

HIGH DISTINCTION CUSTOM PROGRAM REPORT

COS20007

XUAN DAT LE - 103487949



Contents

I. Abstract.....	3
II. UML Diagram Overview	4
III. OOP Core Concepts Implementation.....	5
1. <i>Inheritance</i>	6
2. <i>Polymorphism</i>	9
3. Abstraction & Encapsulation.....	10
IV. Design Pattern.....	12
1. Factory Method Pattern.....	13
2. State Pattern.....	16
3. Strategy Pattern.....	17
V. Gameplay Walkthrough	18
1. Menu Stage & Instruction Stage.....	18
2. Main Game Stage.....	20
3. Game Over & Game Win.....	21
VI. Report Summary.....	22

I. Abstract

This final report presents the development process and features of a top-down shooting game implemented using the SplashKit library in the C# programming language. The objective of the project was to create an engaging and enjoyable game with an object-oriented approach, incorporating various design patterns and adhering to the principles of good software design.

The game boasts several notable features, including a dynamic and immersive gameplay experience. Players take control of a protagonist who navigates through a hostile environment, engaging in intense combat with a variety of enemy characters. The game includes multiple levels, each presenting unique challenges and objectives for the player to overcome. To enhance the gaming experience, various power-ups and upgrades are strategically placed throughout the levels.

This report explains briefly about the UML diagram, which outlines the class hierarchy and relationships among various components of the game, including the player, enemies, weapons, and power-ups, etc. The UML diagram provides a visual representation of the system's architecture, facilitating a better understanding of the project's structure and aiding in the development and maintenance process.

Throughout the development process, several design patterns were employed to promote code reusability, extensibility, and maintainability, which would be further elaborated in this report.

The combination of these design patterns, along with the proper utilization of the SplashKit library, resulted in a well-structured and engaging top-down shooting game. The project serves as an example of how object-oriented design principles, UML diagrams, and various design patterns can be effectively applied to create complex and enjoyable software applications.



Figure 1: Gameplay image

In the following sections, we will explore the different components of the UML diagram in detail, explaining the relationships between the classes and their roles in the game. This overview aims to provide a comprehensive understanding of the game's architecture, aiding in the analysis, development, and maintenance of the top-down shooting game implemented using the SplashKit library in C#

