****

**Software Design Patterns**

**Code Refactoring for Carrot Defense**

Group T02

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Number** | **Phone Number** | **Email Address** |
| **Cheng Qin** | **2250397** | **19552120519** | **2250397@tongji.edu.cn** |
| **Jiale Peng** | **2252215** | **13763281725** | **2252215@tongji.edu.cn** |
| **Wenchao Jia** | **2253968** | **18616581802** | **2253968@tongji.edu.cn** |
| **Zhengyu Zhou** | **2252042** | **13165347188** | **2252042@tongji.edu.cn** |

January 5, 2025

目录

[**1. Project Introduction** 1](#_Toc186919149)

[**1.1 Project Background** 1](#_Toc186919150)

[**1.2 Project Features** 2](#_Toc186919151)

[**2. Project Refactoring Overview** 3](#_Toc186919152)

[**2.1 Project Structure** 3](#_Toc186919153)

[**2.2 Design Patterns Summary Table** 4](#_Toc186919154)

[**3. Design Patterns Details** 5](#_Toc186919155)

[**3.1 Factory Method** 5](#_Toc186919156)

[**3.1.1 Pattern Introduction** 5](#_Toc186919157)

[**3.1.2 Application Scenarios** 5](#_Toc186919158)

[**3.1.3 Class Diagram** 5](#_Toc186919159)

[**3.1.4 Key Code Demonstration** 6](#_Toc186919160)

[**3.2 Singleton** 6](#_Toc186919161)

[**3.2.1 Pattern Introduction** 6](#_Toc186919162)

[**3.2.2 Application Scenarios** 7](#_Toc186919163)

[**3.2.3 Class Diagram** 7](#_Toc186919164)

[**3.2.4 Key Code Demonstration** 7](#_Toc186919165)

[**3.3 Flyweight** 8](#_Toc186919166)

[**3.3.1 Pattern Introduction** 8](#_Toc186919167)

[**3.3.2 Application Scenarios** 8](#_Toc186919168)

[**3.3.3 Class Diagram** 9](#_Toc186919169)

[**3.3.4 Key Code Demonstration** 9](#_Toc186919170)

[**3.4 Decorator** 10](#_Toc186919171)

[**3.4.1 Pattern Introduction** 10](#_Toc186919172)

[**3.4.2 Application Scenarios** 10](#_Toc186919173)

[**3.4.3 Class Diagram** 11](#_Toc186919174)

[**3.4.4 Key Code Demonstration** 11](#_Toc186919175)

[**3.5 Observer** 12](#_Toc186919176)

[**3.5.1 Pattern Introduction** 12](#_Toc186919177)

[**3.5.2 Application Scenarios** 12](#_Toc186919178)

[**3.5.3 Class Diagram** 12](#_Toc186919179)

[**3.5.4 Key Code Demonstration** 13](#_Toc186919180)

[**3.6 Template Method** 14](#_Toc186919181)

[**3.6.1 Pattern Introduction** 14](#_Toc186919182)

[**3.6.2 Application Scenarios** 14](#_Toc186919183)

[**3.6.3 Class Diagram** 14](#_Toc186919184)

[**3.6.4 Key Code Demonstration** 15](#_Toc186919185)

[**3.7 Object Pool** 15](#_Toc186919186)

[**3.7.1 Pattern Introduction** 15](#_Toc186919187)

[**3.7.2 Application Scenarios** 16](#_Toc186919188)

[**3.7.3 Class Diagram** 16](#_Toc186919189)

[**3.7.4 Key Code Demonstration** 16](#_Toc186919190)

**1. Project Introduction**

**1.1 Project Background**

**Carrot Defense** is a classic tower defense game where players build different types of towers on a map to stop the invasion of enemies and protect the carrot from being destroyed. In the game, players must strategically place towers to counter increasingly difficult waves of enemies. By defeating enemies, players earn coins which they can use to build or upgrade towers, gradually enhancing their defense capabilities.

However, as the project progressed and more features were added, several issues emerged in the initial version of the project , which made the project difficult to maintain and expand. The main problems include:

1. **Code Structure Chaos:** The initial implementation was relatively simple, but as more features were added, the code became cluttered and hard to understand. Various modules were tightly coupled, which led to redundant code and made debugging and modification difficult.

2. **Poor Scalability**: While the existing code handles basic functionality, it lacks the flexibility to easily extend with new features, such as adding new turret types, monsters, or levels. Expanding functionality often requires significant changes to existing code, which increases development and maintenance costs.

3. **Performance Bottlenecks:** As the game scales with more monsters and complex maps, the existing code fails to optimize critical parts, such as collision detection and attack logic. This can cause the game to slow down or experience lag, affecting the user experience.

To address these issues, we decided to refactor the existing code. The goal of the refactor is to improve the maintainability, scalability, performance, and overall user experience of the game. The specific objectives include:

1. **Improve Code Maintainability and Readability**: The existing code has redundant and non-standard parts, making it difficult to understand and modify. By refactoring, we will modularize and decouple the code to reduce redundancy and make each module’s functionality clearer and more independent. This will enhance code readability and ease of maintenance, allowing for easier debugging and modification.

