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

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 report. Important design goals will be explained in detail in the following section:

End User Criteria:

Easy to play: Sea adventures is designed as to be a game that can be played by both children and adults. Therefore gaming system should be easy to understand and control. The interface will be designed to be user-friendly so it will be easy to navigate through menus and options. At the same time, gameplay system will be simple enough for the player to understand and enjoy.

Entertaining: Game should be easy to understand however it should be difficult enough for the sake of entertainment. Sea Adventures will have different elements compared to classic space shooters game to increase entertainment value.

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 in future to provide richer gameplay.

Reliability: Sea Adventures should be reliable for the player to enjoy the gameplay. Therefore our system should be bug-free and not crash. In order to reach this outcome, we will do lots of testing simultaneously with the development.

Modifiability: For the same reasons as extendibility, modifiability is an important criteria for Sea Adventures as new game elements is thought to be added later to increase gameplay quality. Therefore functionalities in our system should be easy to modify and improve.

Efficiency: We will focus on increasing gaming speed rather than decreasing the memory usage of the game for quality gameplay. 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:

Easy to Play vs. Functional:

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:

JDK: [1] Java Development Kit

JVM: [1] Java Virtual Machine

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, reliability, modifiability and efficiency on the maintenance side. On the end-user side, we defined our design goals as being easy to play 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. Subsystem Decomposition

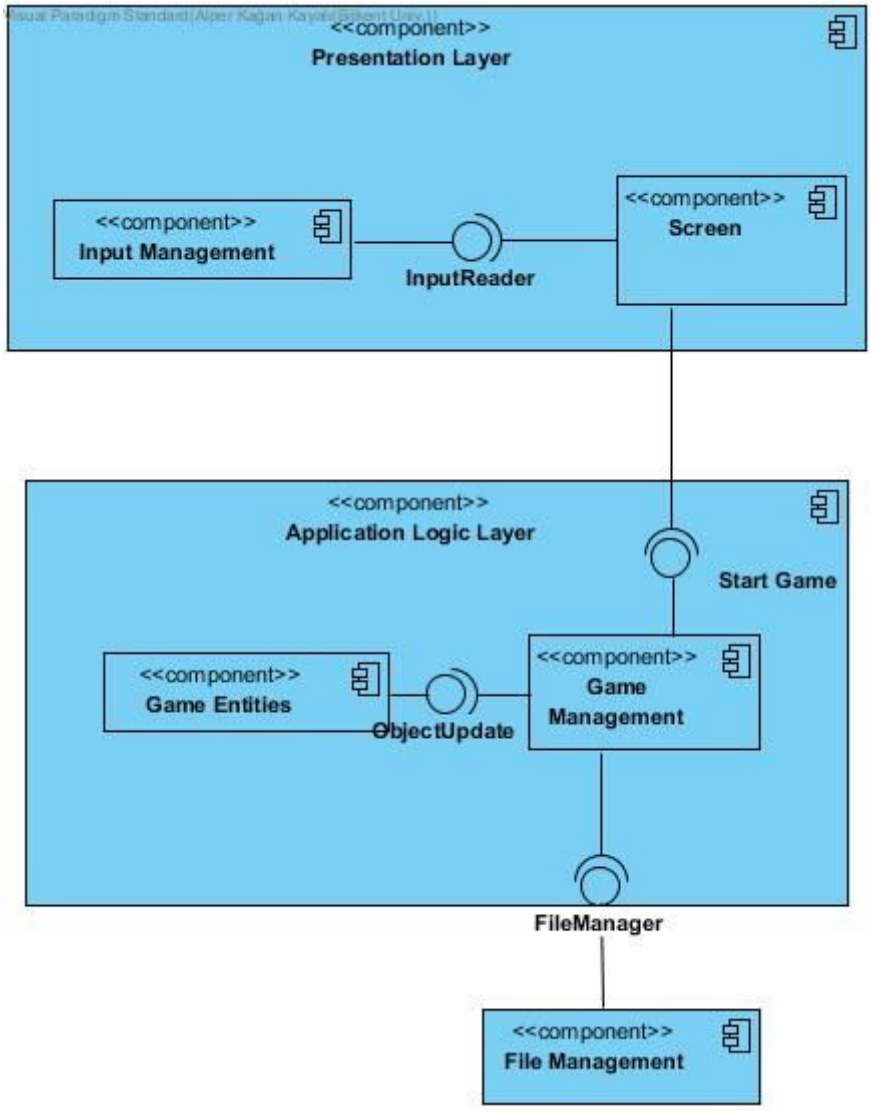


Figure – 1 (Basic Subsystem Decomposition)

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 **3- Layer architectural design** into our system.