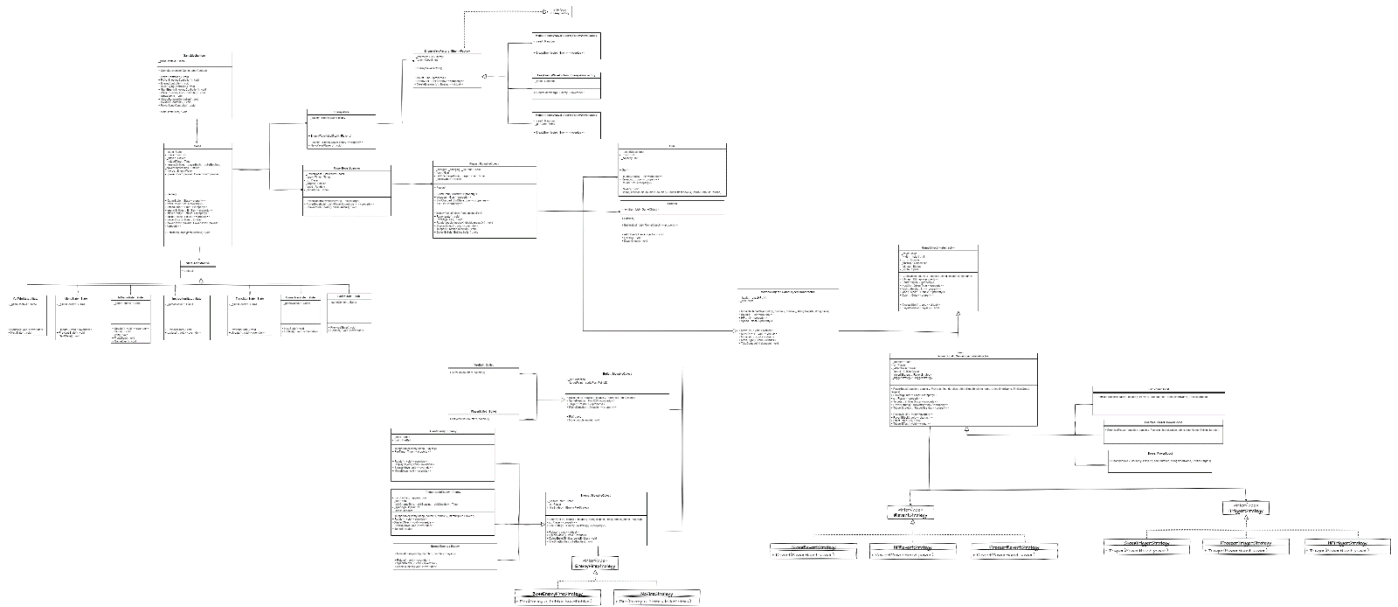


Figure 2: UML Diagram Overview

III. OOP Core Concepts Implementation

The implementation of the four core concepts of Object-Oriented Programming (OOP) is fundamental to the development of robust and scalable software systems. In this section, I will explain how these core concepts, namely encapsulation, inheritance, polymorphism, and abstraction, have been successfully implemented in my top-down shooting game created using the SplashKit library in C#.

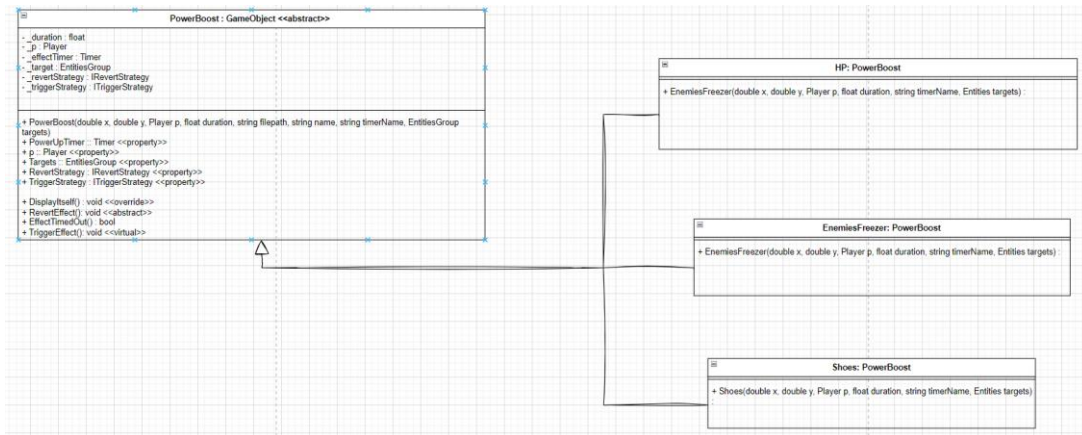


Figure 3: Inheritance example from the UML diagram

```

namespace MyGame
{
    29 references
    public class Player : MovableObject
    {
        private bool _charged;
        private bool _charging;
        private Gun _gun;
        private int _killCount;
        SplashKitSDK.Timer _skillRechargeTimer;
        SplashKitSDK.Timer _regenTimer;
        private Entities _bulletGroup = new Entities();

        1 reference
        public Player() : base(999, 480, 380, "rsz_hero.png", "hero")
        {
            Weapon = new Gun();
            Speed = 2;
            _regenTimer = new SplashKitSDK.Timer("Regen Timer");
            _skillRechargeTimer = new SplashKitSDK.Timer("Skill Recharge Timer");
            _regenTimer.Start();
            SkillCharged = true;
            SkillState = false;
            Kill = 0;
        }
    }
}
  
```

```

namespace MyGame
{
    14 references
    public abstract class GameObject
    {
        private Color _color;
        private int _width, _height;
        private double _x, _y;
        private ObjectType _hostility;
        private Bitmap _bitmap;
        private Sprite _sprite;
        private SplashKitSDK.Timer _spawnTimer;

        3 references
        public GameObject(double x, double y, string filepath, string name)
        {
            _spawnTimer = new SplashKitSDK.Timer(new Random().Next().ToString());
            ModX = x;
            ModY = y;
            Hostility = ObjectType.neutral;
            Bitmap = SplashKit.LoadBitmap(name, filepath);
            Sprite = SplashKit.CreateSprite(Bitmap);
        }
    }
}
  
```

Figure 3&4: Encapsulation and Abstraction code example

1. Inheritance

Inheritance, one of the core concepts of Object-Oriented Programming (OOP), played a pivotal role in the implementation of my top-down shooting game using the SplashKit library in C#. By leveraging inheritance, I established a hierarchical relationship among classes, enabling code reuse, extensibility, and efficient organization of related objects.

