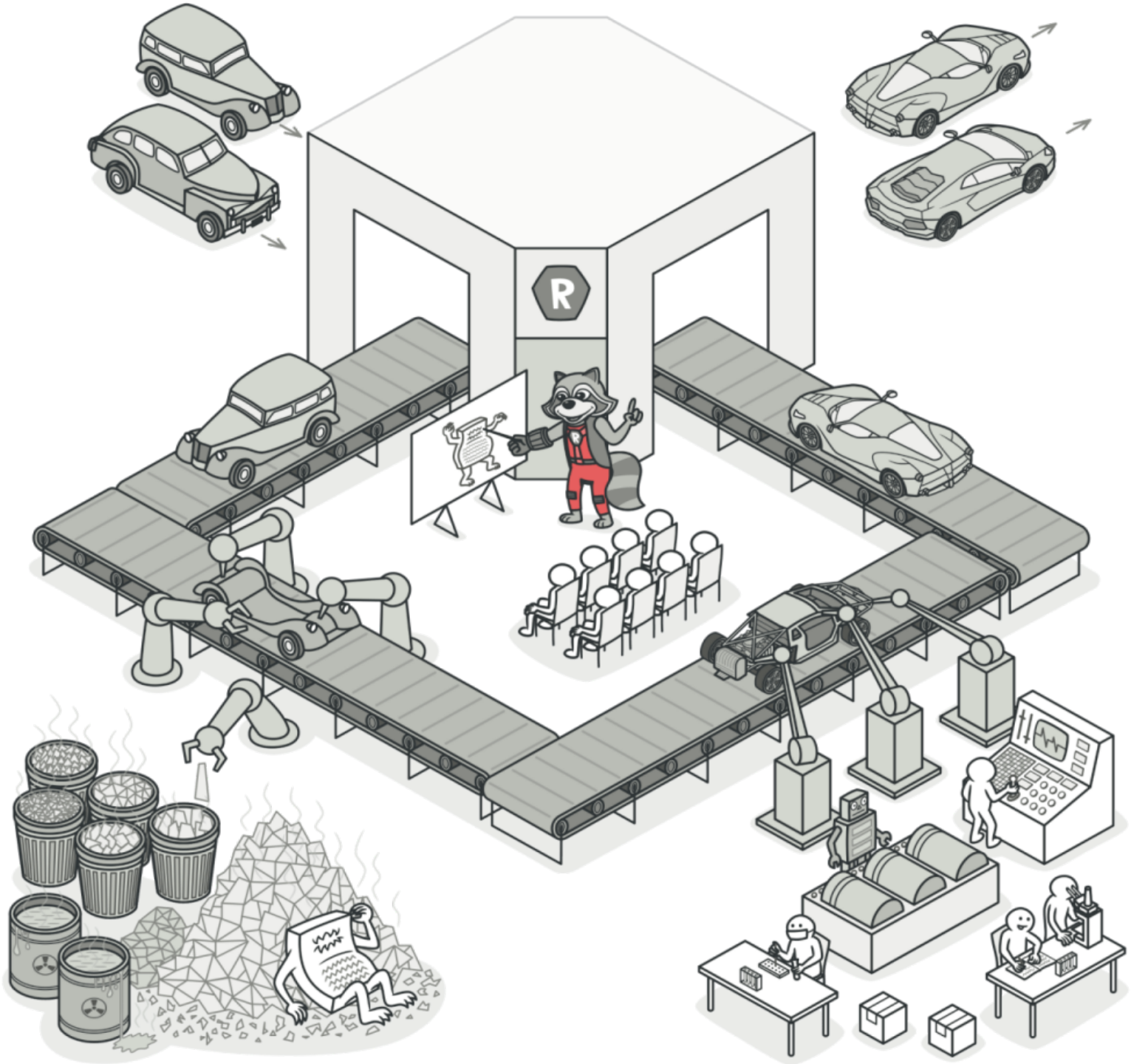


Refactoring Arknight Project

Instructor

Fan Hongfei

Team 18 Group Members

Student name	Student ID	Contact number	Email
Lin Juekai	2253744	13666915662	2253744@tongji.edu.cn
Ma Lixin	2153085	18294983673	2153085@tongji.edu.cn
Yang Junhao	2154284	18506941566	2154284@tongji.edu.cn
Xu Junyi	2252551	15197939227	2252551@tongji.edu.cn
Wu Zhiduo	2151422	13994188078	2151422@tongji.edu.cn

