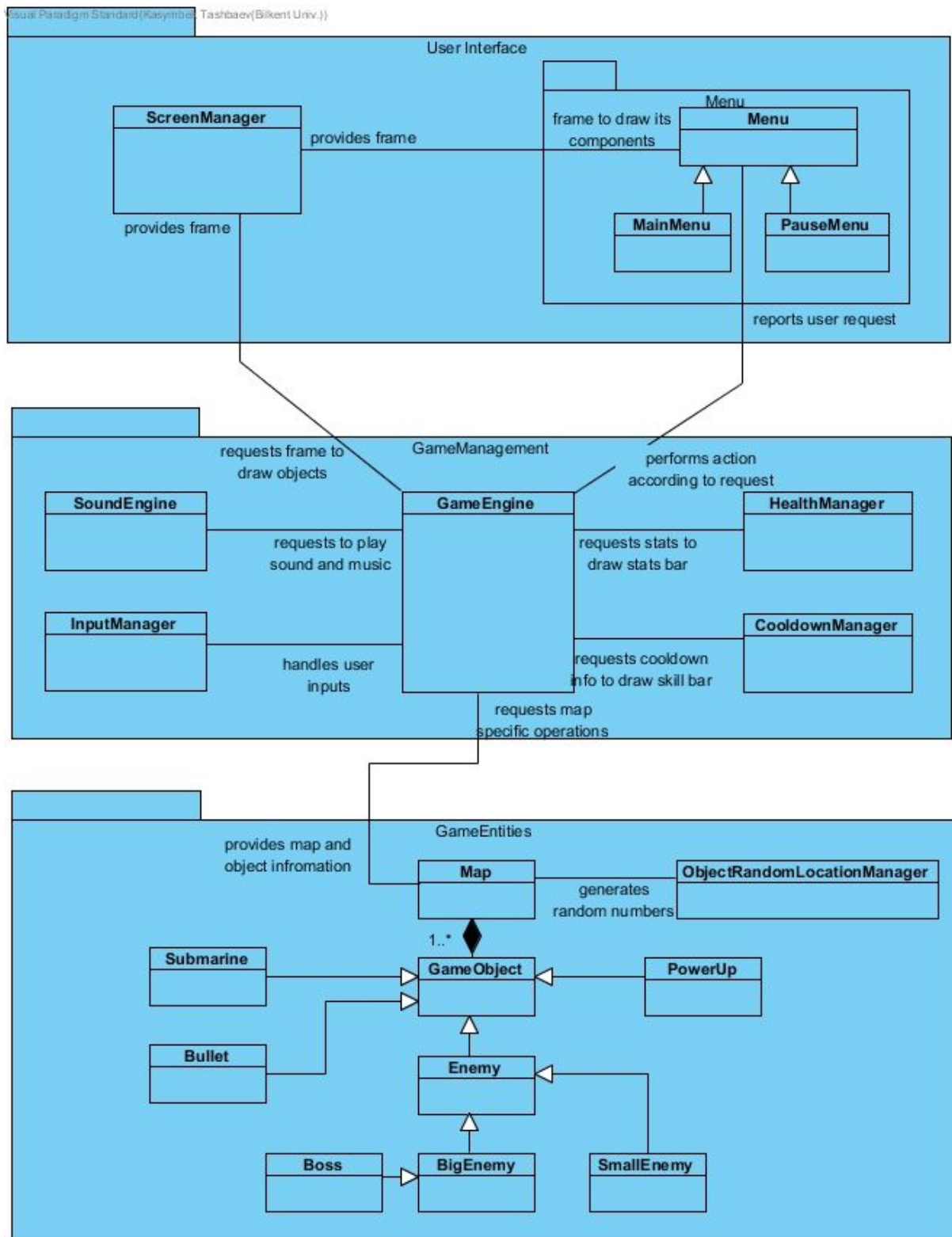


Figure-4(Detailed system design)