Group T02

Code Refactoring for Carrot Defense

FINAL PRESENTATION

Team members :

2253968 Jia Wenchao

2250397 Qin Cheng

2252042 Zhou Zhengyu

2252215 Peng Jiale

CONTENTS

01 Project Introduction and Motivation for Refactoring

02 Design Patterns in Refactoring

03 Key Issues and Solutions in Refactoring

PART 1

Project Introduction

Overview and Objective

Carrot Defense is a classic tower defense game where players strategically build and upgrade towers to protect the carrot from enemy waves.

Through code refactoring, the game's logic is optimized for better reusability, maintainability, and scalability, enhancing the overall player experience.

Implemented Features

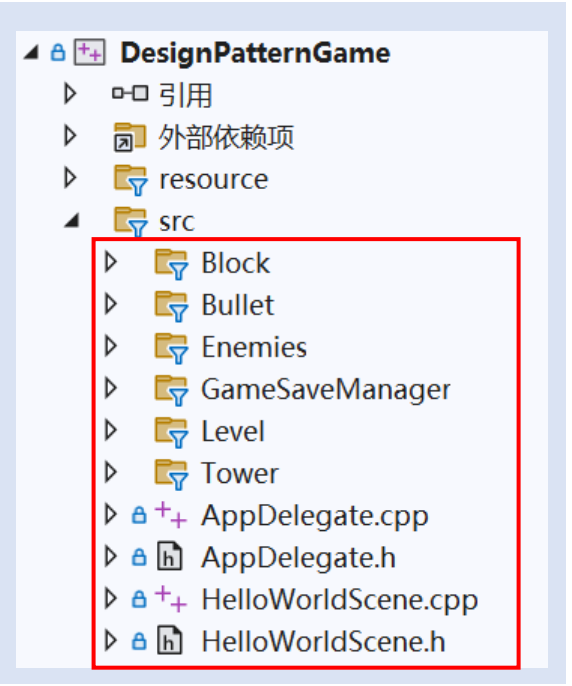
Feature Module	Subfeatures	Feature Module	Subfeatures
Basic Functions		**Advanced Features**	Enemy Creation
Diverse Tower Functions	Build Tower, Delete Tower, Two-level Tower Upgrades	Special Attack Mode	Turret Special Abilities, Activate Special Abilities
Special Effects Display	Tower Attack Effects, Monster Hit Effects	Upgrade Effect Enhancement	Unlock New Effects on Upgrade
Economy System	Earn Gold by Defeating Monsters, Spend Gold to Build and Upgrade Turrets	Flexible Game Flow	Restart During Gameplay, Exit and Select Level
Monsters and Maps	Three Types of Monsters, Two Maps		
Health and Background Music	Display Carrot Health, Background Music		
Save Function	Save Game Progress		

PART 1

Motivation for Refactoring

- Code Structure Chaos
- Poor Scalability
- Performance Bottlenecks
- … …

- Improve Code Maintainability and Readability
- Optimize Performance
- Enhance Flexibility and Scalability
- Improve User Experience
- … …