Consider the coding example from my program:

```
namespace MyGame
{
    30 references
    public class Enemy : MovableObject
    {
        private SplashKitSDK.Timer _fireTimer;
        private Player _p;
        private Blood _blood;
        public IEnemyFireStrategy _fireStrategy;
        4 references
        public Enemy(int hp, double x, double y, string filepath, string zombie_name, Player p) : base(hp, 0, 0, filepath, zombie_name)
        {
            Random _random = new Random();
            int i = _random.Next();
            _fireTimer = new SplashKitSDK.Timer(_random.Next().ToString());
            ModX = x;
            ModY = y;
            SpawnTimer.Start();
            FireTimer.Start();
            _p = p;
            FireStrategy = new NoFireStrategy();
        }

        1 reference
        public virtual void Rotate()
        {
            double angle = Math.Atan2(p.ModY - ModY, p.ModX - ModX);
            Sprite.Rotation = (float)(angle * (180.0 / 3.14));
        }

        2 references
        public virtual void SpecialMove()
        {
            if (ModX < p.ModX)
            {
                MoveRight();
            }
            if (ModX > p.ModX)
            {
                MoveLeft();
            }
            if (ModY < p.ModY)
            {
                MoveDown();
            }
            if (ModY > p.ModY)
            {
                MoveUp();
            }
        }
    }
}
```

Figure 5 & 6: Enemy Class

The “Enemy” class in the coding image above contains common movement attributes such as “Rotate” and “SpecialMove”, as well as some other common behaviors that defines an enemy. These attributes and behaviors are shared by all types of enemies, so these lines of code are placed in the base class to avoid code duplication.

```
namespace MyGame
{
    4 references
    public class CovidEnemy : Enemy
    {
        3 references
        public CovidEnemy(int hp, double x, double y, Player p) : base(hp, x, y, "covid_zombie.png", "covid zombie", p)
        {
            Speed = (float)0.7;
        }
        8 references
        public override void DisplayObject()
        {
            SplashKit.FillRectangle(Color.Red, ModX + 25, ModY - 15, 30, 5);
            //current HP
            SplashKit.FillRectangle(Color.Blue, ModX + 25, ModY - 15, 30 / BaseHP * HP, 5);
            if (SpawnTimer.Ticks > 3000)
            {
                if (Hostility == ObjectType.neutral)
                    Hostility = ObjectType.hostile;
                SpawnTimer.Pause();
                base.DisplayObject();
            }
            else
            {
                SplashKit.FillRectangle(Color.BlueViolet, ModX, ModY, 40, 40);
            }
        }
    }
}
```

Figure 7: CovidEnemy inherits from its father class.

```
7 references
public class NormalEnemy : Enemy
{
    6 references
    public NormalEnemy(int hp, double x, double y, Player p) : base(hp, x, y, "rsz_zombie.png", "normal zombie", p)
    {
        Speed = (float)0.5;
    }
    8 references
    public override void DisplayObject()
    {
        SplashKit.FillRectangle(Color.Red, ModX + 25, ModY - 15, 25, 5);
        //current HP
        SplashKit.FillRectangle(Color.Blue, ModX + 25, ModY - 15, 25 / BaseHP * HP, 5);
        if (SpawnTimer.Ticks > 3000)
        {
            if (Hostility == ObjectType.neutral)
                Hostility = ObjectType.hostile;
            SpawnTimer.Pause();
            base.DisplayObject();
        }
        else
        {
            SplashKit.FillRectangle(Color.Orange, ModX, ModY, 40, 40);
        }
    }
}
```

Figure 8: CovidEnemy inherits from its father class.

The “CovidEnemy” class inherits from “Enemy” class using the “: base ()” syntax in its constructor. This allows the class itself to access the attributes and methods of the base class. Similarly, the “NormalEnemy” class also inherits from the father “Enemy” class.