Since we decided on the 3-Layer architectural design, we divided our architecture into 3 subsystems.

The reason why 3-Layer is suitable for this project is the following:

- The subsystems that we have created are very suitable to the 3-Layer architecture.

    - The **User Interface Subsystem**, which will help us to create the interaction between the game and the user.
    - The **Application Logic Subsystem**, which helps us to manage the logic in the background of the game. In addition, the control mechanism is in this subsystem which helps us to get the inputs from the user. In addition, this subsystem helps us to instantiate the game entities such as enemies, submarine, map etc. Finally, by using these features, this subsystem maintains the **game loop** as well.
    - The **File Management Subsystem**, which helps us to read and write the high scores of the game. This subsystem mainly is controlled and updated by the Game Management Subsystem.
- The relationships we have for the components are more suitable to the 3-Layer architecture
- 3-Layer is well-known and used software architecture. In addition, it is **effective.** It will help us to implement the game easily.

As in figure 1, the system is divided into three subsystems which are called User Interface, Application Logic and File Management subsystems. While each subsystem is connected to other subsystems in order to create the whole system, each of these subsystems has different objectives in order to maintain the system.

By that, we mean:

- User Interface Subsystem is a part to interact the user. In addition, this subsystem's only request of functional action is **'Start Game'** into the Game Management component of the Application Logic Layer. Therefore, any interaction through User Interface Subsystem either will be dealt within the subsystem, or it will be directed into the Application Logic Subsystem. This will help us to resolve issues in the User Interface easily since the problem is either in the subsystem or the Game Management component of the Application Logic Subsystem. Finally, it holds the views of the Game Entities component in the Application Logic Subsystem. Therefore, the updates will be done automatically when the conditions of the entities change.

- **Application Logic Subsystem** acts like the background logic of the game. It is responsible for 2 components:
    - The **Game Management** component is responsible for implementing the logic of the game. Specifically, it is responsible for starting/pausing/ending the main game loop and controlling the Map in order to detect collisions and make updates on the map.
    - The **Game Entities** component is responsible for the objects that will be on the game. The updates will be made on the **Game Management** component will automatically affect the objects here.
    - These Components will be explained in details in the upcoming chapters.
- **File Manager Subsystem** will help us to load or save the scores of the players.

In brief, our subsystem decomposition will help us to obtain high cohesion and low coupling which will make the system more flexible.

2.2. 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.

2.3. 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.

2.4. Access Control and Security

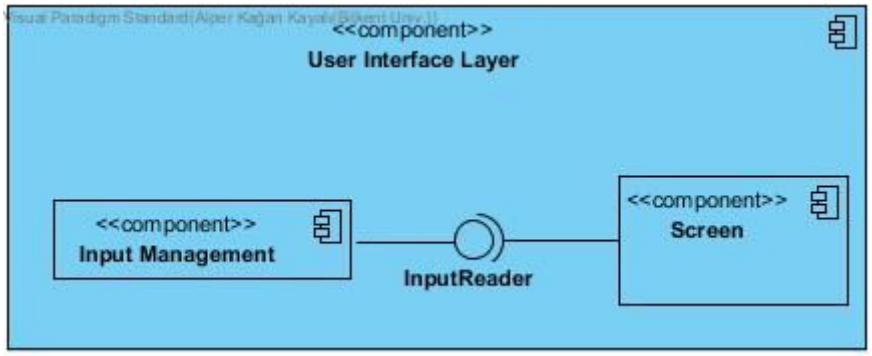
Sea adventures will not have player login system therefore, there will not be any database for storing details for users. For this reason, there will not be any controls or restrictions about accessing the game. Everyone who has the game file can access and play the game. Their data will not be saved 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.