PART 2

Design Patterns in Refactoring

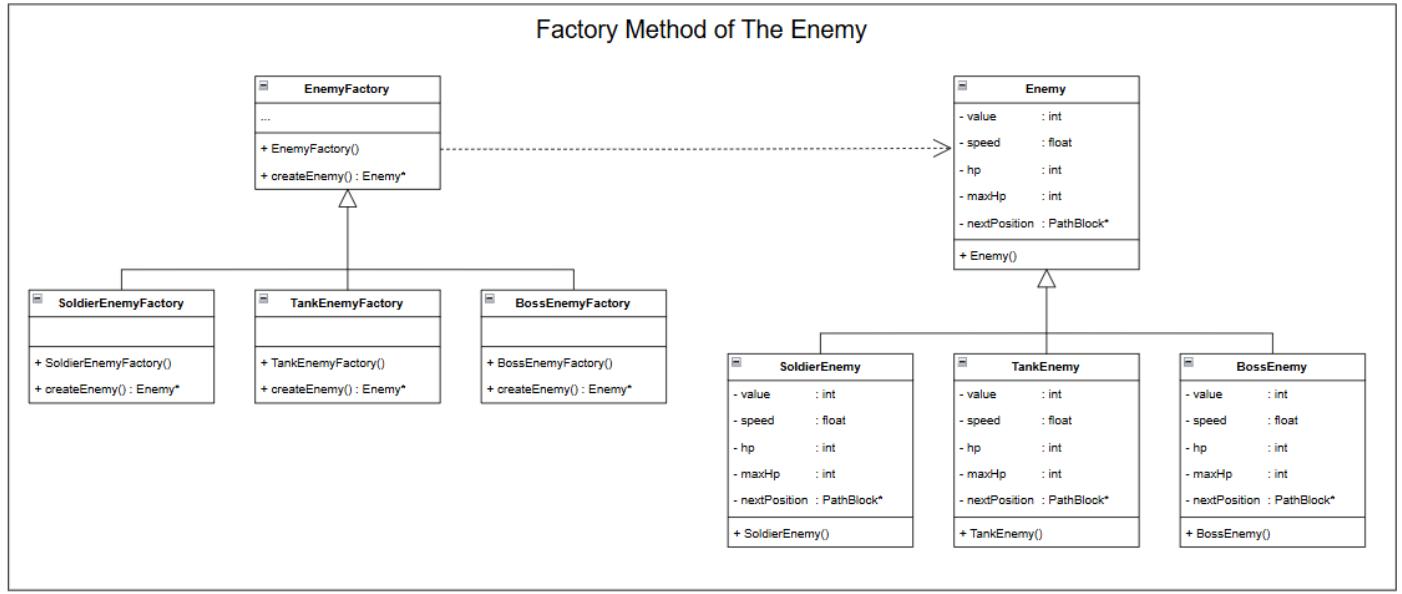
Design Patterns Overview

Number	Name	Type	Brief Description
1	Factory Method	Creational Patterns	Enemy Creation
2	Singleton	Creational Patterns	Singleton of EnemyNotifyManager
3	Flyweight	Structural Patterns	Bullet Texture Sharing
4	Decorator	Structural Patterns	Bullet attack mode expansion
5	Observer	Behavioral Patterns	Enemies' subscription and tower's update
6	Template Method	Behavioral Patterns	Level Initialization Process
7	Object Pool	Others	Bullet Object Pool

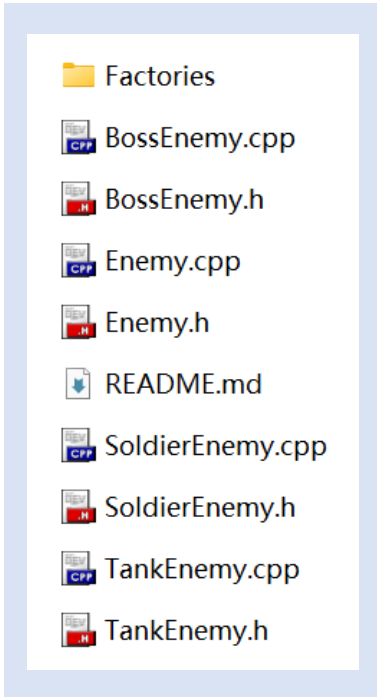
PART 2

Design Patterns in Refactoring——Factory Pattern

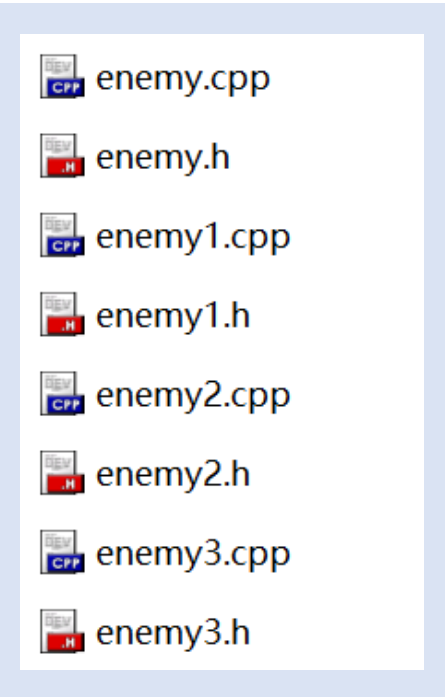
Class diagram of Factory Pattern



Restructure File Structure



Original File Structure



Reasons :

- Clear code structure
- Enhanced scalability
- Ease of management and maintenance
- Reduced code duplication

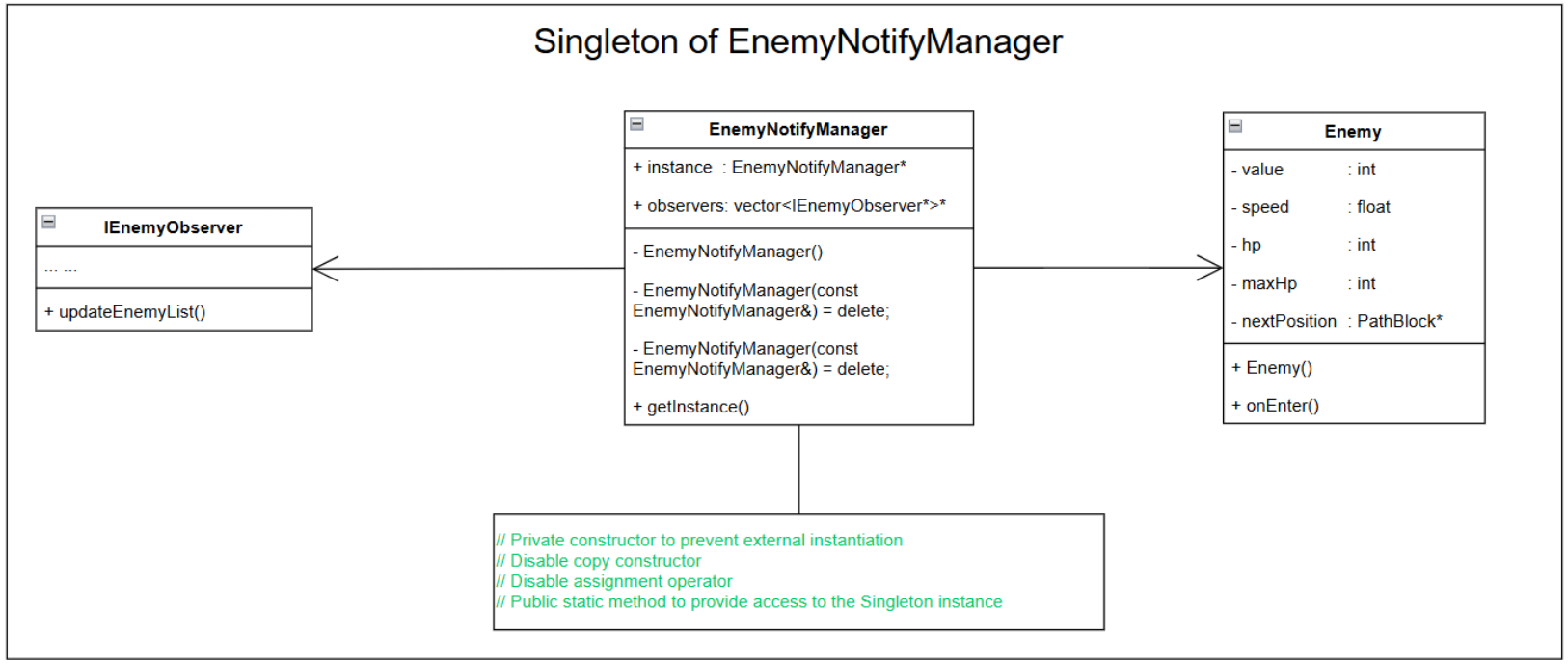
Benefits :

