Design Report for <<Weather Space Shooter>>

Name: Tien Duc Nguyen
Student ID: 103174526

COS20007 – Object Oriented Programming

Table of content:

I. Abstract

This report is designed to provide short and concise insight into the implementation of the design pattern as well as other APIs of Weather Spaceshooter.

Weather SpaceShooter is a redo of conventional Space Shooter game. In this game, player can choose different kind of ship to battle. Each ship has specific appearance and skills that could help player destroy obstacles and gain as much point as possible.

Moreover, if you are the one who care about what happen outside but still want to play game, then this game was designed for you. As you can see in figure 2, on the top left, there are information about the weather, and the game background also can change according to the weather.

Figure 2.

II. Required Roles

Roles	Responsibility	Notes
GameObject	Draw Objects Return co-ordinate(x, y) of objects Return sprite and bitmap of objects	All game objects like spaceship and obstacles are inherited from this class
SpaceShip	Update and using skills Update move (player will moving ship by mouse)	There are 2 types of spaceships
Obstacle	Move Check if there is collision or not Having effect on spaceship when there is a collision	This will affect the ship when there is a collision
Skills	Performing effect on spaceship	Each type of spaceship will have different list of skills
Fire	Move Collide with other objects	
Skills	Use the skills Set how the skills perform	
Obstacle Factory	Get Obstacle	
GameMain	Add Objects Add Fire Check if an object has to be deleted or not Check collision between objects	This is where almost all the game features will be added and interact with each others
SpaceShooter	NextState Run	This class responsible for user interface in each state of the game (MainMenu, Tutorial, Playing, GameOver)
GameState	Change State Update state	This is an abstract class for all game states
FireState	Change state Update Move Update Image	This is an abstract class for all game states(ie. FireLeft, FireRight, FireUp, FireDown)

