HIGH DISTINCTION CUSTOM PROGRAM REPORT

COS20007

XUAN DAT LE - 103487949



Contents

[**I.** **Abstract** 3](#_Toc136746973)

[**II.** **UML Diagram Overview** 4](#_Toc136746974)

[**III.** **OOP Core Concepts Implementation** 5](#_Toc136746975)

[*1.* *Inheritance* 6](#_Toc136746976)

[*2.* *Polymorphism* 9](#_Toc136746977)

[3. Abstraction & Encapsulation 10](#_Toc136746978)

[**IV.** **Design Pattern** 12](#_Toc136746979)

[1. Factory Method Pattern 13](#_Toc136746980)

[2. State Pattern 16](#_Toc136746981)

[3. Strategy Pattern 17](#_Toc136746982)

[**V.** **Gameplay Walkthrough** 18](#_Toc136746983)

[1. Menu Stage & Instruction Stage 18](#_Toc136746984)

[2. Main Game Stage 20](#_Toc136746985)

[3. Game Over & Game Win 21](#_Toc136746986)

[**VI.** **Report Summary** 22](#_Toc136746987)

# **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

# **UML Diagram Overview**

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#

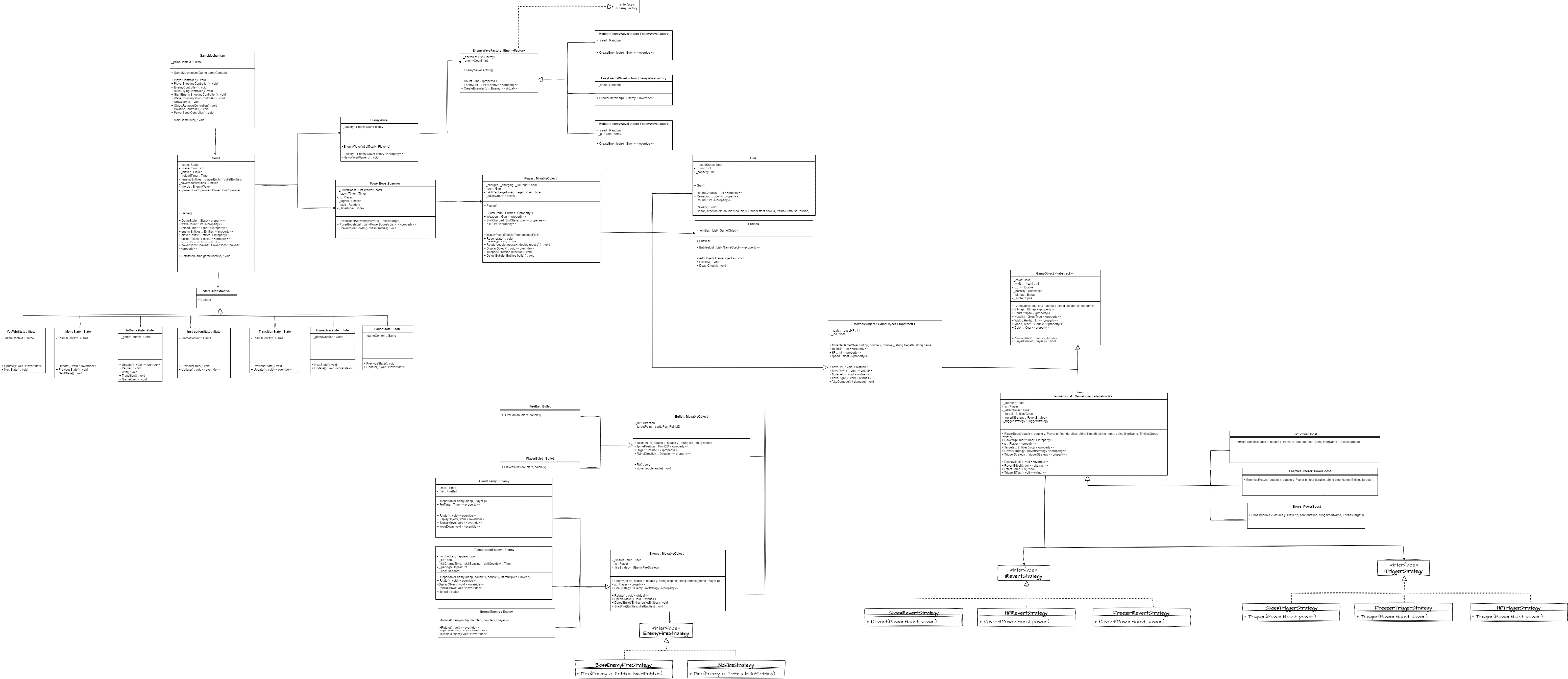


Figure 2: UML Diagram Overview

# **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#.

A picture containing text, line, screenshot, number

Description automatically generated

Figure 3: Inheritance example from the UML diagram

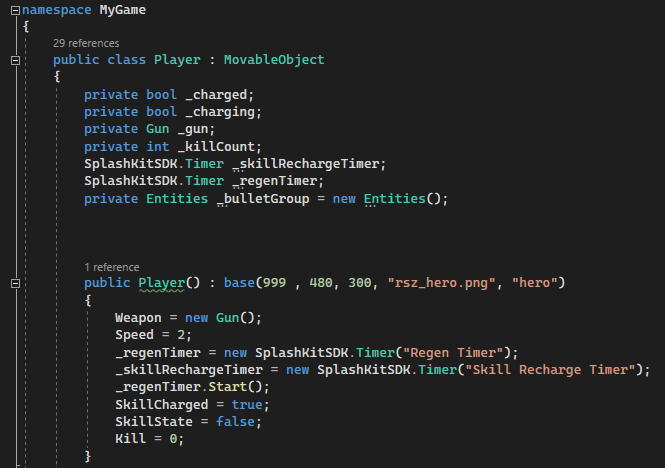
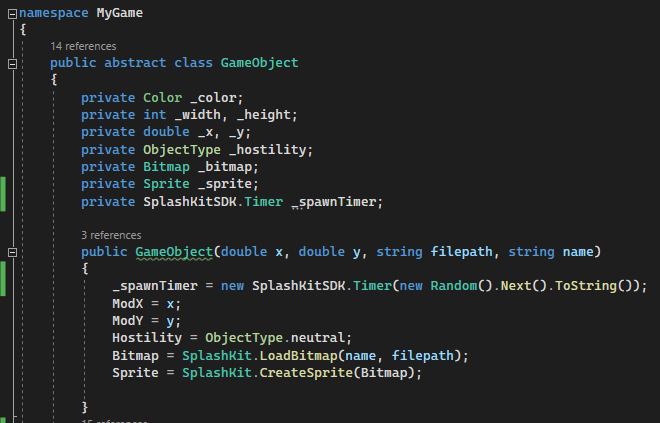
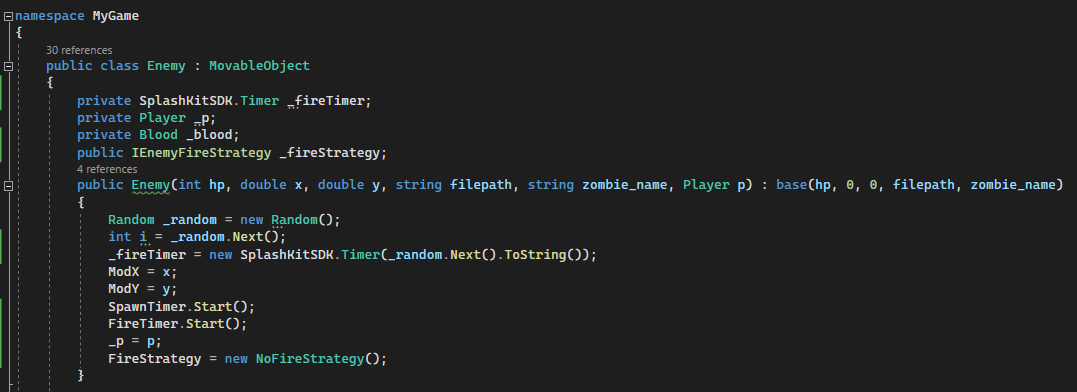