2. **Optimize Performance**: As the game grows, performance bottlenecks, especially in collision detection and attack logic, become more noticeable. We will optimize these critical areas by reducing unnecessary calculations, improving data structures, and using techniques like spatial partitioning to boost the game’s efficiency and ensure smooth gameplay, even with larger numbers of monsters and more complex maps.

3. **Enhance Flexibility and Scalability**: To accommodate future feature expansions—such as adding new turret types, monsters, or levels—we will redesign the code to make it more flexible and modular. This will allow new features to be added easily without requiring large-scale changes to the existing code. By using interfaces and abstract classes, we will ensure that new functionality can be integrated smoothly with the existing system, reducing the cost of future development and maintenance.

**4. Improve User Experience**: The refactor will also focus on enhancing the visual effects, interactions, and overall game experience. We will add more advanced turret attack effects, monster hit animations, and improve the user interface to make the game more immersive and enjoyable. Additionally, we will ensure that the game is responsive and intuitive for players, leading to a better overall experience.

In conclusion, the refactor of Carrot Defense aims to address the key issues encountered in the initial implementation, ensuring that the game is not only easier to maintain and expand but also laying a solid foundation for future feature extensions. This will allow the game to efficiently handle new gameplay elements and features while continuously enhancing the player experience.

**1.2 Project Features**

After the refactor, the Carrot Defense game system has seen significant improvements in structure, performance, scalability, and user experience. The core goal of the refactor was to optimize the implementation of existing features and provide more flexible support for potential future expansions. By modularizing and decoupling the code, we have added more depth and strategic elements to the game, while also enhancing its smoothness and maintainability. Below are the **main features** of the system after the refactor:

**Table 1 Implemented Features**

|  |  |  |  |
| --- | --- | --- | --- |
| Feature Module | Subfeatures | Feature Module | Subfeatures |
| Basic Functions |  | **Advanced Features** |  |
| 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 |  |  |

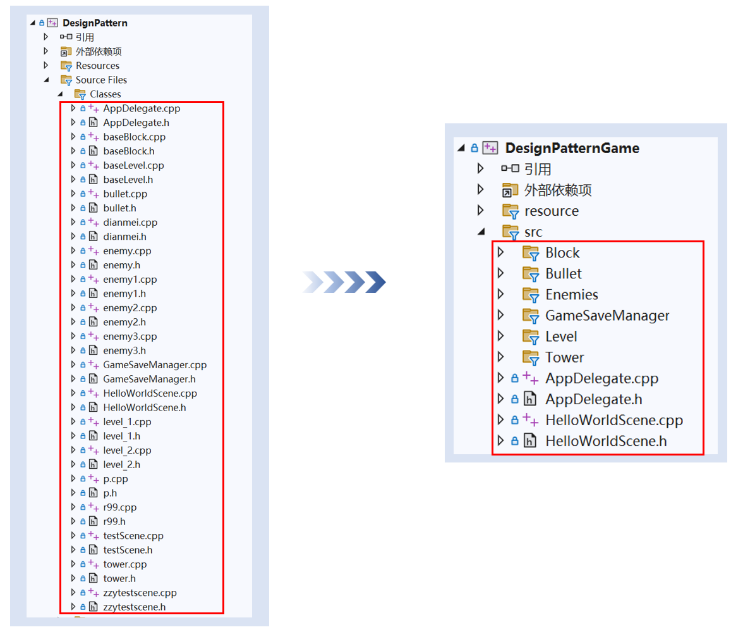
**2. Project Refactoring Overview**

In this section, we will compare the overall structure of the project before and after the refactor, and provide an overview of the design patterns used. Through this comparative analysis, we will clearly demonstrate the architectural optimizations brought by the refactor and how the selected design patterns help improve the maintainability, scalability, and performance of the code. Next, we will present the project structure before and after the refactor, and briefly introduce the design patterns chosen.