III. Top Level Overview

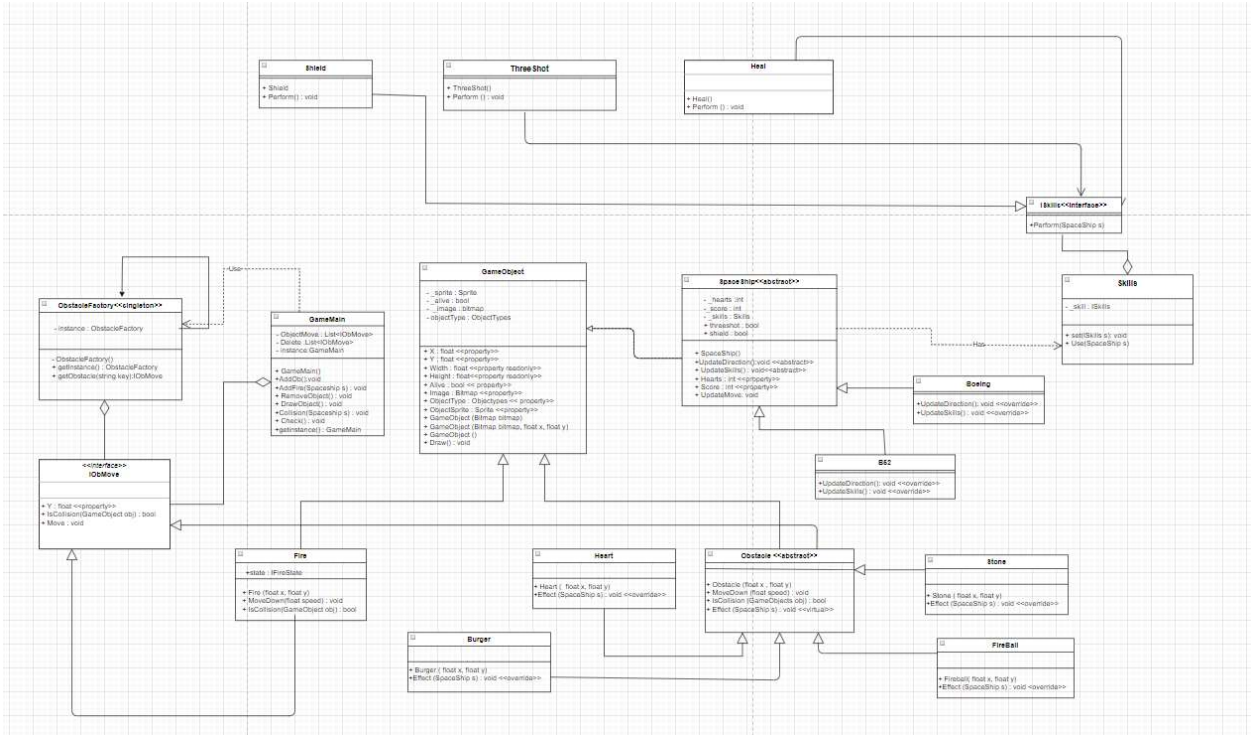
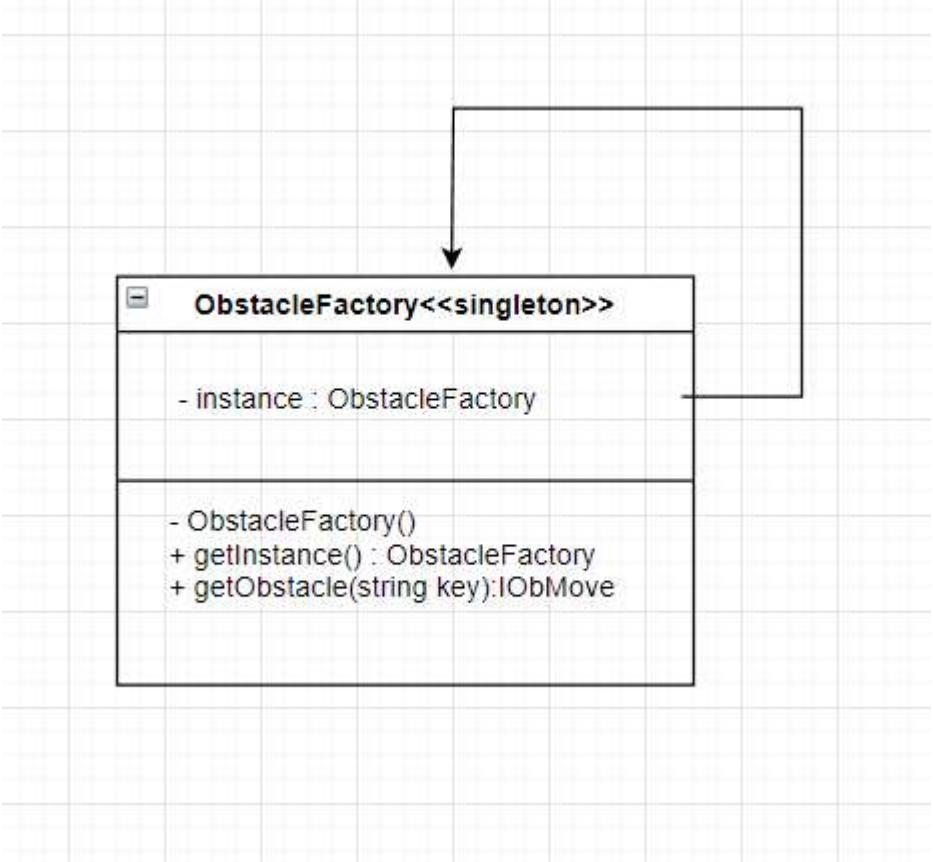


Figure 2.

As you can see on the UML diagram (Figure 2.), there are two big classes include GameObjects and GameMain.
GameObjects is the base of all game objects such as Fire, SpaceShip and Obstacles.
GameMain is where all objects interact with each other (it acts like the Deck in live online class's example) it is an aggregation of IObMove (an interface that was implemented by Obstacles and Fire). This class is where Objects will be added and deleted according to what happen in the game.