By utilizing inheritance, I can create instances of the “CovidEnemy” and “NormalEnemy” classes that can access the common attributes and behaviors derived from the base class. For example:

```
Player p = new();
NormalEnemy exampleNormalEnemy = new NormalEnemy(10, 0, 0, p);
CovidEnemy exampleCovidEnemy = new CovidEnemy(15, 0, 0, p);

exampleNormalEnemy.SpecialMove();
exampleCovidEnemy.SpecialMove();
```

Figure 9: Example of “CovidEnemy” & “NormalEnemy” inheritance

The child class can also extend their methods, if necessary, in my case, the image below can illustrate this point:

```
namespace MyGame
{
    3 references
    public class TeleportableEnemy : Enemy
    {
        private bool _cordLocked;
        private bool _idle;
        private Point2D _coordinate;
        private SplashKitSDK.Timer _skillChargeTime;
        private SplashKitSDK.Timer _skillSpacing;
        private int _spacingMultiplier;
        private Random _seed;

        2 references
        public TeleportableEnemy(int hp, double x, double y, int multiplier, Player p) : base(hp, x, y, "tele.png", "teleportable enemy", p)
        {
            _seed = new Random();
            _skillChargeTime = new SplashKitSDK.Timer("Skill Charge Time " + _seed.Next().ToString());
            _skillSpacing = new SplashKitSDK.Timer("Skill Spacing Time " + _seed.Next().ToString());
            _cordLocked = false;
            _idle = true;
            _spacingMultiplier = multiplier;
            _skillChargeTime.Start();
            _skillSpacing.Start();
        }

        0 references
        public void Jump(int state)
        {
            switch (state)
            {
                case 1:
                    SplashKit.DrawRectangle(Color.DarkGreen, p.ModX - 6, p.ModY - 6, p.Width + 20, p.Height + 20);
                    break;
                case 2:
                    SplashKit.DrawRectangle(Color.Orange, p.ModX - 4, p.ModY - 4, p.Width + 12, p.Height + 12);
                    break;
                case 3:
                    SplashKit.DrawRectangle(Color.Red, _coordinate.X - 2, _coordinate.Y - 2, p.Width + 8, p.Height + 8);
                    break;
                case 4:
                    ModX = (int)_coordinate.X;
                    ModY = (int)_coordinate.Y;
                    _skillChargeTime.Stop();
                    _skillChargeTime.Reset();
                    _skillChargeTime.Start();
                    break;
            }
        }
    }
}
```

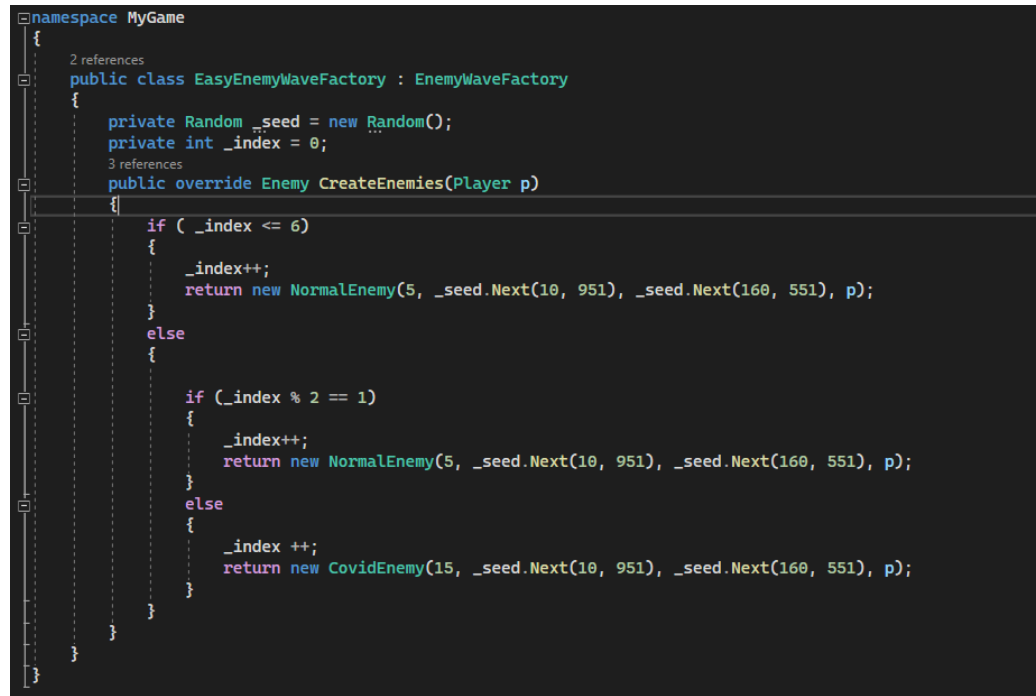
Figure 10: Child class extends method.