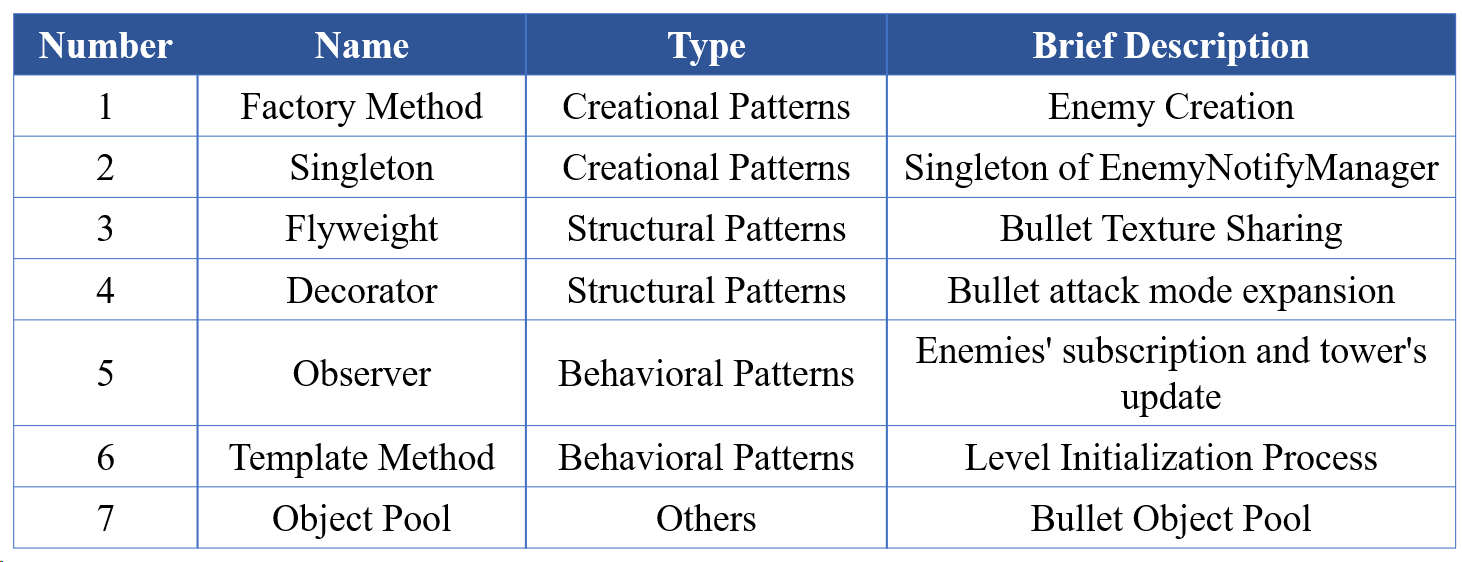
**2.1 Project Structure**

In this section, we will compare the changes in the project's structure before and after the refactor. The diagram below provides a clear visualization of the code organization before and after the refactor.

Before the refactor (left), the code structure was relatively chaotic, with files for various functional modules scattered and lacking effective categorization. This made code maintenance and feature expansion more challenging.

After the refactor (right), we implemented a modular design for the project structure, grouping related functional files into their respective directories, such as Block, Bullet, Enemies, and others. This clear separation of responsibilities among different functional modules makes the code easier to maintain and extend while also improving its overall readability.

**2.2 Design Patterns Summary Table**

During the process of code refactoring, we introduced various classic design patterns to enhance the system's maintainability and scalability. The application of these patterns not only optimized the code structure but also significantly improved code reusability and readability. Below is an overview and analysis of the design patterns utilized:

**3. Design Patterns Details**

**3.1 Factory Method**

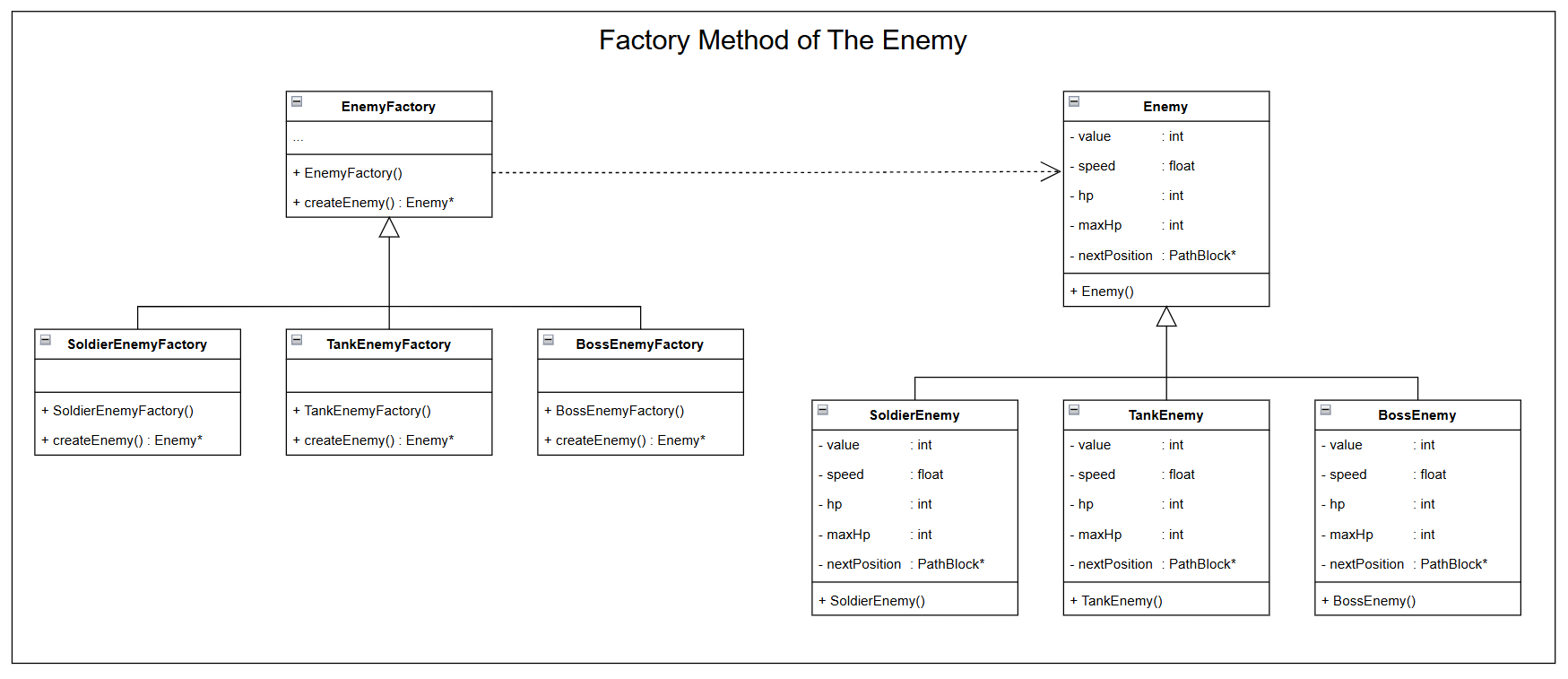
**3.1.1 Pattern Introduction**

The Factory Method is a creational design pattern that defines an interface for creating objects but allows subclasses to alter the type of objects that will be created. Instead of directly instantiating objects using the new keyword, this pattern provides a flexible approach to encapsulate the creation logic within a factory method, thereby promoting loose coupling and scalability.

In the context of game development, particularly for creating Enemy objects, the Factory Method ensures that the creation logic for enemies is centralized and consistent, which facilitates future modifications and enhances code reusability.

**3.1.2 Application Scenarios**

Before implementing the Factory Method, the creation of the Enemy objects was scattered across multiple parts of the code and was done through direct instantiation. This approach resulted in duplicated logic, poor scalability, and made it difficult to manage changes to the Enemy creation process. After the refactor, the creation of all Enemy objects was centralized in a factory class, which not only facilitates future modifications but also improves the reusability of the code.

**3.1.3 Class Diagram**

This class diagram illustrates a design based on the Factory Method Pattern. The core idea is that the abstract factory class EnemyFactory defines an interface for creating enemies, which is implemented by its subclasses SoldierEnemyFactory, TankEnemyFactory, and BossEnemyFactory to create corresponding enemy objects SoldierEnemy, TankEnemy, and `BossEnemy. These concrete enemy classes inherit from the abstract base class `Enemy`. This design separates the object creation logic from its usage, making it easy to extend new enemy types and demonstrating the flexibility and scalability of the Factory Method Pattern.

**3.1.4 Key Code Demonstration**

#include "BossEnemyFactory.h"

Enemy\* BossEnemyFactory::createEnemy() {

return BossEnemy::create();

}

#include "SoldierEnemyFactory.h"

Enemy\* SoldierEnemyFactory::createEnemy() {

return SoldierEnemy::create();

}

#include "TankEnemyFactory.h"

Enemy\* TankEnemyFactory::createEnemy() {

return TankEnemy::create();

}

**3.2 Singleton**

**3.2.1 Pattern Introduction**

The Singleton Pattern is a design pattern that restricts the instantiation of a class to a single object and provides a global point of access to that object. It ensures that only one instance of the class exists throughout the application's lifecycle, which is particularly useful for managing shared resources or controlling access to a specific component, such as logging, database connections, or in this case, managing the state of game enemies.

The Singleton Pattern generally involves:

1. **Private constructor:** To prevent external instantiation of the class.