Figure 3&4: Encapsulation and Abstraction code example

## *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:



A picture containing text, screenshot, software

Description automatically generated

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.

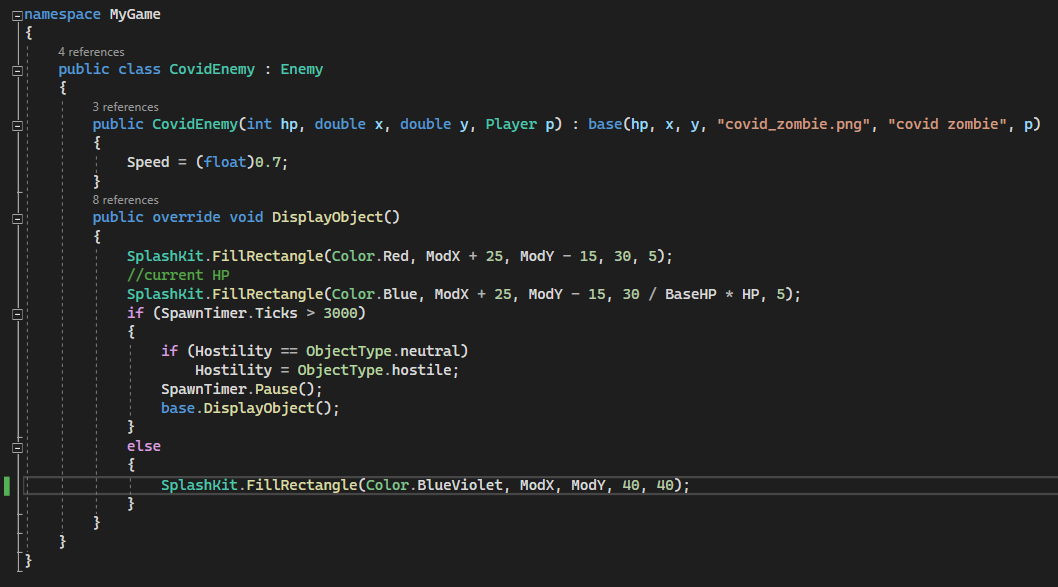


Figure 7: CovidEnemy inherits from its father class.

Figure 8: CovidEnemy inherits from its father class.

A picture containing text, screenshot, software, multimedia software

Description automatically generatedThe “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:

A screen shot of a computer

Description automatically generated with low confidence

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:

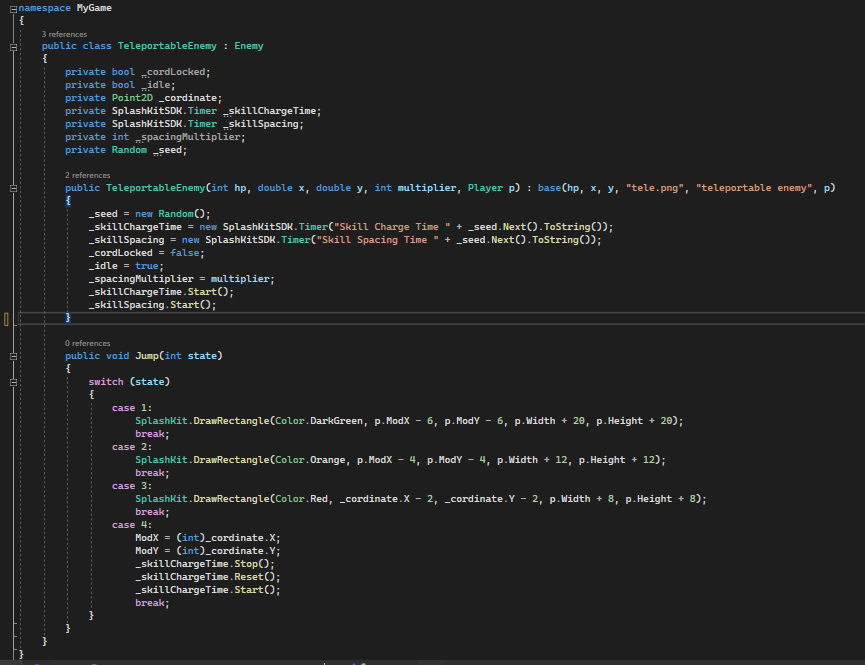


Figure 10: Child class extends method.

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.

## *Polymorphism*

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

A picture containing text, screenshot, software

Description automatically generated

Figure 1: 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.

Figure 12: Implementation Example

A screenshot of a computer

Description automatically generated with low confidence

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.

## Abstraction & Encapsulation

A picture containing text, software, multimedia software, screenshot

Description automatically generated

Figure 13: Abstraction coding example from my program

A screen shot of a computer program

Description automatically generated with medium confidence

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.