The “TeleportableEnemy”, besides the fact that it can access all the common attributes and behaviors from the “Enemy” class, the class itself also broaden their feature with the “Jump” method, because of its unique moving mechanism.

Through inheritance, I achieve code reuse by defining common attributes and behaviors in the base class and extending them in the derived classes. This approach simplifies the codebase, promotes maintainability, and allows for easy addition of new entity types or variations in the future.

2. Polymorphism

Take the coding image extracted from my program below as an example:



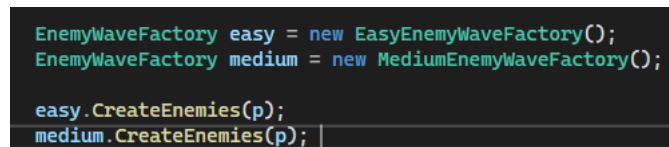
```

namespace MyGame
{
    2 references
    public class EasyEnemyWaveFactory : EnemyWaveFactory
    {
        private Random _seed = new Random();
        private int _index = 0;
        3 references
        public override Enemy CreateEnemies(Player p)
        {
            if ( _index <= 6)
            {
                _index++;
                return new NormalEnemy(5, _seed.Next(10, 951), _seed.Next(160, 551), p);
            }
            else
            {
                if ( _index % 2 == 1)
                {
                    _index++;
                    return new NormalEnemy(5, _seed.Next(10, 951), _seed.Next(160, 551), p);
                }
                else
                {
                    _index ++;
                    return new CovidEnemy(15, _seed.Next(10, 951), _seed.Next(160, 551), p);
                }
            }
        }
    }
}

```

Figure 11: Polymorphism overview

In the above code, the Abstract class “EnemyWaveFactory” class has a abstract method “CreateEnemies(Player p)”, indicating that it would be overridden in the derived class, which in this case, is the “EasyEnemyWaveFactory” class.



```

EnemyWaveFactory easy = new EasyEnemyWaveFactory();
EnemyWaveFactory medium = new MediumEnemyWaveFactory();

easy.CreateEnemies(p);
medium.CreateEnemies(p);

```

Figure 12: Implementation Example

In the example above, I create instances of “EasyEnemyWaveFactory” and “MediumEnemyWaveFactory” objects and assign them to variables of type “EnemyWaveFactory”.

When the “CreateEnemies” method is called, despite the variables being declared as “EnemyWaveFactory”, the actual behavior is determined by the type of object they reference.

Polymorphism enables code to be written in a more generic and reusable manner, as it allows us to work with objects at a higher level of abstraction. It simplifies code maintenance and promotes extensibility by facilitating the addition of new derived classes that can be used interchangeably with existing base classes.

3. Abstraction & Encapsulation

```

namespace MyGame
{
    8 references
    public abstract class MovableObject : GameObject
    {
        private int _health, _baseHP;
        private float _spd;

        3 references
        public MovableObject(int hp, double x, double y, string filepath, string name) : base(x, y, filepath, name)
        {
            HP = hp;
            _baseHP = hp;
            Speed = 1;
        }
    }
}

```

Figure 13: Abstraction coding example from my program

```

namespace MyGame
{
    31 references
    public class Player : MovableObject
    {
        private bool _charged;
        private bool _charging;
        private Gun _gun;
        private int _killCount;
        SplashKitSDK.Timer _skillRechargeTimer;
        SplashKitSDK.Timer _regenTimer;
        private Entities _bulletGroup = new Entities();

        2 references
        public Player() : base(999, 480, 300, "rsz_hero.png", "hero")
        {
            Weapon = new Gun();
            Speed = 2;
            _regenTimer = new SplashKitSDK.Timer("Regen Timer");
            _skillRechargeTimer = new SplashKitSDK.Timer("Skill Recharge Timer");
            _regenTimer.Start();
            SkillCharged = true;
            SkillState = false;
            Kill = 0;
        }
    }
    0 references
}

```

Figure 14: Player class inherit from abstract class MovableObject

In the example above, we have an abstract class called “MovableObject” which represents a generic Game object that can move, it serves as a template of Movable Object, which later be implemented and broaden by child class, in this case: the “Player” class.