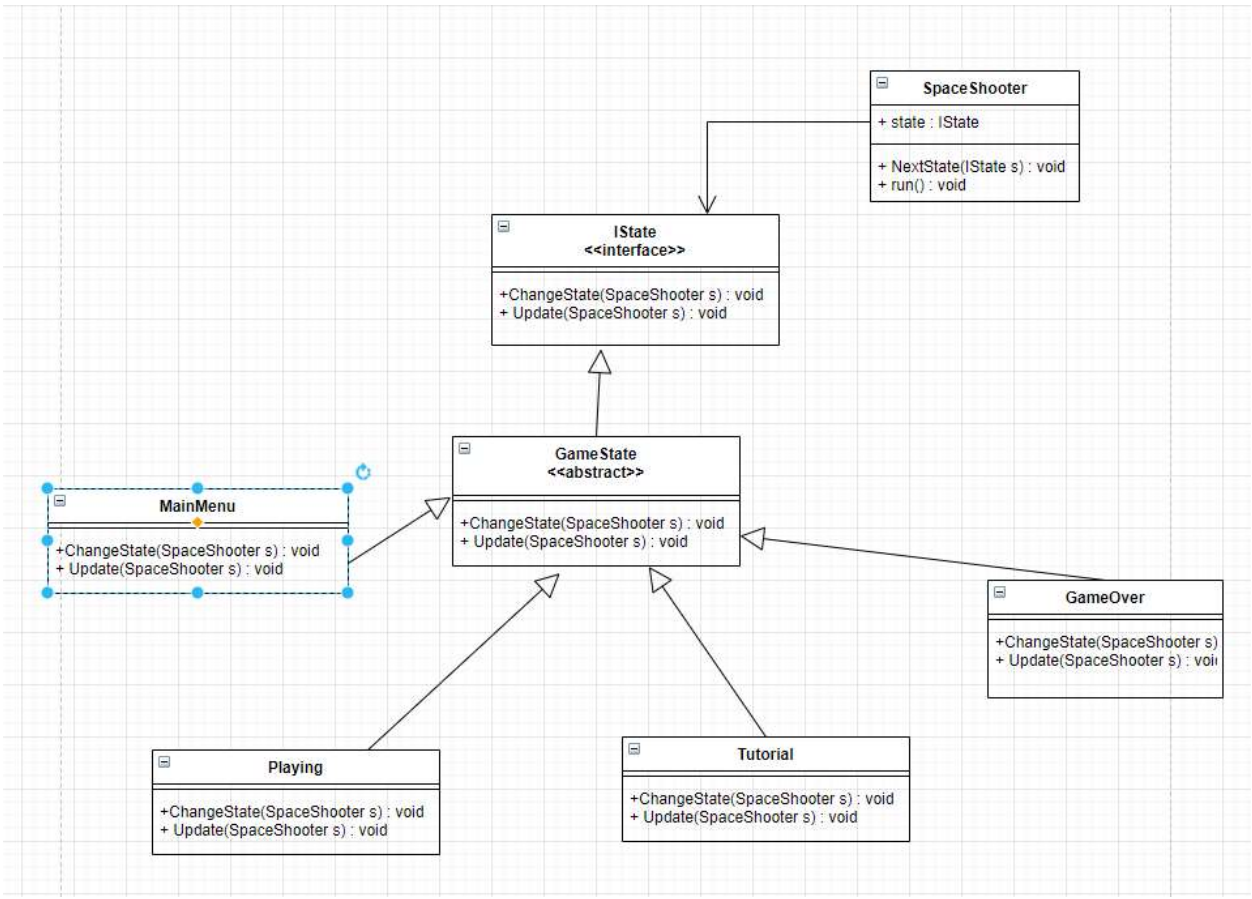
IV. Design Pattern

1. ObstacleFatory – Singleton pattern



ObstacleFactory is where obstacles instantiate, and there is only one ObstacleFactory in this program. Therefore, by using singleton pattern ensures the same instance can be used from everywhere.

2. Game State and Fire – State Pattern
   To manage the transition between each state of the game. I have implemented State Pattern. There will be 4 main states including: MainMenu, Tutorial, Playing, GameOver(Figure 3.). This state pattern helps to reduce the duplication of code as well as "if" statement when we want to change the state. This design pattern also apply to Fire when they help to change movement and image direction according to the direction of SpaceShip.