2.5. Boundary Conditions

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

3. Subsystem Services

3. 1. User Interface Subsystem



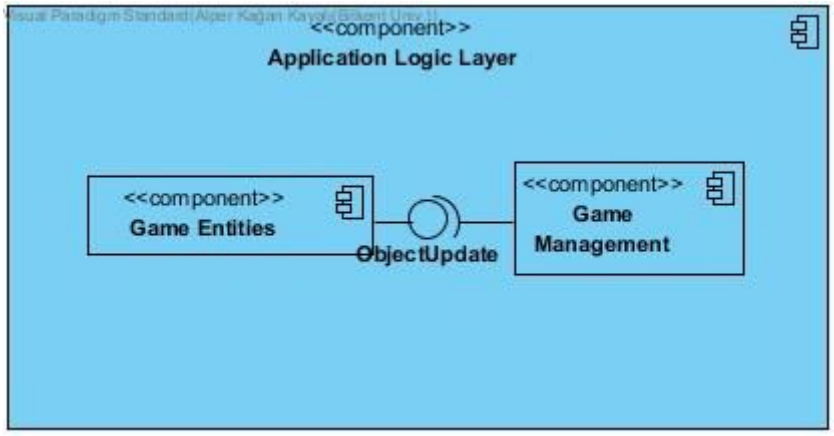
User Interface Layer

User Interface Subsystem is responsible for providing the user interface for Sea Adventures. It has two major components and they are called:

- Input Management Component
- Screen Component

When the user opens the game, User Interface Subsystem calls the **Screen** component in order to provide a menu to the user. The user will be able to select anything from this menu such as help screen, settings screen, credits screen etc. All of these interactions on the screen will be dealt with **Input Management** component. This component will take the input of the interaction and provide it into the Screen component in order to make the proper updates on the screen. If the user wishes to play, the **Screen** component will invoke the Application Logic Layer.

3. 2. Application Logic Subsystem



Application Logic Layer

When the user starts the game, the **Game Management** component will be invoked. This component will help us to update the state of the game, check collisions in the game, update the score etc. However, all of these updates and controls will be done based on the **Game Entities** component, which will provide the necessary game entities such as power-ups, enemies, submarine, map etc. Any kind of change done on the map will be controlled, and depending on the necessity, updates should be invoked by the **Game Management** component. When the game is finished, the stats of the user should be saved to the **File Manager**

3. 3. File Management Subsystem

File Management Layer

When the game ends, the **File Management** subsystem will be invoked in order to save the score of the player. In addition, when the high scores are displayed, the **File Management** subsystem needs to be invoked as well in order to load the data.

4. Low-level design

4.1 Object design trade-offs

Information Hiding vs. Efficiency:

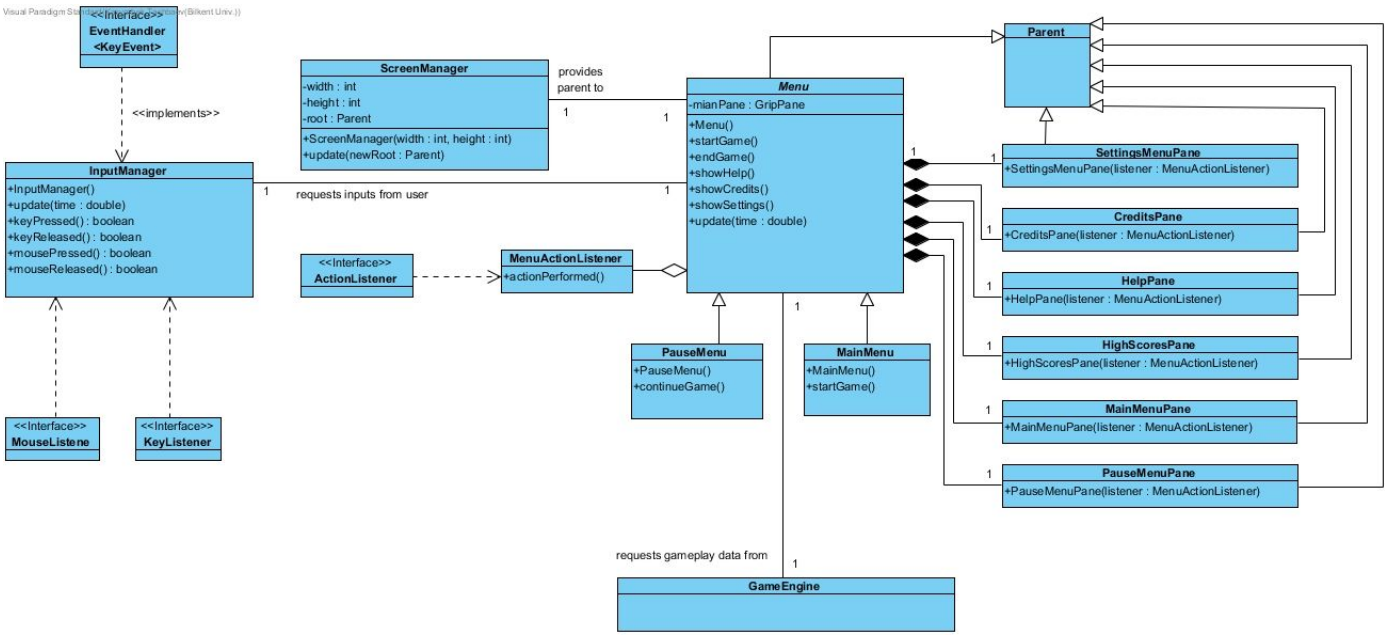
Our object design is based on the Parnas principle, which means as much encapsulation as possible. Since this is the case, the modifying attributes or reaching them will be through methods, which is more time-consuming than using them as public and directly reaching them. However, information hiding is better for our case and this trade-off is resolved by selecting information hiding.