Encapsulation is explained within the “Player” class as well, as all the attributes are encapsulated within the class itself and can only be access by the instance of the respective class.

```
1 reference
public void Regenerate()
{
    //player HP regen (every 5 sec restores 1HP out of 20HP)
    if (_regenTimer.Ticks >= 5000 && HP > 0)
    {
        if (HP < BaseHP)
            HP++;
        _regenTimer.Stop();
        _regenTimer.Start();
    }
}

1 reference
public void RechargeSkill()
{
    SkillCharged = true;
}
```

Figure 15: Encapsulation demonstration

IV. Design Pattern

Design patterns provide proven solutions to recurring design problems in software development. They serve as templates that guide developers in structuring their code to achieve flexibility, maintainability, and extensibility. In this section, we will explore the implementation of design patterns in my top-down shooting game, which leverages the SplashKit library in C#.

In my program, I applied three design patterns: the State Pattern, Factory Method Pattern, and Strategy Pattern. Each of these patterns addresses specific design challenges, offering elegant solutions to enhance the overall architecture and functionality of the game.

Throughout this section, I will delve into the implementation details of each design pattern, providing examples and explanations of how they were applied in the context of my top-down shooting game. By understanding the utilization of these patterns, fellow developers can gain insights into the architectural decisions made and the benefits derived from their application. Exploring the State Pattern, Factory Method Pattern, and Strategy Pattern will shed light on the thought process behind the design choices and demonstrate how design patterns can elevate the quality and structure of a game's codebase.