SpaceShooter
+ state : IState
+ NextState(IState s) : void
+ run() : void
IState
<<interface>>
+ChangeState(SpaceShooter s) : void
+ Update(SpaceShooter s) : void
GameState
<<abstract>>
+ChangeState(SpaceShooter s) : void
+ Update(SpaceShooter s) : void
MainMenu
+ChangeState(SpaceShooter s) : void
+ Update(SpaceShooter s) : void
GameOver
+ChangeState(SpaceShooter s)
+ Update(SpaceShooter s) : voi
Playing
+ChangeState(SpaceShooter s) : void
+ Update(SpaceShooter s) : void
Tutorial
+ChangeState(SpaceShooter s) : void
+ Update(SpaceShooter s) : void
Figure 3.

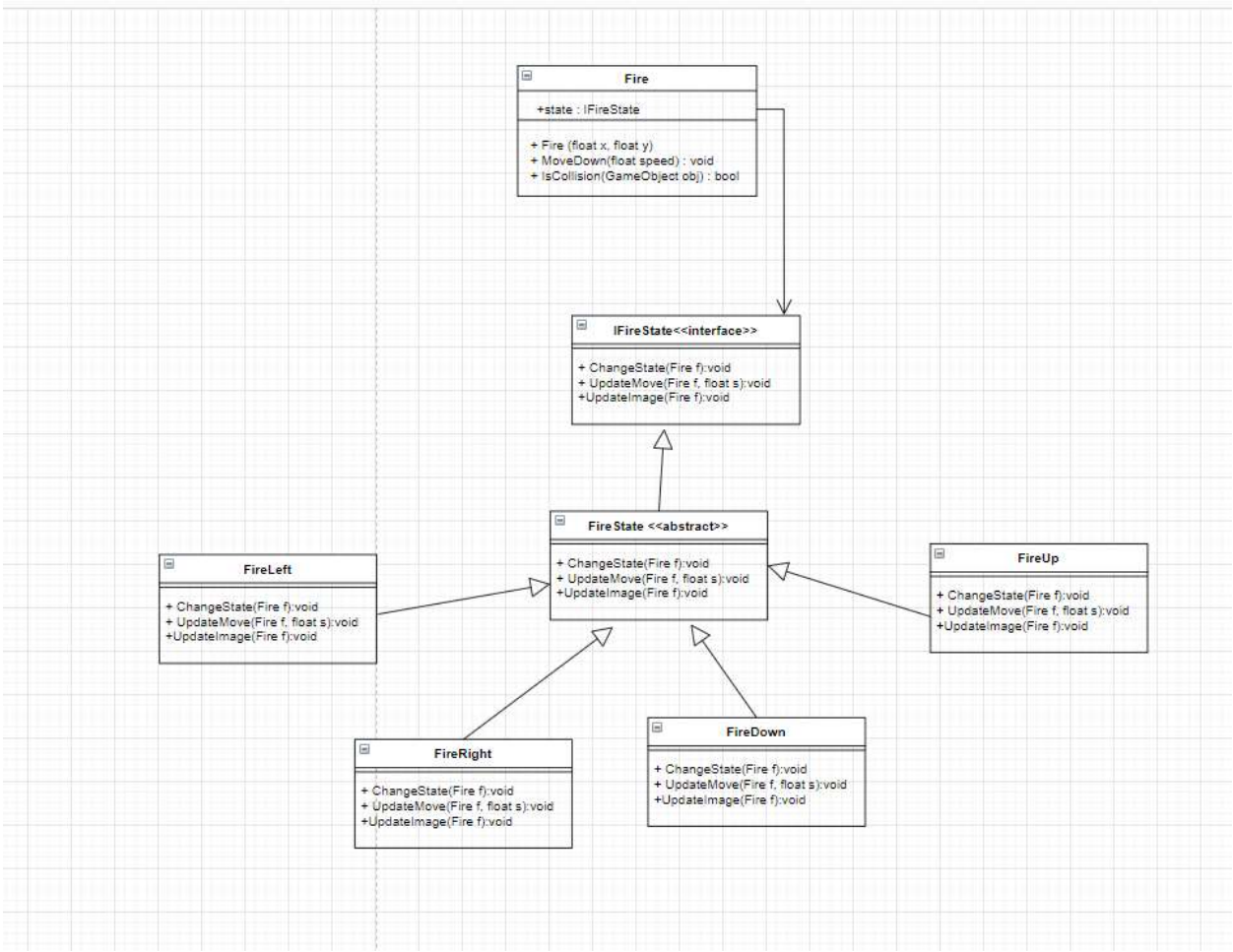


Figure 3.1

3. Obstacles – Factory pattern

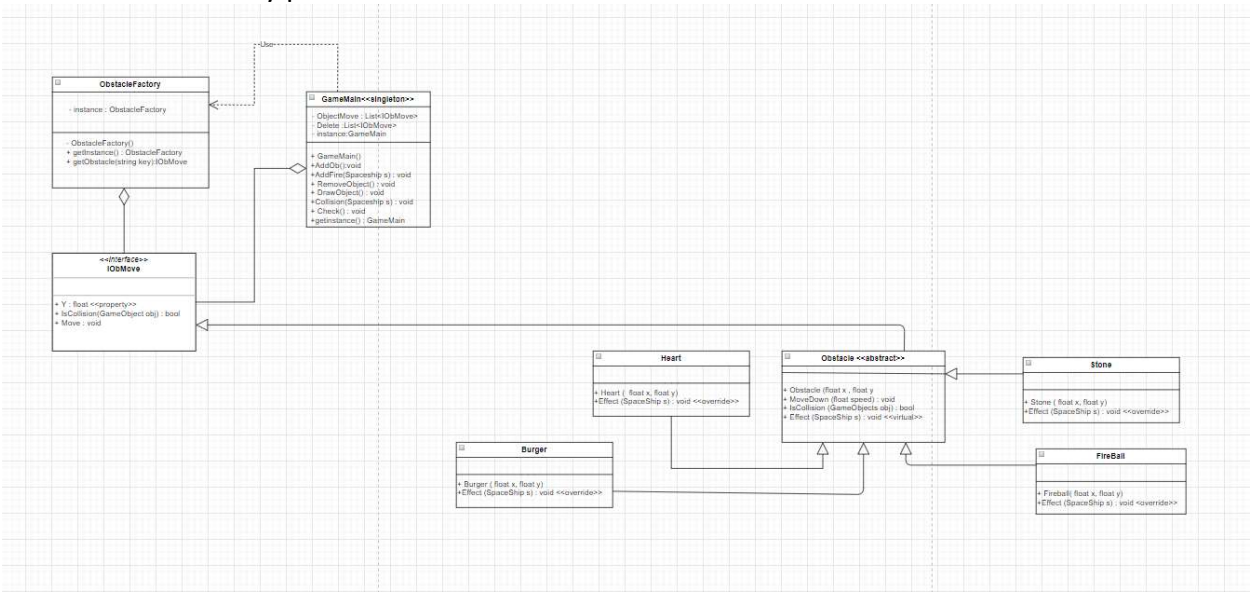


Figure 4.

```
public void AddObject()
{
    ObstacleFactory f = new ObstacleFactory();
    string[] s = new string[] { "burger", "stone", "heart", "fireball" };
    for (int i = 0; i < s.Length; i++)
    {
        if (SplashKit.Rnd(0, 1000) % 100 == 0)
        {
            IObMove burger = f.GetObstacles(s[i]);
            ObjectMove.Add(burger);
        }
    }
}
```

Figure 5.1.

```
public void AddObject()
{
    if (SplashKit.Rnd(0, 1000) % 100 == 0)
    {
        Burger burger = new Burger(SplashKit.Rnd(0, 500), 0);
        ObjectMove.Add(burger);
    }
    if (SplashKit.Rnd(0, 1000) % 100 == 0)
    {
        Fireball burger = new Fireball(SplashKit.Rnd(0, 500), 0);
        ObjectMove.Add(burger);
    }
    if (SplashKit.Rnd(0, 1000) % 100 == 0)
    {
        Stone burger = new Stone(SplashKit.Rnd(0, 500), 0);
        ObjectMove.Add(burger);
    }
    if (SplashKit.Rnd(0, 1000) % 100 == 0)
    {
        Heart burger = new Heart(SplashKit.Rnd(0, 500), 0);
        ObjectMove.Add(burger);
    }
}
```

Figure 5.2.

In game main we need to generate a number of obstacles, but instead of creating it directly using new operator, the game main will ask the Obstacle Factory for a new obstacles by providing the name of Obstacles. Then Obstacle factory will instantiate it and return to the game main.