- Decouples creation and usage
- Improves code reusability
- Adheres to design principles
- Easily extends new features
- Enhances code readability

PART 2

Design Patterns in Refactoring——Singleton

Class diagram of Singleton



Benefits :

- Avoiding Duplicate Notifications

  The Singleton pattern ensures a single instance of EnemyNotifyManager, preventing monsters from notifying the Tower multiple times, thus improving efficiency.

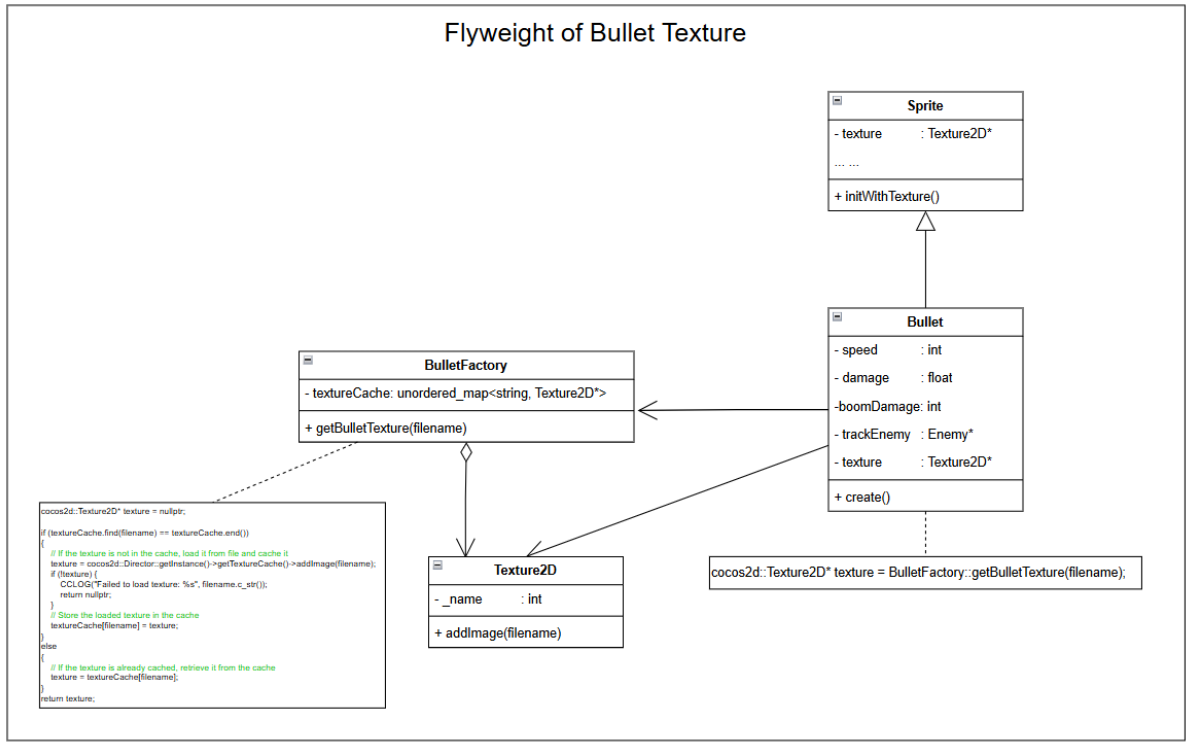
- Centralized Notification Management

  With the Singleton pattern, all notifications are managed by one instance, simplifying code maintenance and expansion.