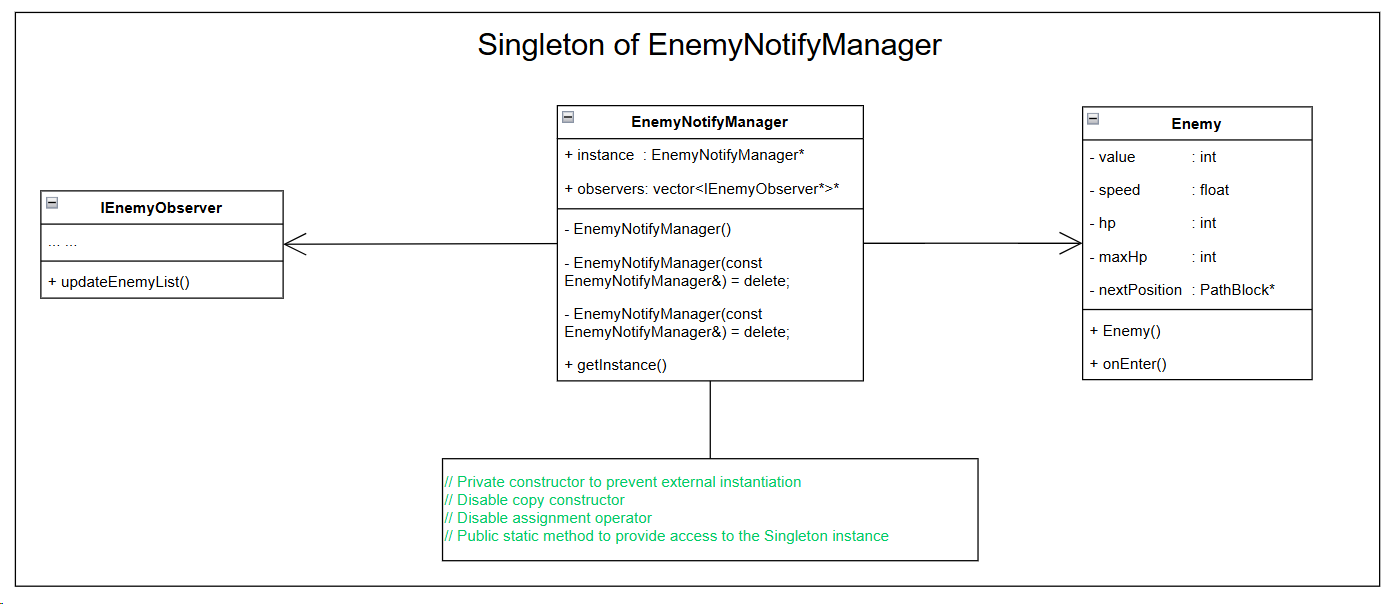
2. **Static instance variable:** To hold the single instance of the class.

3. **Public static method:** To provide access to the instance, ensuring that it is created only once.

**3.2.2 Application Scenarios**

In our code, the EnemyNotifyManager uses the Singleton Pattern to manage the list of enemy observers. It ensures that there is only one manager responsible for notifying observers when an enemy is created or removed. This centralization makes the notification process more efficient and prevents potential conflicts between multiple instances.

**3.2.3 Class Diagram**



The class implements the singleton pattern to ensure the uniqueness of EnemyNotifyManager: the constructor is made private to prevent external instantiation, and the copy constructor and assignment operator are deleted to avoid duplication or reassignment. A static pointer is used to store the single instance of the class, and a static method getInstance() is provided to access this instance. If the instance has not been created, it is initialized when the method is called. This design guarantees that only one instance of EnemyNotifyManager exists throughout the program.

**3.2.4 Key Code Demonstration**

class EnemyNotifyManager {

private:

static EnemyNotifyManager\* instance;

std::vector<IEnemyObserver\*> observers;

// Private constructor to prevent external instantiation

EnemyNotifyManager() {}

// Disable copy constructor

EnemyNotifyManager(const EnemyNotifyManager&) = delete;

// Disable assignment operator

EnemyNotifyManager& operator=(const EnemyNotifyManager&) = delete;