Using this pattern in this situation help me reduce a lot of duplication code. Instead of calling four new operators, I just need a loop to create 4 obstacles (figure 5.1 and 5.2).

4. Skills – Strategy pattern

All sub-classes of Skills in this situation differ only in he way they perform. Therefore, with this case, I think is a good idea to isolate the algorithms of perform method in separate class in order to have the ability to select different skills when the game is running. For that

reason, strategy pattern will be a good approach. You can see the overview of strategy pattern in the diagram (Figure 6)

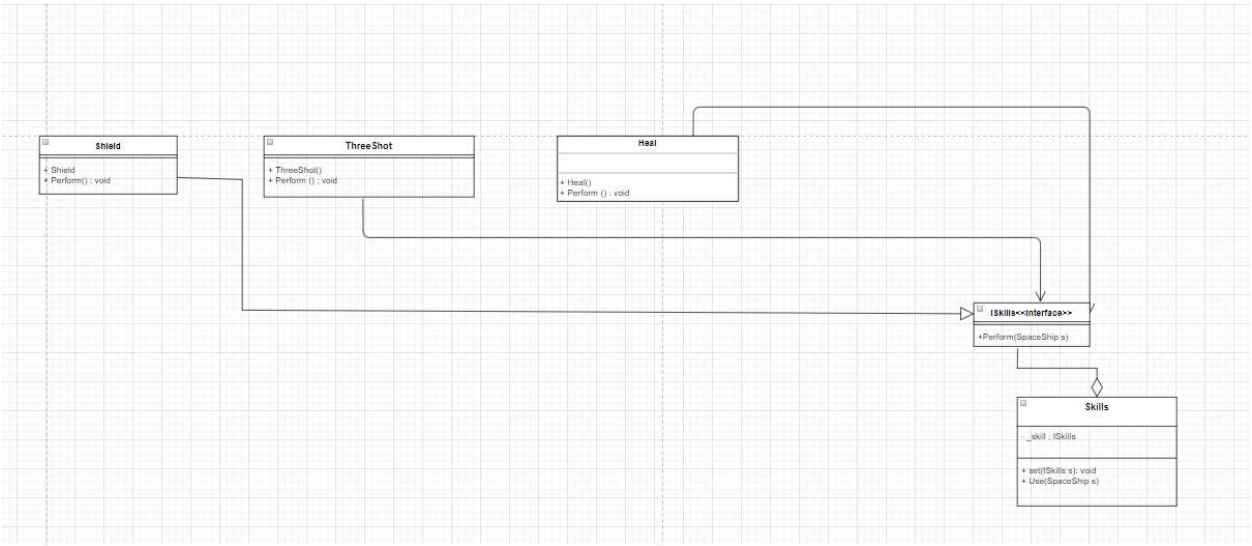


Figure 6.

V. APIs

```
public void getWeather()
{
    using (WebClient web = new WebClient())
    {
        string url = string.Format("http://api.openweathermap.org/data/2.5/weather?q={0}&appid={1}", city, APIkey);
        var json = web.DownloadString(url);
        var result = JsonConvert.DeserializeObject<WeatherInfo.root>(json);
        WeatherInfo.root Info = result;
        this.Cond = Info.weather[0].main;
        this.Temp = Info.main.temp.ToString();
    }
}
```

Figure 6.

```
public class WeatherInfo
{
    1 reference
    public class weather
    {
        1 reference
        public string main { get; set; }

    }

    1 reference
    public class main
    {
        1 reference
        public double temp { get; set; }

    }

    2 references
    public class root
    {
        1 reference
        public List<weather> weather { get; set; }
        1 reference
        public main main { get; set; }

    }
}
```

Figure 7.

The APIs that I use in this project is open weather map. It was implemented by using WebClient to download a string in JSON format, then it will be deserialize (need to implement Newton.Soft to deserialize the JSON docs) according to the root we create. In this case, I want information about main and temp (Figure 7.)

VI. Sequence Diagram

This is the sequence diagram of the process from adding Fire to adding score to SpaceShip. After player press backspace, the GameMain will instantiate Fire, then if Fire collide with Obstacles, SpaceShip will get a score and GameMain will delete the Obstacle.