1. Project Introduction

1.1 Project Background

This project is the course project for *Programming Paradigms* in the Fall 2023 semester, aiming to replicate the gameplay of *Arknights*. The project is a strategy tower defense mobile game, where players take on the role of a character called the "Doctor" to lead a pharmaceutical company named "Rhodes Island" and address threats posed by catastrophic infections caused by Originium. The core gameplay involves deploying "Operators" (characters) on various maps to defend against enemy attacks. Operators are divided into multiple classes (such as Vanguard, Sniper, Medic, etc.), and players need to strategically deploy them based on enemy types and terrain.

1.2 Project Function

Key features include level selection, operator deployment, skill activation, and diverse enemy and terrain designs. The game emphasizes strategic depth and immersive gameplay, supporting interaction between multiple roles and tactical combinations. Additionally, it introduces innovative features like terrain effects and boss battles to enhance replayability.

1.2.1 Player(Ganyuan) Features

- **Ganyuan Deployment**: Operators can be placed on valid positions on the map, including ground units (e.g., Defenders) and high-platform units (e.g., Snipers and Medics).
- **Choose Ganyuan**
  - **Defenders**: Block enemies, increase attack temporarily, and specialize in ground deployment.
  - **Shooter**: Precisely target enemies, deploy on high platforms, and activate attack-boosting skills.
  - **Medics**: Heal teammates based on health levels, deploy on high platforms, and activate healing-based skills.
- **Ganyuan Skills**
  - Tracks health, attack, and defense stats.

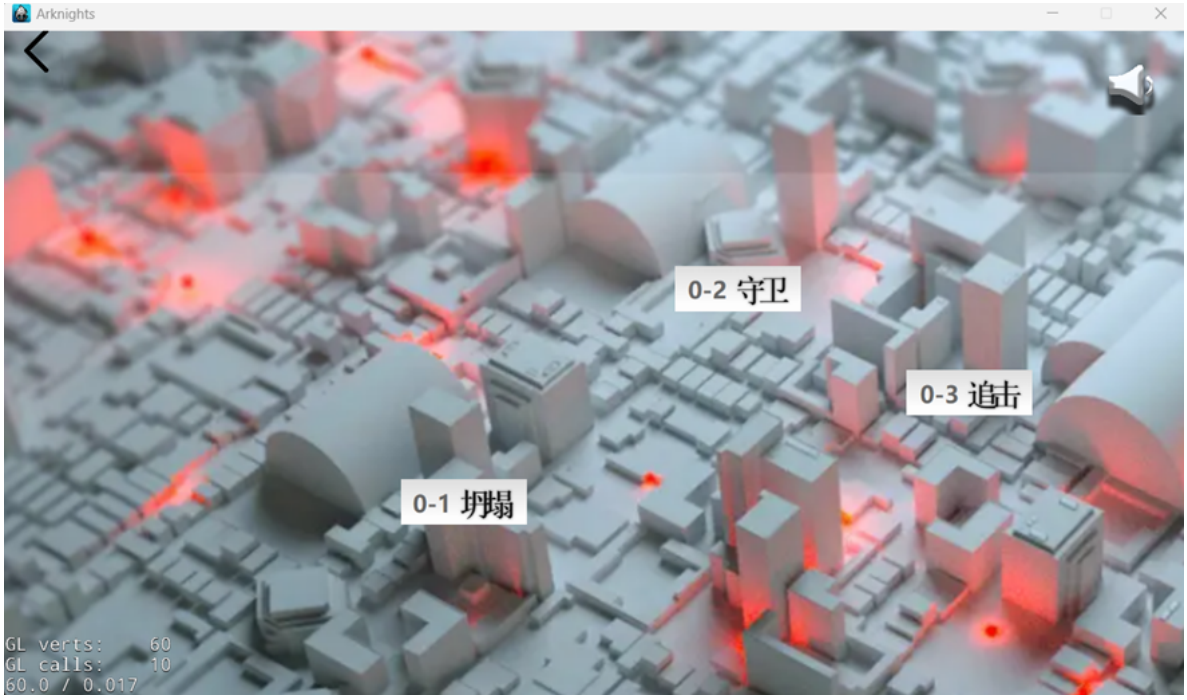
    - Supports manual activation of skills with visual feedback (e.g., attack boost for 3 seconds or health restoration).
- **Operator Revival**: Operators respawn after a cooldown period upon death.