}

**3.3 Flyweight**

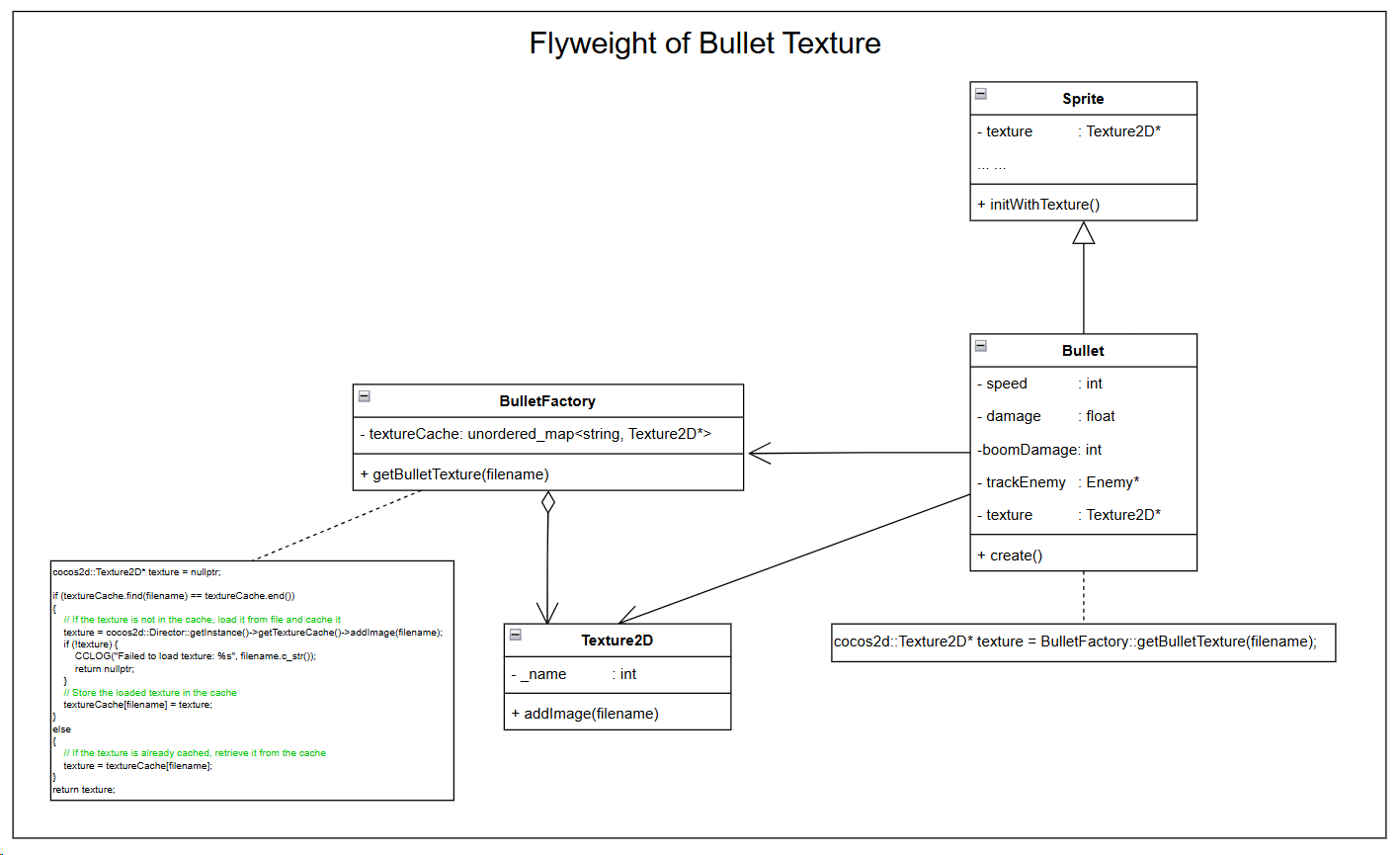
**3.3.1 Pattern Introduction**

The Flyweight Pattern is a structural design pattern that minimizes memory usage by sharing as much data as possible with similar objects. It is particularly useful when there are many objects that share common properties, as it avoids the need to duplicate shared data across all objects. This is achieved by using a centralized pool of shared objects and referencing these objects where needed.

**3.3.2 Application Scenarios**

The Flyweight Pattern is ideal for scenarios where a large number of objects share common attributes, significantly optimizing memory usage and improving performance. In the bullet system's texture management, it addresses the problem of excessive memory consumption by storing Texture2D objects in a centralized mapping table (e.g., unordered\_map) with filenames as keys, allowing textures to be reused if they already exist or newly created only when necessary. It also reduces memory usage and loading times during dynamic loading by ensuring each texture is loaded only once and reused across objects, such as bullets, particle effects, or UI elements. Furthermore, this pattern is particularly effective for low-memory devices, where shared textures minimize resource usage and enhance the game's efficiency and smoothness on resource-constrained platforms.

**3.3.3 Class Diagram**



This class diagram demonstrates the implementation of the Flyweight Pattern for bullet textures to optimize memory usage. The BulletFactory class uses a caching mechanism (e.g., unordered\_map) to centrally manage Texture2D objects, ensuring that the same texture file is loaded only once and reused across multiple bullets. The Bullet class obtains the required texture by calling methods from BulletFactory and passes the texture to the Sprite class for initialization. Texture2D represents the actual texture data, while BulletFactory serves as the core factory class responsible for creating or retrieving texture objects. This design effectively reduces memory overhead by sharing texture objects and embodies the key principle of the Flyweight Pattern, which is to enhance resource efficiency through object sharing.

**3.3.4 Key Code Demonstration**

// Static function to get a texture for a bullet by filename

static cocos2d::Texture2D\* getBulletTexture(const std::string& filename) {

cocos2d::Texture2D\* texture = nullptr;

if (textureCache.find(filename) == textureCache.end())

{

// If the texture is not in the cache, load it from file and cache it

texture = cocos2d::Director::getInstance()->getTextureCache()->addImage(filename);

if (!texture) {

CCLOG("Failed to load texture: %s", filename.c\_str());

return nullptr;

}

// Store the loaded texture in the cache

textureCache[filename] = texture;

}

else

{

// If the texture is already cached, retrieve it from the cache

texture = textureCache[filename];

}

return texture;

}

**3.4 Decorator**

**3.4.1 Pattern Introduction**

The Decorator Pattern is a structural design pattern that allows functionality to be dynamically added to a single object without affecting the behavior of other objects of the same class. It provides a flexible alternative to subclassing for extending functionality. The Decorator Pattern is typically used in scenarios where an object needs to have multiple features added, and these features can exist independently or in different combinations.

**3.4.2 Application Scenarios**

In a game, for example, bullets may have different attack types, such as poison or explosive attacks. To extend the attack modes of the base bullet class without modifying it, the Decorator Pattern is an ideal choice. It enables dynamic addition of attack modes to bullets while maintaining modular and maintainable code.