A screen shot of a computer program

Description automatically generated with low confidence

Figure 15: Encapsulation demonstration

# **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.

## Factory Method Pattern

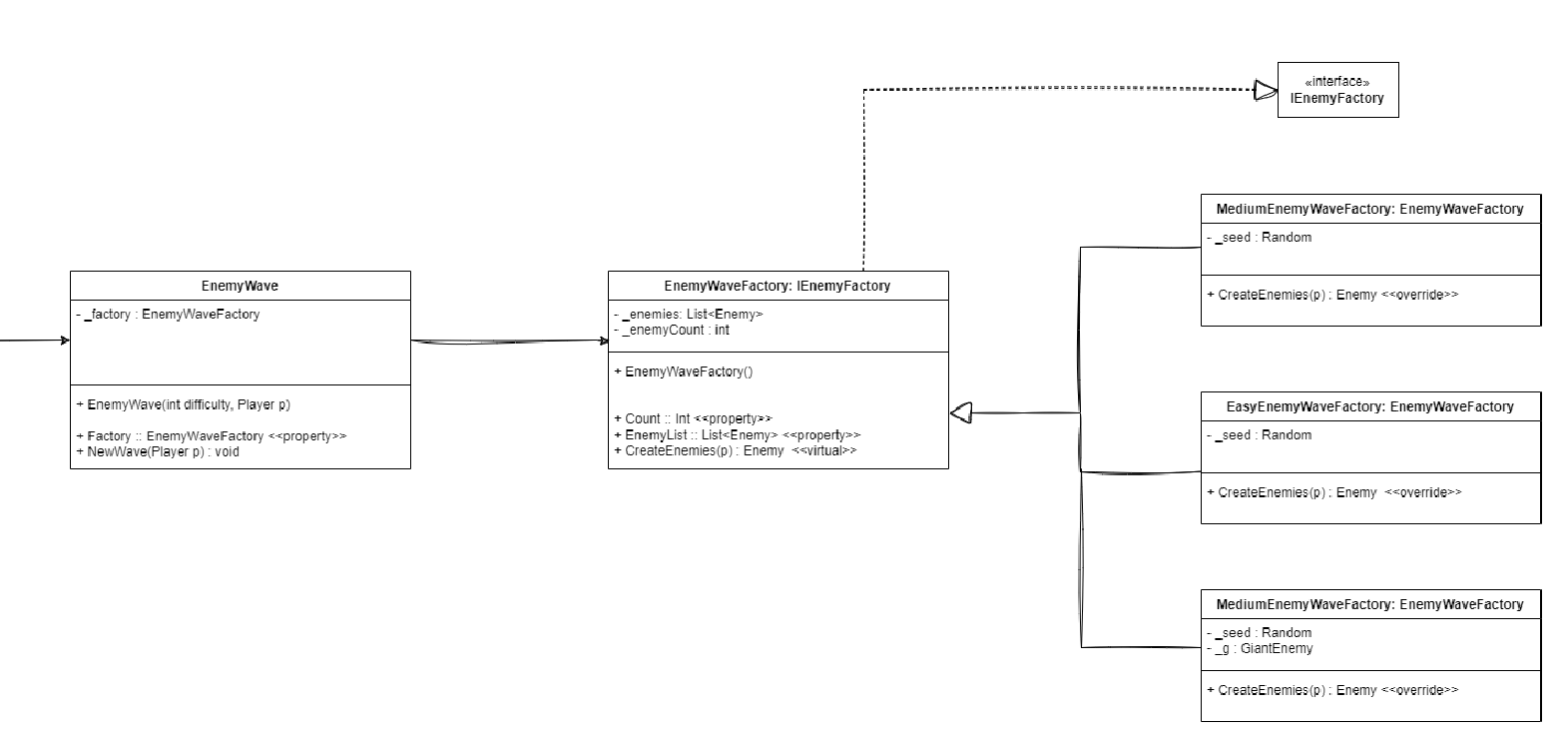


Figure 16: UML Diagram of the Factory Method Design Pattern

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.



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.

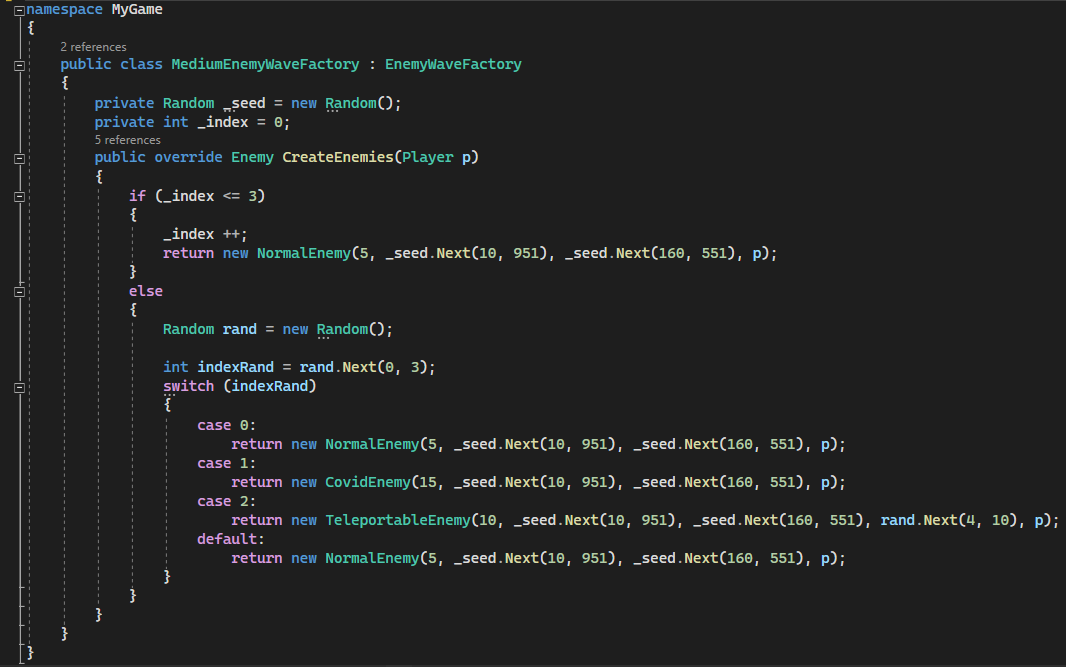


Figure 18: MediumEnemyWaveFactory

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.