PART 2

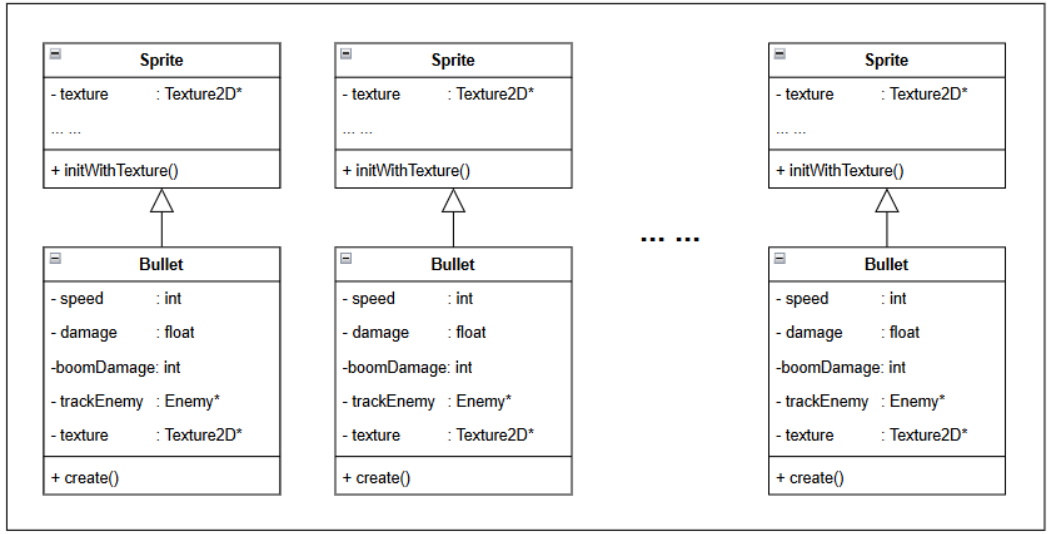
Design Patterns in Refactoring——Flyweight

Class diagram of Flyweight



Original Structure

Each time a bullet is created, a texture image that takes up a large amount of space is saved.



Reasons :

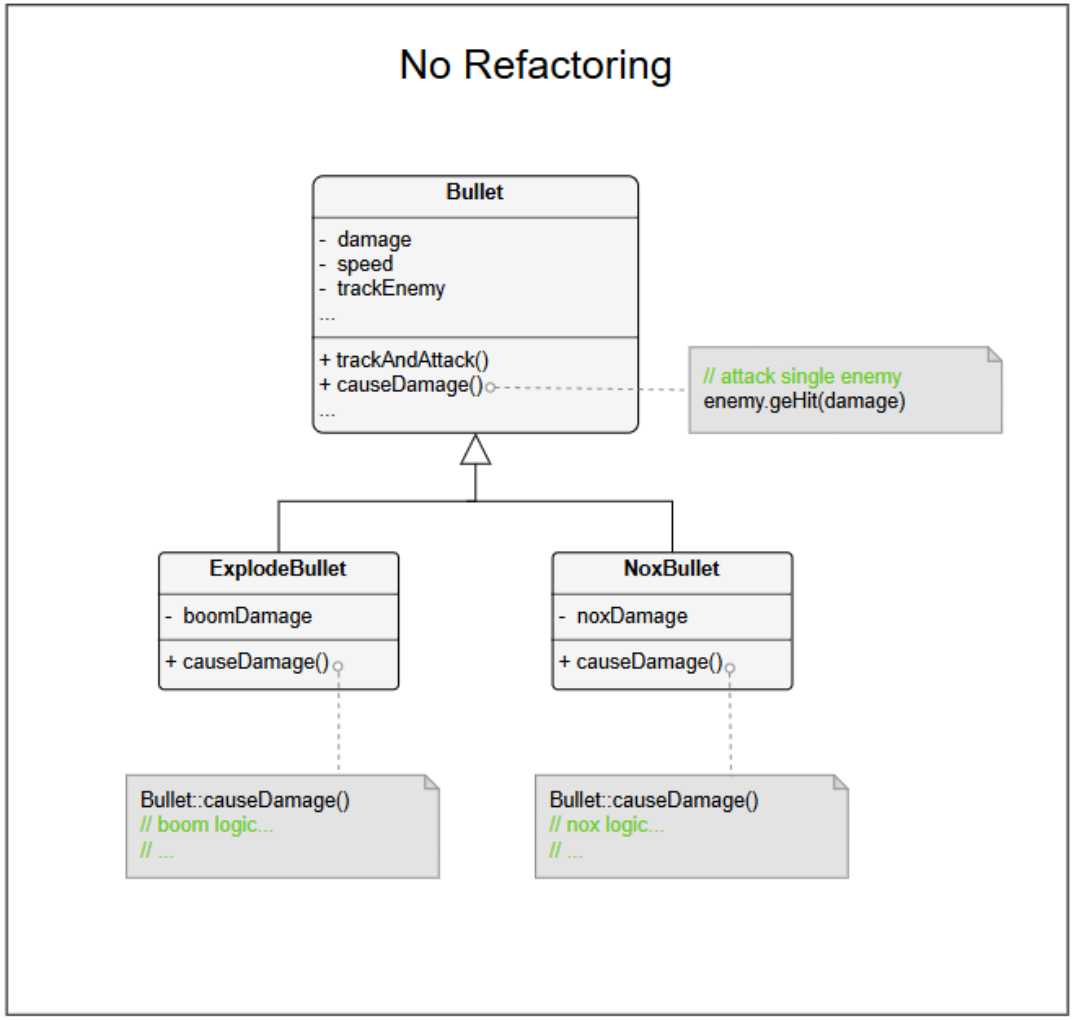
- High memory usage
- Low texture loading efficiency
- Disorganized resource management

Benefits :

- Reduces memory consumption
- Improves loading efficiency
- Supports high-concurrency scenarios
- Facilitates resource management

PART 2 Design Patterns in Refactoring——Decorator

Original Structure



Issues or Reason for Refactoring

Rigid Structure:

If new attack types or combinations (e.g., an explosive bullet with poison damage) need to be implemented, the system requires the creation of new subclasses for each combination. This results in a proliferation of subclasses, making the system harder to manage and extend.