1. Factory Method Pattern

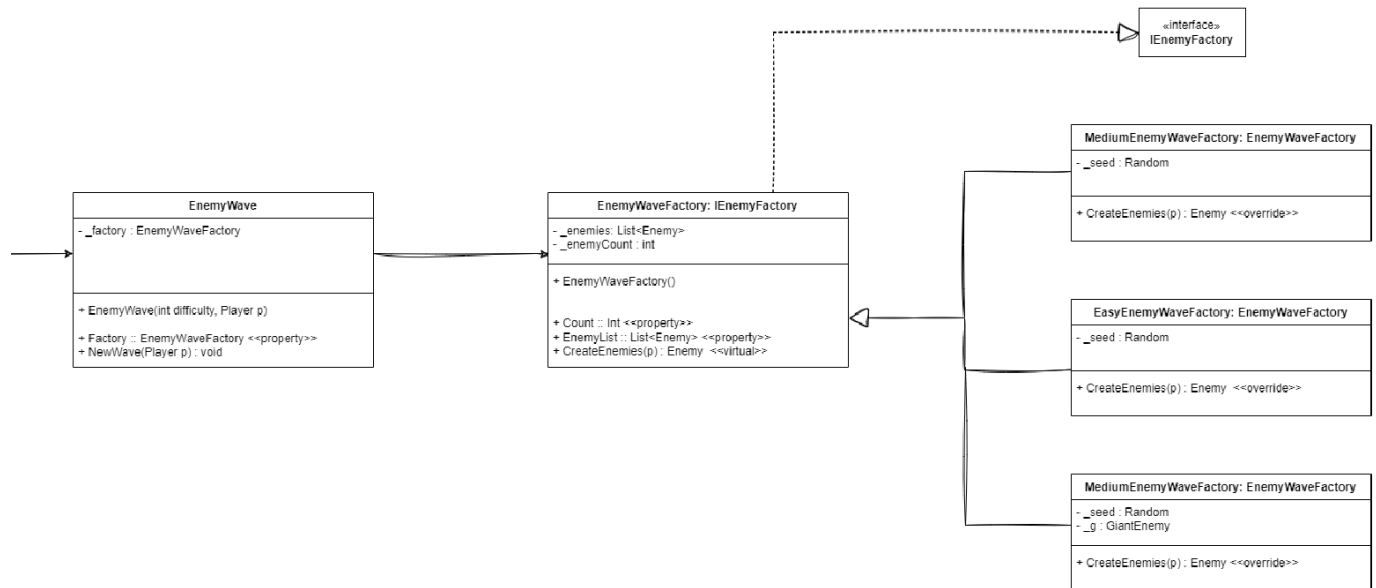


Figure 16: UML Diagram of the Factory Method Design Pattern

The EasyWaveFactory focused on creating enemy hordes suitable for the game's easy level. It added two types of enemies: the normal enemy and the covid enemy. By employing the Factory Method pattern, the EasyWaveFactory encapsulated the logic of creating these enemy types, ensuring consistent and centralized object creation.

```

namespace MyGame
{
    3 references
    public class EasyEnemyWaveFactory : EnemyWaveFactory
    {
        private Random _seed = new Random();
        private int _index = 0;
        5 references
        public override Enemy CreateEnemies(Player p)
        {
            if (_index <= 6)
            {
                _index++;
                return new NormalEnemy(5, _seed.Next(10, 951), _seed.Next(160, 551), p);
            }
            else
            {
                if (_index % 2 == 1)
                {
                    _index++;
                    return new NormalEnemy(5, _seed.Next(10, 951), _seed.Next(160, 551), p);
                }
                else
                {
                    _index++;
                    return new CovidEnemy(15, _seed.Next(10, 951), _seed.Next(160, 551), p);
                }
            }
        }
    }
}

```

Figure 17: EasyEnemyWaveFactory code

For the medium level of the game, I implemented the MediumWaveFactory. This factory introduced an additional enemy type called the teleportable enemy, along with the normal enemy and the covid enemy. By employing the Factory Method pattern, the MediumWaveFactory provided a flexible mechanism to

create random combinations of these enemy types for the medium level waves.

```

namespace MyGame
{
    2 references
    public class MediumEnemyWaveFactory : EnemyWaveFactory
    {
        private Random _seed = new Random();
        private int _index = 0;
        5 references
        public override Enemy CreateEnemies(Player p)
        {
            if (_index <= 3)
            {
                _index++;
                return new NormalEnemy(5, _seed.Next(10, 951), _seed.Next(160, 551), p);
            }
            else
            {
                Random rand = new Random();

                int indexRand = rand.Next(0, 3);
                switch (indexRand)
                {
                    case 0:
                        return new NormalEnemy(5, _seed.Next(10, 951), _seed.Next(160, 551), p);
                    case 1:
                        return new CovidEnemy(15, _seed.Next(10, 951), _seed.Next(160, 551), p);
                    case 2:
                        return new TeleportableEnemy(10, _seed.Next(10, 951), _seed.Next(160, 551), rand.Next(4, 10), p);
                    default:
                        return new NormalEnemy(5, _seed.Next(10, 951), _seed.Next(160, 551), p);
                }
            }
        }
    }
}

```

Figure 18: MediumEnemyWaveFactory

To challenge experienced players at the hard level, I utilized the HardWaveFactory. This factory encompassed all four previously mentioned enemy types: the normal enemy, the covid enemy, the teleportable enemy, and a formidable addition, the Giant Boss enemy. The Giant Boss enemy possessed special abilities, including the ability to fire projectiles, making it a significant challenge for players. Through the Factory Method pattern, the HardWaveFactory orchestrated the creation of these diverse enemy types, enriching the gameplay experience for advanced players.

```

namespace MyGame
{
    2 references
    public class HardEnemyWaveFactory : EnemyWaveFactory
    {
        private Random _seed = new Random();
        private int _index = 0;
        1 reference
        public HardEnemyWaveFactory()
        {
        }

        5 references
        public override Enemy CreateEnemies(Player p)
        {
            int index = _index % 5;

            if (index == 0)
            {
                _index++;
                return new GiantEnemy(30, p);
            }
            else if (index <= 3 && index > 0)
            {
                _index++;
                return new NormalEnemy(5, _seed.Next(10, 951), _seed.Next(160, 551), p);
            }
            else if (index == 3)
            {
                _index++;
                return new TeleportableEnemy(10, _seed.Next(10, 951), _seed.Next(160, 551), _seed.Next(4, 10), p);
            }
            else
            {
                _index++;
                return new CovidEnemy(15, _seed.Next(10, 951), _seed.Next(160, 551), p);
            }
        }
    }
}