Memory space vs. Response time:

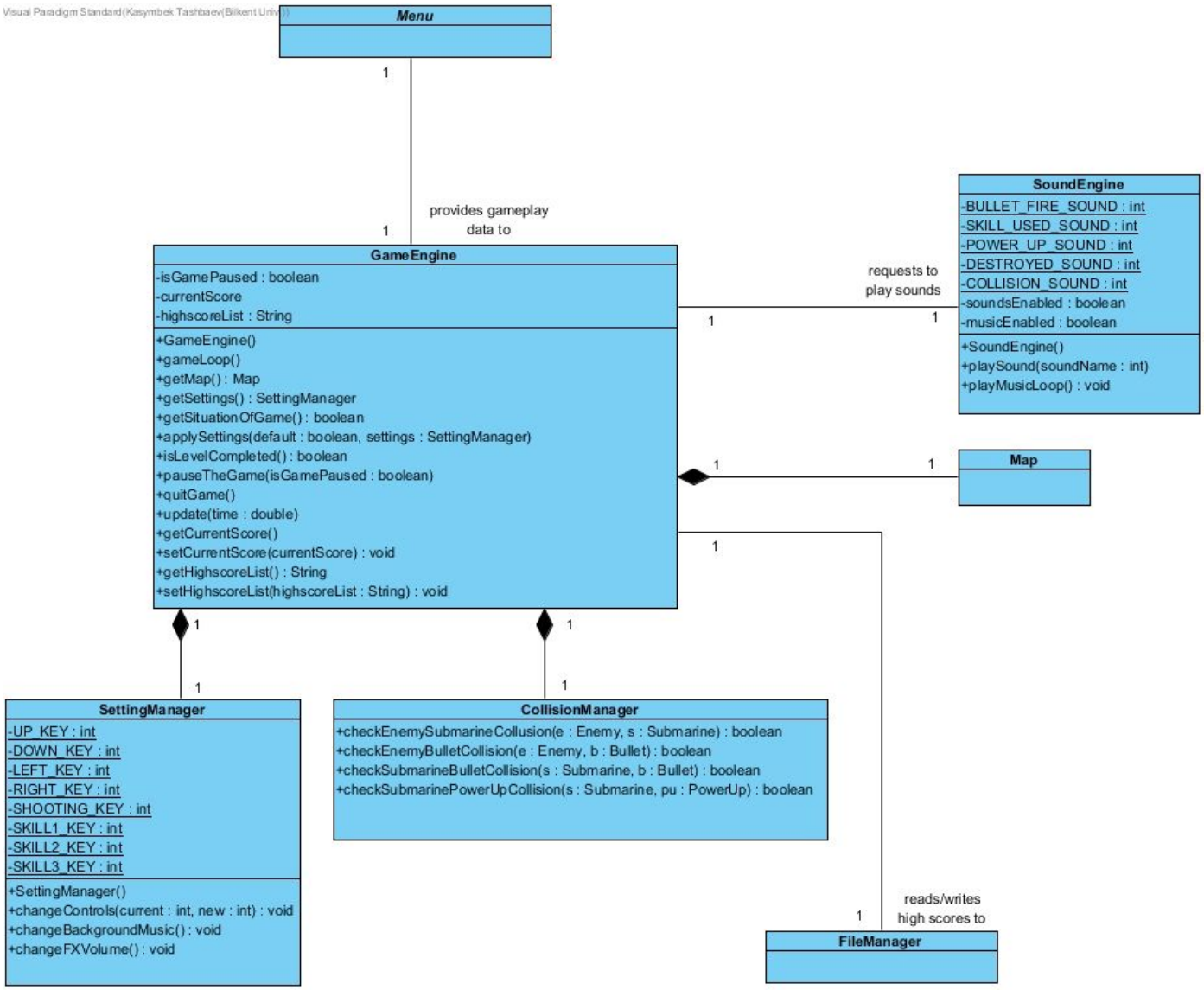
Our object design is based on the performance, because we want to give the user a performance-based game. Therefore, objects are designed in such a way that their response time is as small as possible. However, this will affect the memory space negatively. In the end, this trade-off is resolved by choosing response time over memory space.