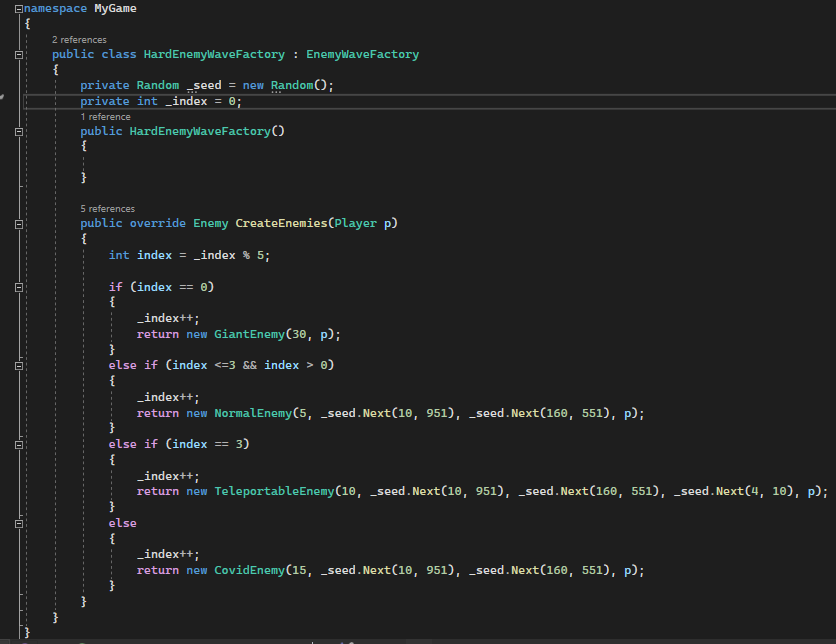


Figure 19: HardEnemyWaveFactory class

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:

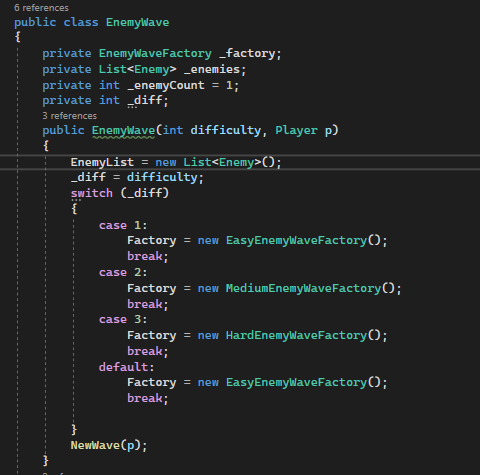


Figure 20: Factory Method implementation

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.