```

Figure 19: *HardEnemyWaveFactory* class

In the “EnemyWave” is where the factory classes are implemented, depending on the difficulty level, the Factory property of the “EnemyWave” would be assigned to different type of factory:

```

6 references
public class EnemyWave
{
    private EnemyWaveFactory _factory;
    private List<Enemy> _enemies;
    private int _enemyCount = 1;
    private int _diff;
    3 references
    public EnemyWave(int difficulty, Player p)
    {
        EnemyList = new List<Enemy>();
        _diff = difficulty;
        switch (_diff)
        {
            case 1:
                Factory = new EasyEnemyWaveFactory();
                break;
            case 2:
                Factory = new MediumEnemyWaveFactory();
                break;
            case 3:
                Factory = new HardEnemyWaveFactory();
                break;
            default:
                Factory = new EasyEnemyWaveFactory();
                break;
        }
        NewWave(p);
    }
}

```

Figure 20: *Factory Method* implementation

Overall, the Factory Method Pattern played a crucial role in the creation of game objects. By abstracting the object creation process into a factory method, I achieved decoupling between the client code and the actual object creation. This pattern enabled me to extend the game with new types of objects without modifying the existing client code, promoting code reusability, and ensuring that the creation logic remains centralized and consistent.

V. Gameplay Walkthrough

1. Menu Stage & Instruction Stage



SHOOTER

Press H to open up the tutorial

This game has 3 difficulties - Easy, Normal or Hard.

Press 1 [Easy], 2 [Normal] or 3 [Hard] to start!

Figure 23: Menu Stage when the game start

If user presses the H key, the Instruction screen will pop up:

In this game, you are a black block.

Basically you just shoot anything that's not you.

If you touch zombies, you lose HP equal to theirs

ESC - Pause

W A S D - Move

Space Key - shoot at a direction created by Mouse Position

R - reload

W/A/S/D + LEFT SHIFT - teleport

The range of teleportation is the green circle around you

Teleportation cooldown: 1.5s | Reload time: 1s

Press H again or ESC to return to the main screen

Figure 24: Instruction Screen

2. Main Game Stage

In the Menu Stage, users will have 3 options of difficulty, the game begins with corresponding level when they pressed one of the 1, 2 or 3 key.

Ammo: 25/25

Skill: READY

Wave: 2 Kills: 6

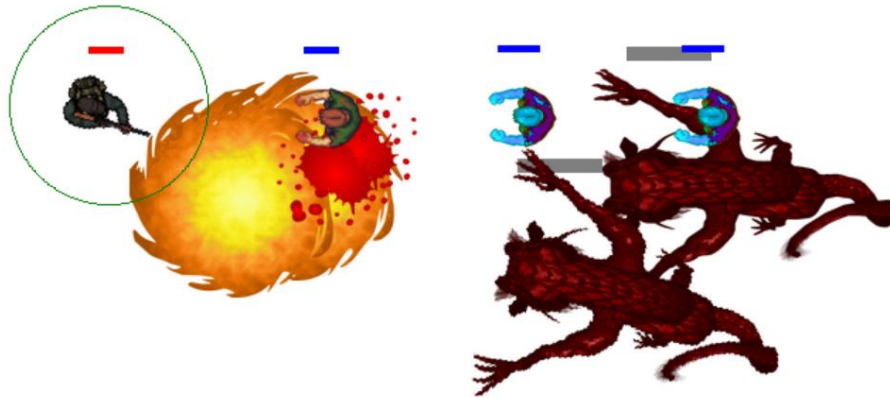


Figure 25: Main Gameplay

3. Game Over & Game Win

Users who survive after a certain number of waves (depending on the level of hardness) will be the Winner! Then, the Congratulation screen will pop up to celebrate the outstanding achievement of users.

CONGRATULATION! YOU WONNNNN !!!

You survived 1 waves

You destroyed 2 blocks

Figure 26: Congratulation screen

On the other hand, if the users could not manage to get through, they will get a Game Over notification.

GAME OVER!

You survived 1 waves

You destroyed 0 blocks

Figure 27: GameOver screen

VI. Report Summary

In conclusion, the development of my top-down shooting game incorporating the SplashKit library in C# has showcased the effective utilization of various Object-Oriented Programming (OOP) concepts and design patterns. Throughout the report, we explored the game's features, the implementation of key OOP core concepts, and the application of three design patterns: the State Pattern, Factory Method Pattern, and Strategy Pattern.