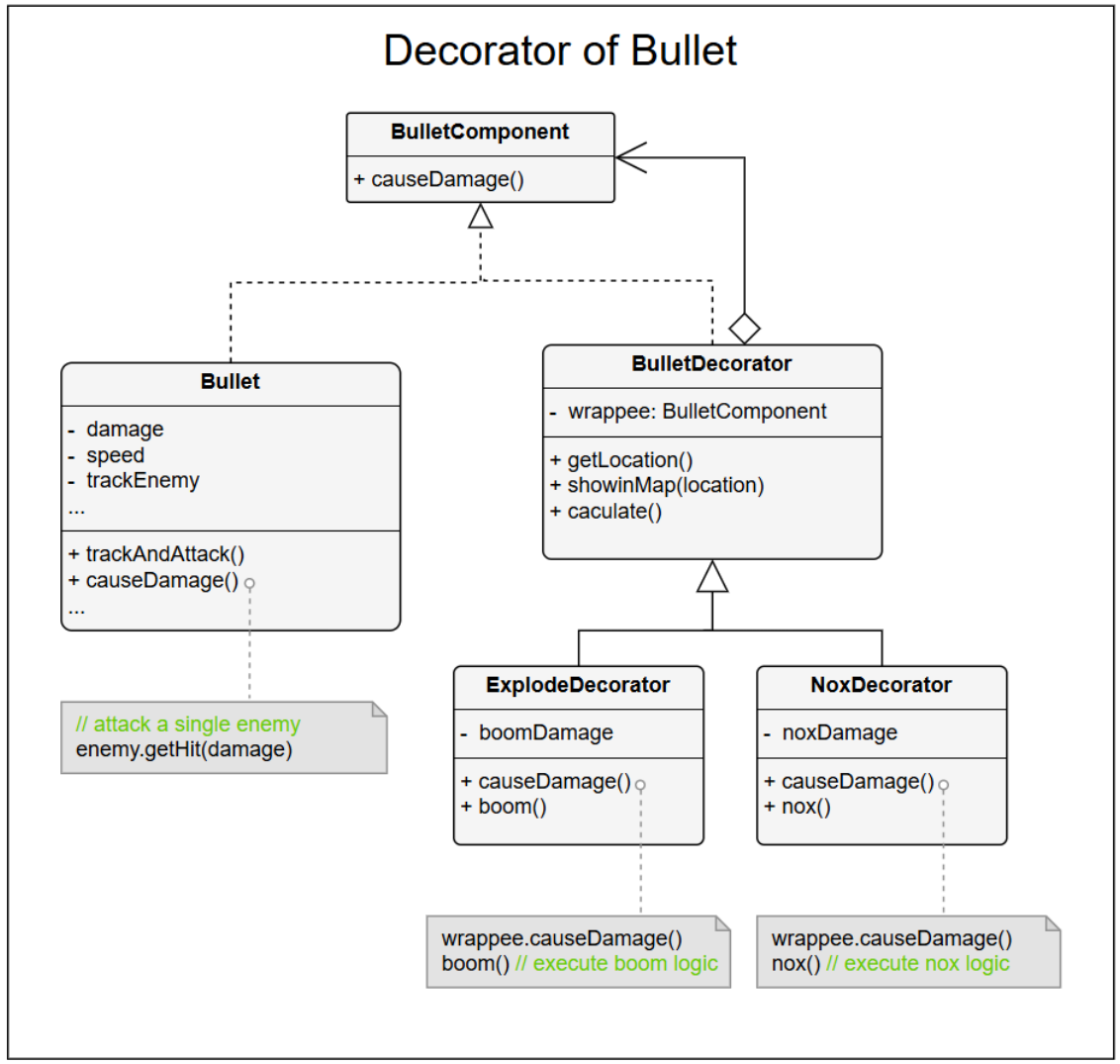
Limited Flexibility:

Inheritance tightly couples behaviors to specific subclasses, making it difficult to dynamically change or combine behaviors during runtime.