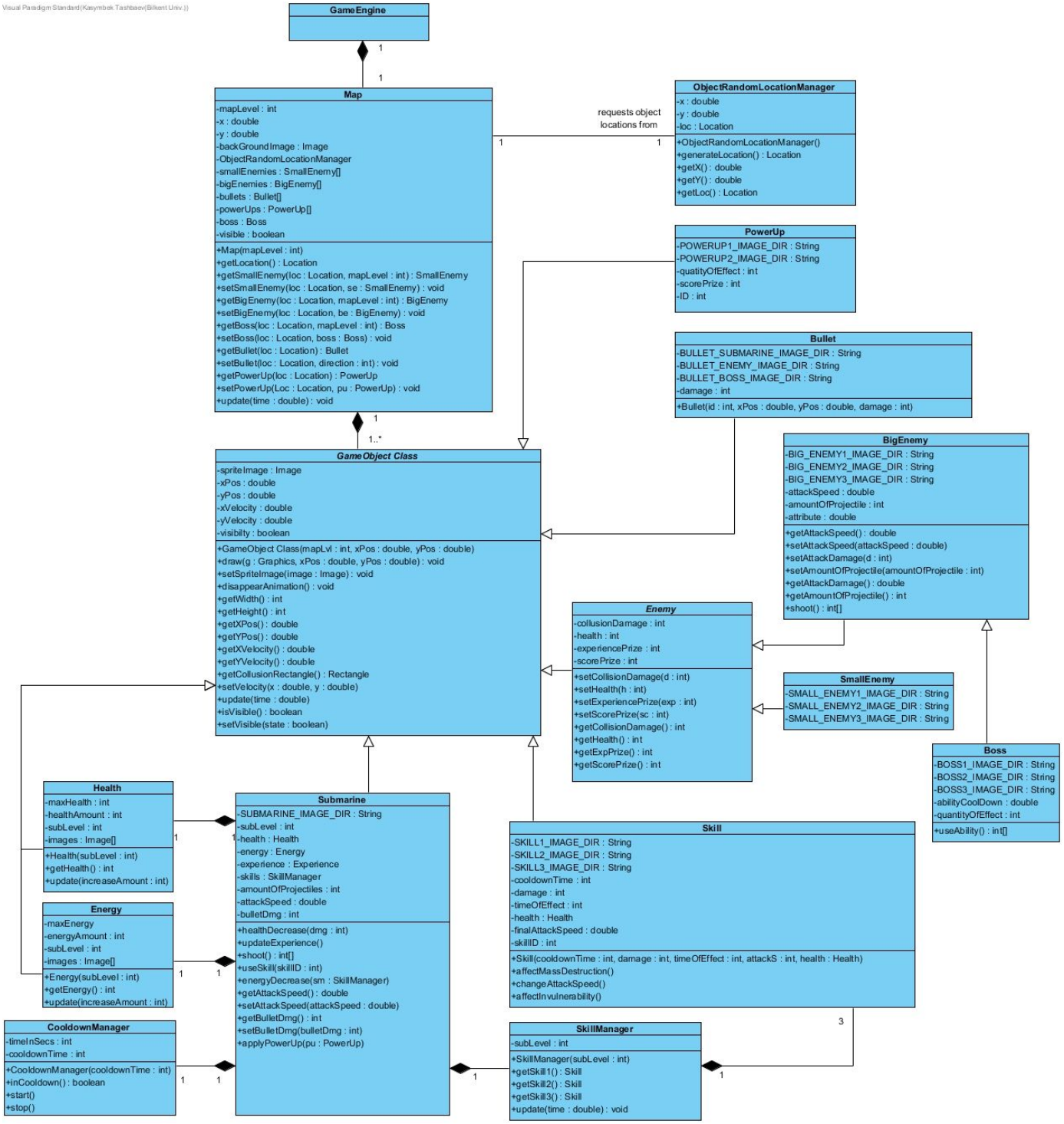
4.2 Final object design

4.2.1 User Interface Layer

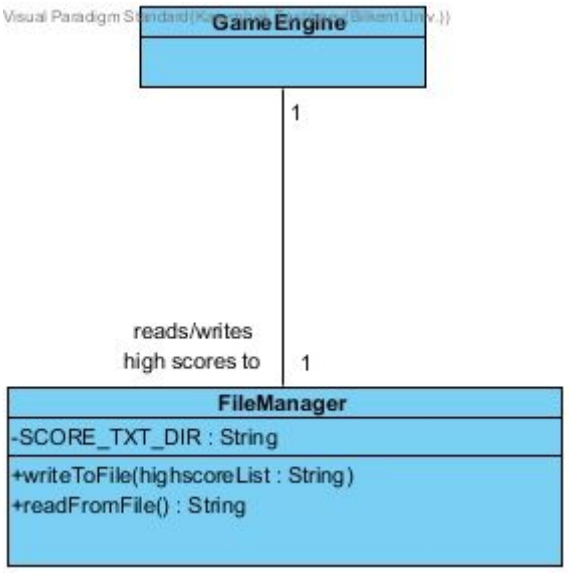


4.2.2 Application Logic Layer





4.2.3 File Management



4.3 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, Facade classes are Menu class in User Interface Layer and GameEngine in Application Logic Layer to communicate with other layers. Since in the File Management Layer there is only one class it will directly communicate with other layer and there is no Facade class in this layer. However, since Application Logic Layer is larger than other layers, in order to reduce dependency between classes in this layer there is an additional Facade class which is Map. Som in total we have three Facade classes.

Singleton Design Pattern:

Since we have many controller objects such as GameEngine, SoundEngine, etc. and they should have only one instance that other classes can access, we are using Singleton Design pattern in these classes. That will increase the maintainability of the game and ensure

that there is no more than one instance of controller objects. Singleton is applied to the following classes: FileManager, GameEngine, Map, Submarine, SettingManager, CollisionManager, SoundEngine, ScreenManager, InputManager.

Delegation Design Pattern:

Energy and Health classes are game objects but very different from classes that extend GameObject since they do not move, do not collide with other objects and cannot apply other function of GameObject. Therefore, Delegation design pattern is applied to Energy and Health classes, so they will have an instance of GameObject and use only some of its functions.

4.4 Packages

The packages will be ordered by taking consideration of the given component diagram. Since the structures of the packages and component diagram are the same, the packages are not included in this section.

4.5 Class Interfaces

ScreenManager Class

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

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

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.

public void update(): This method will help us with updating the current situation and current pane. How we will do is, we will take the input of the MenuActionListener, and we will understand which button did the user click. When we understood this, if any other option on the main menu is selected, it will update the game engine and main pane accordingly, nothing will change otherwise.

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