**3.4.3 Class Diagram**

**3.4.4 Key Code Demonstration**

**3.5 Observer**

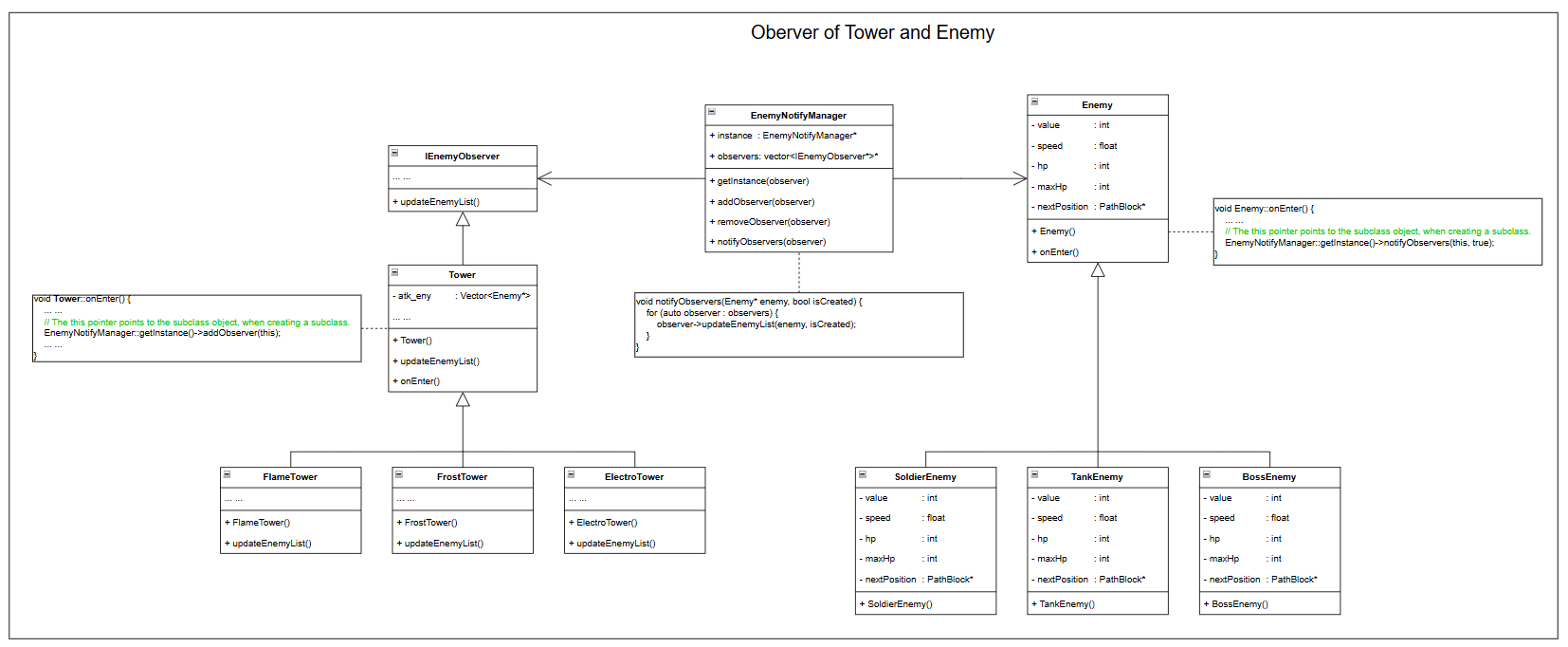
**3.5.1 Pattern Introduction**

The Observer Pattern is a behavioral design pattern that establishes a one-to-many dependency relationship between objects. When the state of one object (the "subject") changes, it notifies all its dependent objects (the "observers") to update themselves automatically. This pattern promotes loose coupling between the subject and its observers, as the subject does not need to know the exact implementation details of the observers. Instead, observers adhere to a common interface for receiving notifications, making the system flexible and extensible.

**3.5.2 Application Scenarios**

In the original design, all enemies were stored in the level, and towers attacked by iterating through the enemy list and calculating the relative positions of each enemy. This approach led to performance bottlenecks as the number of enemies increased. To optimize this, the observer pattern was introduced. Now, when an enemy is created, it notifies all relevant towers to add themselves as observers. When the enemy dies, it notifies the towers to remove itself. This way, towers no longer need to iterate through all enemies every frame. Instead, they perform attacks based on notifications from the enemies, significantly reducing unnecessary iteration and calculations. The coupling between enemies and towers is reduced, improving system performance, especially when there are many enemies. Overall, the architecture becomes more modular, easier to maintain, and more extensible.

**3.5.3 Class Diagram**



This class diagram shows the relationship between Towers and Enemies using the Observer Pattern. The EnemyNotifyManager, a singleton, manages observers (e.g., Towers) that implement the IEnemyObserver interface. When an enemy appears or dies, it notifies all observers. The Tower class registers with the EnemyNotifyManager using addObserver() and updates its status via updateEnemyList().Specific tower types, like FlameTower or FrostTower, extend Tower for extra functionality. This design decouples Towers from Enemies, ensuring flexibility and maintainability.