PART 2

Design Patterns in Refactoring——Decorator

Class diagram of Decorator



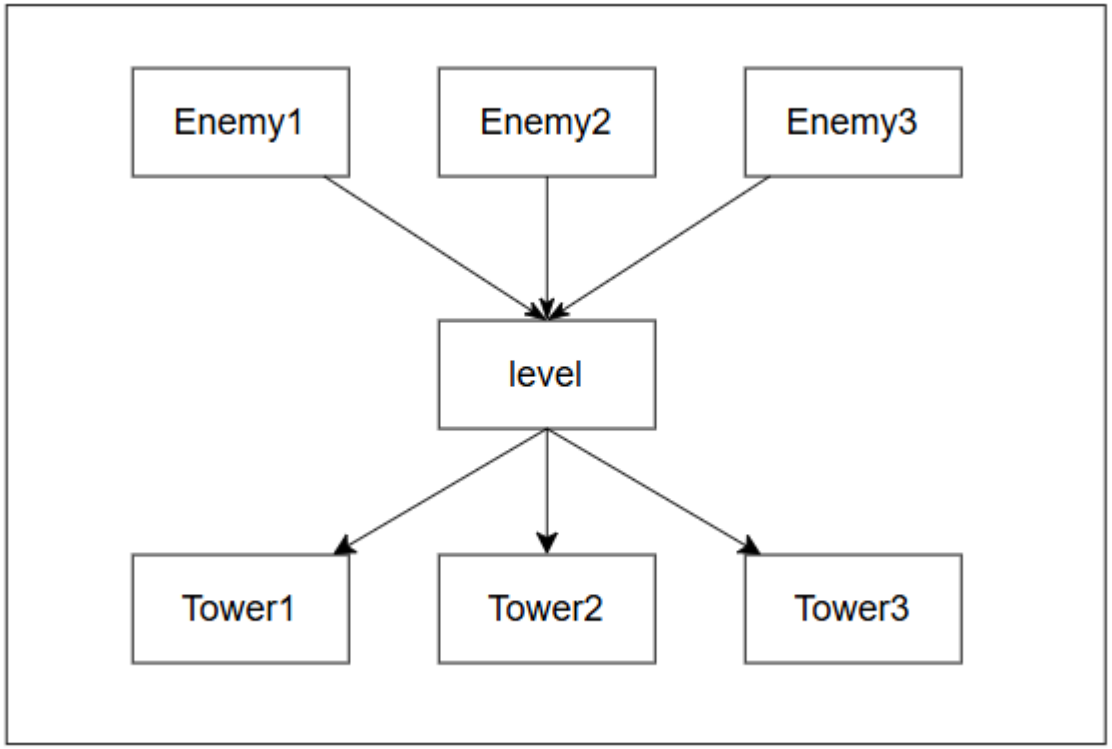
Benefits :

- **Flexibility:** Dynamically add or combine features (e.g., explosion, toxic damage) without changing existing code.
- **Extensibility:** Add new functionalities via decorators without modifying the Bullet class.
- **Open/Closed Principle:** Extend behavior without altering existing code.
- **Code Reuse:** Reusable decorators reduce duplication and enhance maintainability.

PART 2

Design Patterns in Refactoring——Observer

Original Structure



Original Logic for Tower to Obtain Enemies

- The enemies of each wave are stored in the level.
- The tower obtains the enemies of each wave by iteration, regardless of whether the enemy has appeared or is already dead.

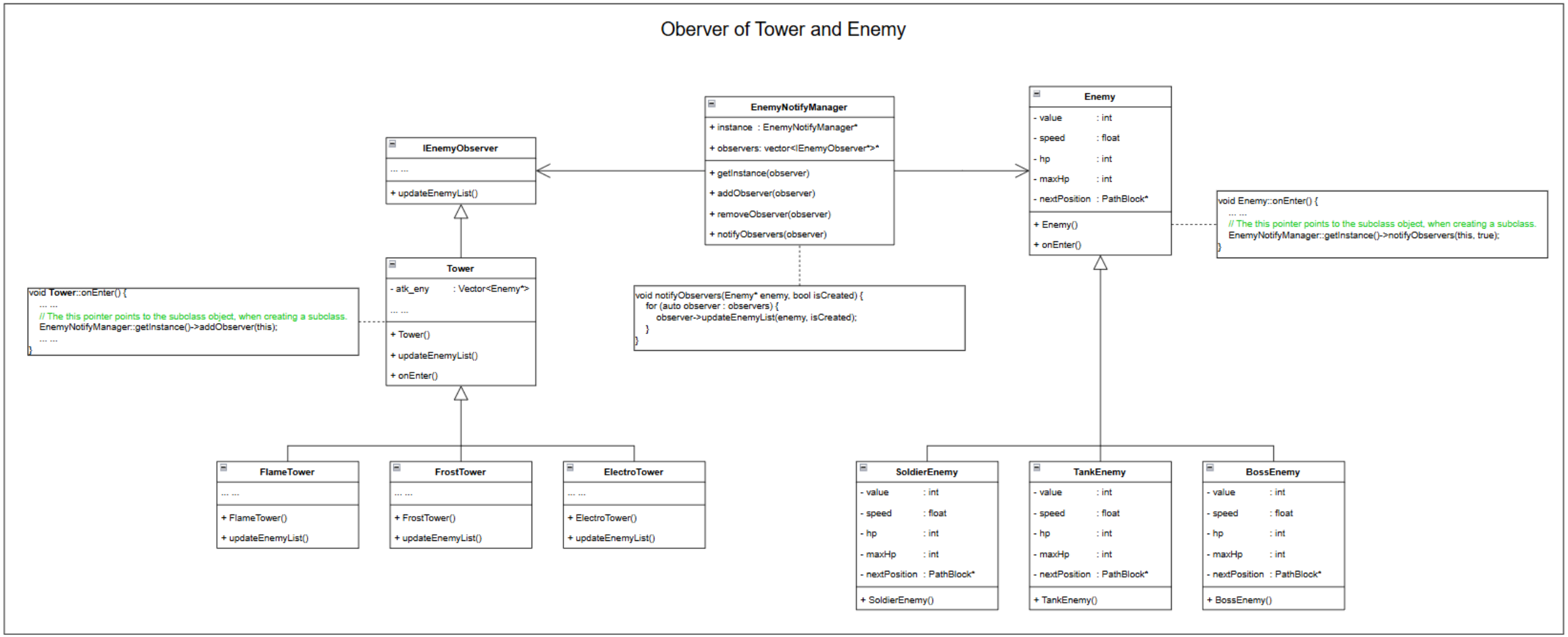
Disadvantages:

- Resource Waste
- High Coupling
- Poor Scalability

PART 2

Design Patterns in Refactoring——Observer

Class diagram of Observer

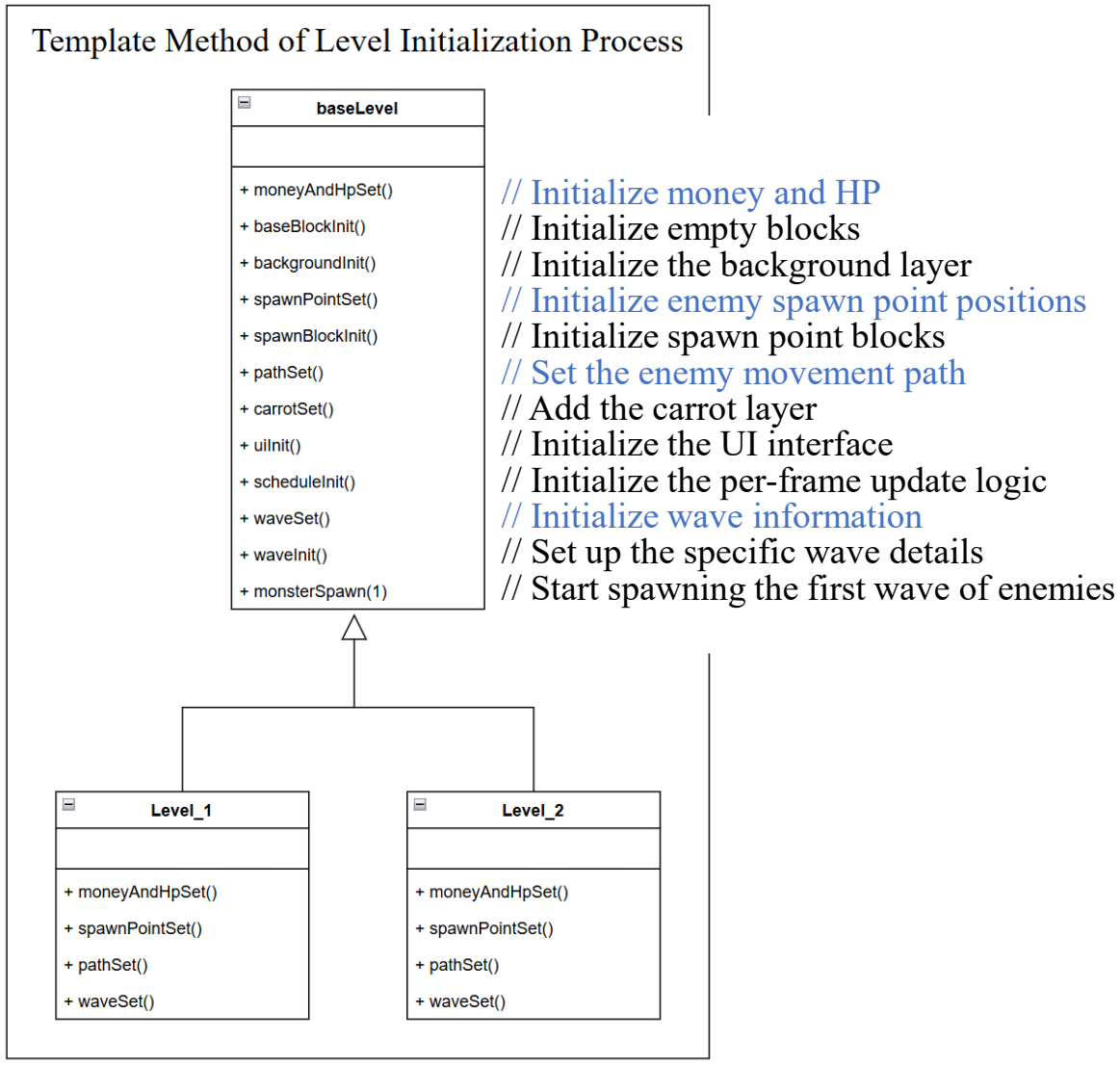


Benefits : Improved Efficiency; Reduced Coupling; Cleaner Logic;

PART 2

Design Patterns in Refactoring ——Template Method

Class diagram of Template Method



Reasons :

Lack of Unified Interface. The level classes don't have a consistent initialization interface, increasing differences and the risk of errors.

Unclear Control Flow. Each level handles its own initialization, making the code messy and error-prone.

Difficult to Modify Initialization. Changing the initialization process requires modifying each level individually, which is time-consuming and hard to maintain.