MainMenu Class

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. This will be implemented as creating a new Game Engine or referring to it if it is created before, and starting the game loop method in Game Engine.

PauseMenu class

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. This will be done easily since we use a singleton design pattern, we will refer to the game engine that is already created and simply, we will continue the game loop method in the Game Engine. In addition, this will override the startGame() method of the Menu class.

GameEngine Class:

public void gameLoop(boolean isGamePaused): This will basically create the infinite loop of the game. This will be done by basically creating the new map object, or referring it if it is already created, then updating the map in the infinite loop if the isGamePaused parameter is true. If the game is paused, then this parameter will be false and the game loop will be stopped. Since we use singleton design pattern, the map object will still remain during the pause. Therefore, when the user continues the game all of his/her stats will be there.

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. This will be done by checking the pause condition and the setting condition.

public void applySettings(boolean default, SettingManager settings): This method will help us to apply the new settings on the game engine. This will be implemented as if the default is false, then something is changed and the change is done by calling the necessary function on the settings parameter. Do nothing otherwise.

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.

public int update(double time): This method updates the key pressed and returns it. The implementation of this method will be taking the time that the button is pressed, comparing it with some time in order to understand whether it is an accidental press or intentionally. If it is accidental, the method should return 0. The int value of the key pressed otherwise.

SoundEngine Class

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

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

public void changeMusic(): Stops the music that is currently playing. Starts other music.

SettingManager Class:

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

public void changeControls(int current, int new): Changes current key ID to new key ID. This will be implemented by finding the appropriate attribute which holds current ID and changing that attribute into the new key ID.