**3.5.4 Key Code Demonstration**

class EnemyNotifyManager {

private:

static EnemyNotifyManager\* instance;

std::vector<IEnemyObserver\*> observers;

… …

public:

// Public static method to provide access to the Singleton instance

static EnemyNotifyManager\* getInstance() {

if (instance == nullptr) {

instance = new EnemyNotifyManager();

}

return instance;

}

void addObserver(IEnemyObserver\* observer) {

observers.push\_back(observer);

CCLOG("EnemyNotifyManager: addObserver");

}

void removeObserver(IEnemyObserver\* observer) {

auto it = std::find(observers.begin(), observers.end(), observer);

if (it != observers.end()) {

observers.erase(it);

}

}

void notifyObservers(Enemy\* enemy, bool isCreated) {

for (auto observer : observers) {

observer->updateEnemyList(enemy, isCreated);

CCLOG("EnemyNotifyManager: notifyObservers");

}

}

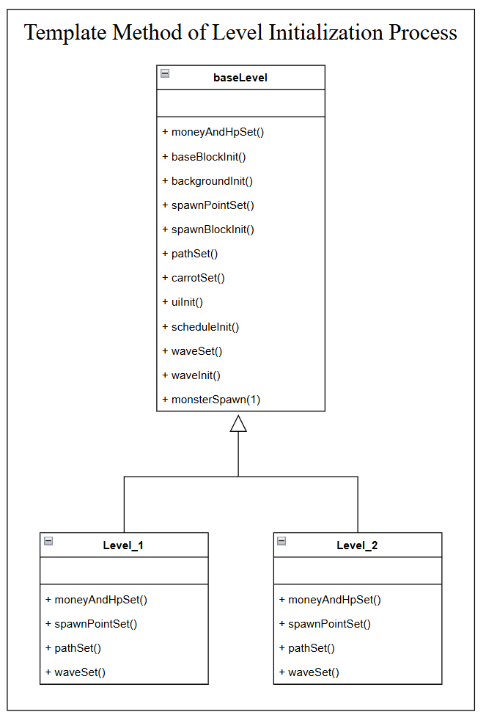
};

**3.6 Template Method**

**3.6.1 Pattern Introduction**

The Template Method Pattern defines the skeleton of an algorithm in a base class and allows its subclasses to override specific steps of the algorithm without changing its overall structure. This pattern promotes code reuse by putting the common logic in the base class while enabling customization through subclass implementation of certain steps. The core idea is to encapsulate the invariant parts of the algorithm in the base class while leaving the variant parts to the subclasses.

**3.6.2 Application Scenarios**

In our game initialization process, there is a fixed sequence of operations that need to be executed, such as setting money and HP (moneyAndHpSet()), initializing base blocks (baseBlockInit()), and so on. However, for levels that are not passed, part of the initialization process is different. To handle this, we adopt the Template Method Pattern. By using the Template Method Pattern, we can standardize the game initialization process, making it easy to maintain and extend as new types of initialization are added in the future.

**3.6.3 Class Diagram**

This class diagram illustrates the design of the level initialization process using the Template Method pattern. The baseLevel class serves as an abstract class that defines the template method for level initialization, including steps like moneyAndHpSet(), baseBlockInit(), and pathSet(), ensuring a fixed sequence of initialization steps. Level\_1 and Level\_2 inherit from baseLevel and override some of the parent class methods to implement the specific initialization logic for individual levels. The Template Method pattern ensures uniformity and code reusability in the initialization process. The base class provides a general framework, while the subclasses flexibly implement specific behaviors, catering to the needs of different levels and making it easier to extend functionality for new levels.

**3.6.4 Key Code Demonstration**

bool baseLevel::init()

{

moneyAndHpSet(); // Initialize money and HP

baseBlockInit();

backgroundInit();

spawnPointSet(); // Initialize enemy spawn point positions

spawnBlockInit();

pathSet(); // Set the enemy movement path

carrotSet();

uiInit();

scheduleInit();

waveSet(); // Initialize wave information

waveInit();

monsterSpawn(1);

return true;

}

**3.7 Object Pool**

**3.7.1 Pattern Introduction**

The Object Pool pattern is used to manage the reuse of objects that are costly to create and destroy frequently. Instead of creating and destroying objects each time they are needed, an object pool maintains a collection of reusable instances. These objects are checked out when needed and returned when no longer in use, reducing the overhead of constant object creation and garbage collection. This pattern is beneficial in scenarios where object creation is resource-intensive, and its usage improves performance by minimizing the cost associated with object allocation and deallocation.

**3.7.2 Application Scenarios**

In our game project, the Object Pool pattern is primarily used to manage the creation and destruction of objects such as bullets, enemies, explosion effects, and particle effects. By pre-creating a certain number of objects and storing them in a pool, the system can retrieve objects from the pool when needed, instead of creating new ones each time. This reduces the overhead of frequent object creation and destruction, effectively lowering memory usage and garbage collection pressure, thereby improving game performance.

**3.7.3 Class Diagram**

**3.7.4 Key Code Demonstration**

**4. Key Issues and Solutions in Refactoring**