## State Pattern

A picture containing text, rectangle, diagram, sketch

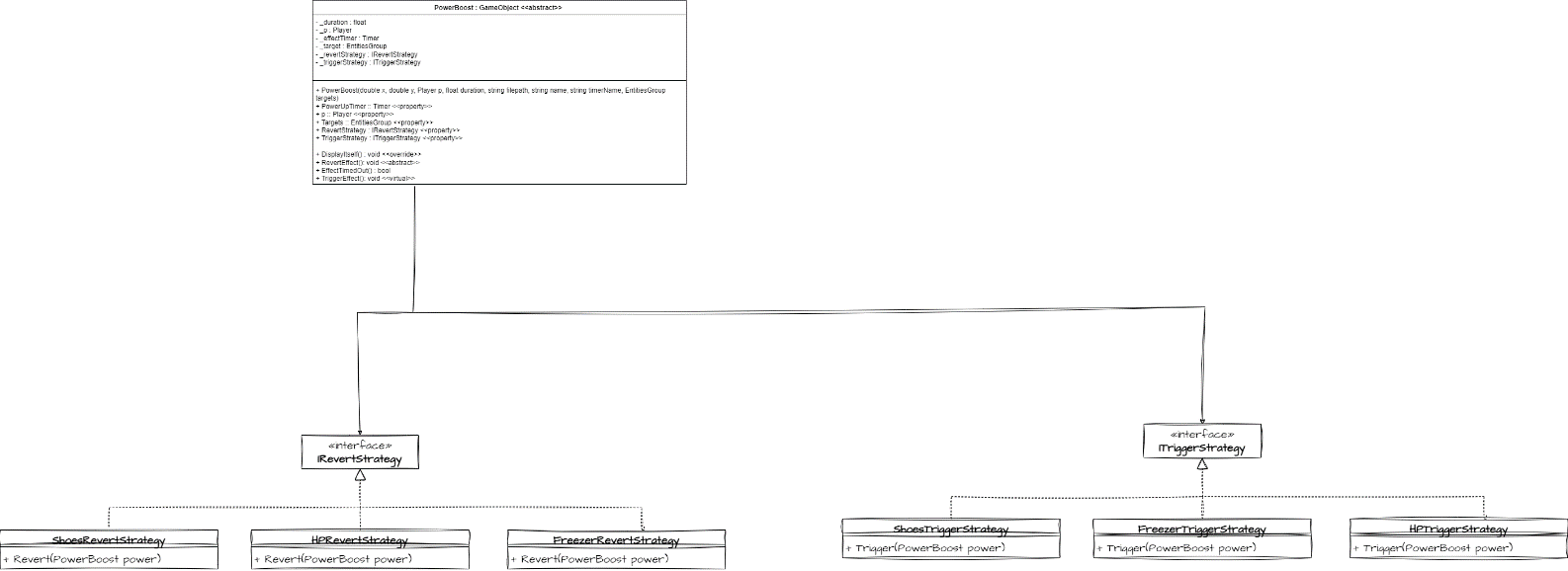
Description automatically generated

Figure 1: State Pattern UML Diagram

Each state-class has a “\_gameContext” field, which consists of the current game state, in that way the game can transit between states without losing the current context.

The State Pattern allowed me to manage the behavior and transitions of the game in a clean and modular way. By representing different states of the game entities as distinct classes, I achieved encapsulation of state-specific behaviors. This pattern facilitated the management of complex state transitions, providing a flexible and maintainable approach to handle the dynamic behavior of game entities.

## Strategy Pattern



The Strategy Pattern proved invaluable in managing different strategies or algorithms for gameplay elements. By encapsulating these strategies within separate classes and allowing them to be interchangeable at runtime, I achieved enhanced flexibility and extensibility. This pattern empowered me to introduce new strategies or modify existing ones without impacting the core game logic, fostering adaptability, and promoting code maintainability.

There are different types of PowerBoost classes, each of them has different strategy of triggering and reverting effect on the Subjective Object, in this case is the Player. This could be done simply by inheritance and overridden methods; however, this could lead to bad coupling and struggle in maintenance and scalability.

A screen shot of a computer program

Description automatically generated with low confidence

Figure 22: Implementation of Strategy Pattern.

Depends on the different types of Triggers or Revert strategies, classes derived from the “PowerBoost” class such as “Shoes”, “EnemiesFreezer” or “HPBoost” could have different effects on the subject. With the strategy pattern applied, we can avoid code duplication as there is no need for modification in the child classes.

# **Gameplay Walkthrough**

## Menu Stage & Instruction Stage

A screenshot of a video game

Description automatically generated

Figure 23: Menu Stage when the game start

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

A screenshot of a video game

Description automatically generated with medium confidence

Figure 24: Instruction Screen

## 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.

A screenshot of a video game

Description automatically generated

Figure 25: Main Gameplay

## 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.

A screenshot of a video game

Description automatically generated with low confidence

Figure 26: Congratulation screen

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

A screenshot of a video game

Description automatically generated with medium confidence

Figure 27: GameOver screen

# **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.