CollisionManager Class:

public boolean checkEnemySubmarineCollision(Enemy e, Submarine s): Returns true if e.getCollisionRectangle touches s.getCollisionRectangle, otherwise false.

public boolean checkEnemyBulletCollision(Enemy e, Bullet b): Returns true if e.getcollisionRectangle touches b.getcollisionRectangle, otherwise false.

public boolean checkSubmarineBulletCollision(Submarine s, Bullet b): Returns true if s.getcollisionRectangle touches b.getcollisionRectangle, otherwise false.

public boolean checkPowerUpSubmarineCollision(PowerUp pu, Submarine s): Returns true if pu.getcollisionRectangle touches s.getcollisionRectangle, otherwise false.

Map

public void update(double time): Call update(time) method of all objects in the map.

GameObject Class

private static String EXPLOSION_IMAGE_DIR: directory of explosion's image

private Image spriteImage: an instance of Image object from Javafx standard package that holds the image of the game object.

private boolean visible: when Object's x and y positions in the screen, set to true, otherwise false. All non-visible Objects move towards the screen with constant velocity, then when they are on the screen, their original velocity is restored.

public void draw(Graphics g): draws the spriteImage of object current x and y positions.

public void disappearAnimation(): Sets spriteImage to the image of explosion and velocities to 0.

public Rectangle getCollisionRectangle(): Gets width and height of the spriteImage, then creates and returns an instance of Rectangle object from JavaFX standard library.

public void update(double time): Sets xPos = xPos + xVel*time and yPos = yPos + yVel*time. Calls setVisible(true) when the object enters the map and calls disappearAnimation() when the object is destroyed.

PowerUp Class

private int quantityOfEffect: Holds the amount of energy or health that will be added to current health or energy.

private int scorePrize: Holds the amount of score that will be increased when this power-up is hit.

private int ID: holds a unique ID for each type of powerup. There are two types: first regenerates health, the second regenerates energy.

Submarine Class

This class extends GameObject class

private static String SUBMARINE_IMAGE_DIR: directory of Submarine's image

private int attackSpeed: the time between each shot.

private int attackCooldown: Initially set to attackSpeed's value, each time update(time) is called decreased by time. When the value is less than or equal to zero set to attackSpeed's value again.

private int attackDamage: the damage of the bullet released when submarine shoots.

private int amountOfProjectile: Submarine can release more than one bullet when shoots, so this variable holds the amount of bullets released at each shoot.

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

public void energyDecrease(int skillID): 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. Calls affectMassDestruction, changeAttackSpeed or affectInvulnerability methods of Skill objects according to skillID.

public int[] shoot(): (To decrease dependency between classes, instead of creating Bullet objects and returning them, this function creates an array of bullets' data and returns it, so Map creates Bullet objects according to this data.) When attackCooldown is 0, generates an array of size 2 + amountOfProjectile, the first two indexes store damage and x position, others store y positions of each bullet. Then returns this array.

public void applyPowerUp(PowerUp pu): According to the PowerUp given as parameter increases health or energy by PowerUp's quantityOfEffect.

public void update(double time): Each time changes submarine's attributes, if they are affected. Increases subLevel according to experience. Increases maxHealth, maxEnergy in Health, Energy and bulletDamage according to subLevel. Changes healtAmount, energyAmount in Health, Energy and experience when a collision occurs, according to collision type.

Enemy Class:

Abstract class that extends GameObject.

private int collisionDamage: Damage that dealt to Submarine when a collision between enemy and Submarine occurs.

private int experiencePrize: Experience points that will be gained by the player when enemy defeated.

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

SmallEnemy Class:

This class extends enemy class.

Constant Strings contain directories of SmallEnemy's images.

BigEnemy Class:

This class extends enemy class.

Constant Strings contain directories of BigEnemy's images.

private int attackSpeed: the time between each shot.

private int attackCooldown: Initially set to attackSpeed's value, each time update(time) is called decreased by time. When the value is less than or equal to zero set to attackSpeed's value again.

private int attackDamage: the damage of the bullet released when the enemy shoots.

private int amountOfProjectile: enemy can release more than one bullet when shoots, so this variable holds the amount of bullets released at each shoot.

public int[] shoot(): When attackCooldown is 0, generates an array of size 2 + amountOfProjectile, the first two indexes store damage and x position, others store y positions of each bullet. Then returns this array.

Boss Class:

This class will extend BigEnemy class.