Benefits :

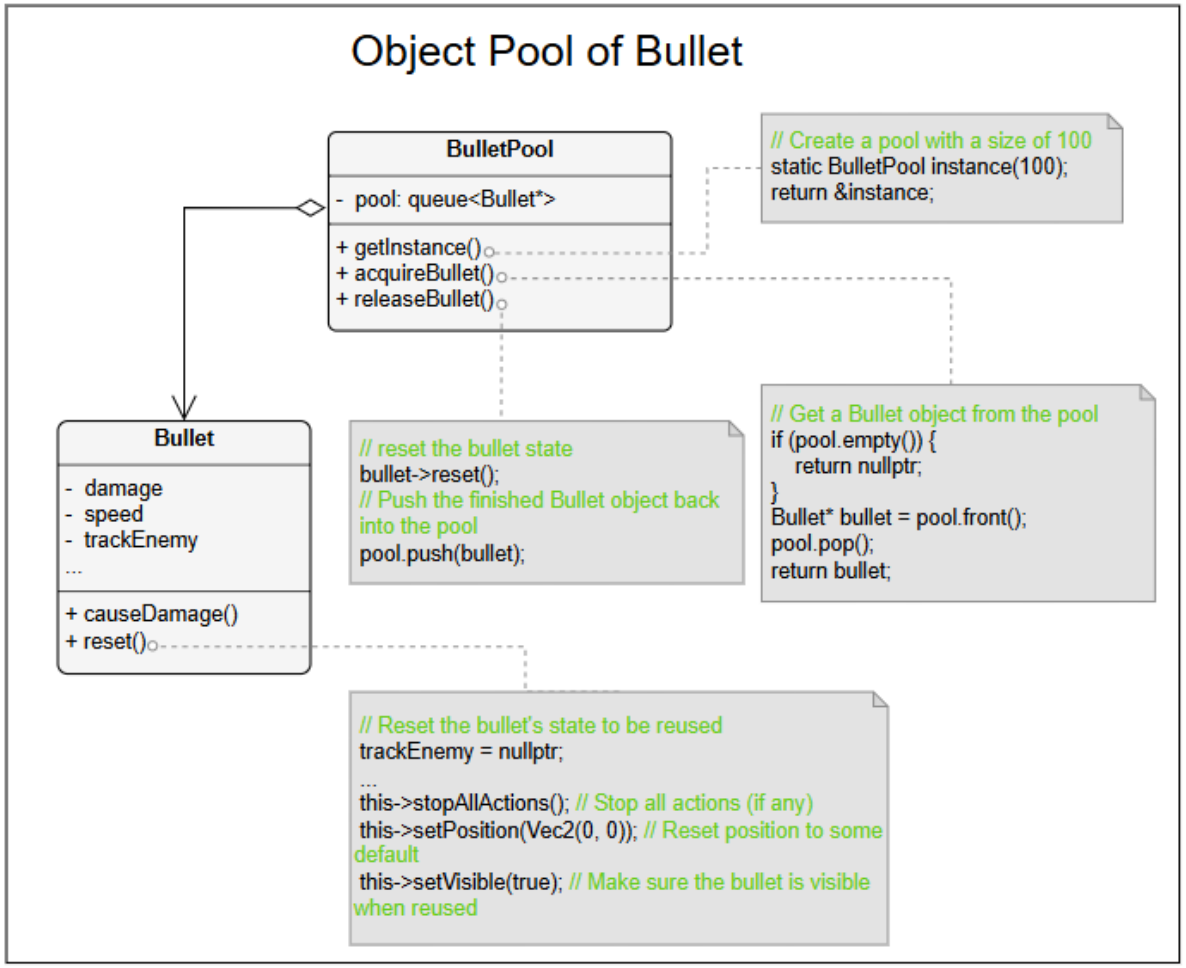
Higher Reusability. Common logic is reused directly in subclasses, with only unique parts needing overrides.

Easier Feature Expansion. Adding steps like soundInit() only requires updates in the parent class.

Consistency. Ensures all levels follow the same initialization flow, avoiding discrepancies.

PART 2 Design Patterns in Refactoring——Object Pool

Class diagram of Object Pool



The Way to Create Bullets

Before

```
// Create a new Bullet
Bullet* bullet = Bullet::create("nox.png");
...
// Bullet destruction
bullet->removeFromParent()
```

Benefits :

- Performance Boost
- Resource Efficiency
- Memory Leak Prevention
- Enhanced Game Efficiency

After

```
// Create a new Bullet
Bullet* bullet = BulletPool::getInstance()->acquireBullet();
...
// Bullet destruction
BulletPool::getInstance()->releaseBullet(bullet);
```

PART 3

Key Issues and Solutions in Refactoring

1 The Flyweight of Bullet

Issue: In the Flyweight pattern, since bullet needs to inherit Cocos' sprite class and properties like position are stored in the base class, the entire sprite cannot be shared directly.

Solution: Only the shared part of the bullet, i.e., the texture, can be shared. A texture pool is created to manage the shared textures for all bullets. Different bullet objects will use the same texture when created.

2 Tower cannot attack

Issue: When a new Tower is created, it cannot attack enemies that have already entered the field.

Solution: When a new Tower is created, it should be able to detect the enemies already present on the field. Therefore, when the Tower is created, the NotifyManager will notify the tower of enemies' presence, allowing it to become aware of them and immediately start tracking and attacking.

The presentation is over, Thank you for watching!