Constant Strings contain directories of Boss's images.

private int abilityCooldown: Initially set to a constant value, each time update(time) is called decreased by time. When the value is less than or equal to zero set to constant value again.

private int quantityOfAbility: the damage of the bullet released when ability is used.

public int[] useAbility(): when the ability is used Boss releases fireball, which is Bullet object with a larger size. When abilityCooldown is 0, generates an array of size 3, that stores x, y positions and damage of the bullet and returns it.

ObjectRandomLocationManager Class:

private Location loc: holds both x and y position as a Location object from Java standard library.

public Location generateLocation(): creates two random double values for y and x and set them to x and y attributes. Then creates Location object with the created values and returns it.

CooldownManager Class:

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

private int cooldownTime: Desired cooldown time.

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:

public void update(double time): calls update(time) methods of all 3 skills.

Health Class:

private Image[] images: the array of images of health bar that represent different amount of current health from 0% to 100%.

public void update(int increaseAmount): increases/decreases healthAmount by the value of parameter but less than maxHealth and greater than 0. Then changes the image to the one that represents the new amount of health. This method is called when submarine collides with PowerUp, Enemy or Bullet.

Energy Class:

private Image[] images: the array of images of energy bar that represent different amount of current energy from 0% to 100%.

public void update(int increaseAmount): increases/decreases energyAmount by the value of parameter but less than maxEnergy and greater than 0. Then changes the image to the one that represents the new amount of energy. This method is called when submarine collides with power-up and skill is used.

Skill Class:

private int skillID: holds a unique ID for each skill: 1 for Mass Destruction, 2 for Attack Speed Booster, 3 for Invulnerability.

private int damage: holds damage done when skill #1 is used.

private double timeOfEffect: when skill #2 and #3 is used set to constant value MAX_TIME and decrease each time is updated until it is 0.

private double finalAttackSpeed: set to the value of the current attack speed of submarine when skill #2 is used in order to restore normal attack speed after the time of effect of skill #2 ends.

private boolean enabled: initially false and skill cannot be used but when submarine reached the required level it is set to true.

public void affectMassDestruction(): decrease the health of all visible enemies to damage amount.

private void changeAttackSpeed(): saves the current attack speed value in finalAttackSpeed before applying the skill, then sets the speed of Submarine by constant value when timeOfEffect is greater than 0.

private void affectInvulnerability(): when timeOfEffect greater than zero, damages done to Submarine do not decrease health.

private void update(double time): if the time of effect greater than 0 decreases timeOfEffect by the value of time.

FileManager Class:

private static final string SCORE_TXT_DIR: directory of txt file that contains high score list

public void writeToFile(String highscoreList): gets an instance of BufferedWriter from Java standard packages and using it clears data in txt file then writes String value given as a parameter to the file.

public String readFromFile(): gets an instance of BufferedReader from Java standard packages and using it gets data in the file as String and returns it.

5. Improvement Summary

In the second iteration of the design report, a lot of changes have been made. Firstly, since we added extra design patterns into the low level design, the low level design is modified based on these design patterns. Since our object design is modified, we added additional trade-offs for high level design and for object design as well. If we briefly mention what are changed:

1. We have completely changed our subsystem decomposition. Our main software architecture was MVC. However, we agreed on that 3-Layer architecture is more suitable to our system than MVC. In addition, our subsystems have changed. We have divided our system into three layers which are **User Interface Layer, Application Logic Layer,** and **File Management Layer.**
2. The UML Diagram used on the subsystem decomposition is completely changed. In order to show the relations between subsystems, we have used a **component diagram.** In brief, there is a one-way association from **User Interface Layer** into **Application Logic Layer**. In addition, there is a one-way association from **Application Logic Layer** into **File Management Layer**.

3. We decomposed our subsystems if it's possible. By possible, we considered the jobs of the components in the subsystems. Thus, we have divided User Interface Layer into two components, which are Input Management and Screen. In addition, we have divided Application Logic Layer into two components, which are Game Management and Game Entities. The components' explanations are briefly given on above.
4. We have added some methods and attributes which are necessary because of either the object design or the design pattern.
5. We have deleted some methods and attributes because they were redundant.
6. We added some extra design patterns which are **Singleton and Delegation Pattern**. These design patterns are included because they will make the implementation easier.
7. From associations to the contents of the classes, our object design is updated.
8. Class interfaces are updated in order to give further information about the important classes.

6. References

[1] Java (programming language). (2018, April 17). Retrieved April 17, 2018, from 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.*