1.2.2 Enemy Features

- **Enemy Types**
    - **Ground Enemies**: Standard enemies that follow paths on the ground.
    - **Aerial Enemies**: Requires specialized high-platform operators like Snipers to attack.
- **Enemy Skills**
    - Tracks health stats and triggers attack effects.
    - Attacks operators and progresses toward the base.
- **Wave-Based Spawning**: Enemies appear in predefined waves, increasing difficulty incrementally.

1.2.3 Other Features

- **Game Levels and Difficulty**: Supports level and difficulty selection to customize challenges.
- **Terrain Effects**: Includes varied terrains such as grass, pits, and mud, which affect deployment and behavior.
- **Victory and Defeat Screens**: Displays corresponding screens with immersive sound effects based on outcomes.
- **Pause and Restart**: Allows players to pause and restart the game at any time.
- **Immersive Sound Effects**: Enhances gameplay with battle-focused audio.



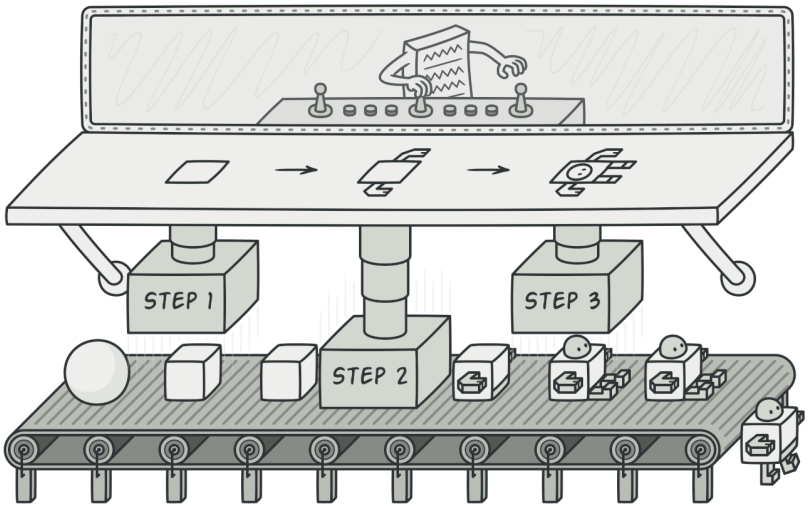
2. Creational Design Pattern - Builder

2.1 Brief of Builder Pattern

Builder Pattern is a creational design pattern that allows the construction of complex objects step by step. It separates the construction of a product from its representation, so that the same construction process can create different representations. The builder pattern helps in constructing objects in a controlled manner, ensuring that all the components are correctly initialized.

The builder pattern typically involves a director class that controls the construction process and a builder interface that defines the methods for creating different parts of the product. Concrete builder classes implement these methods and return the finished product. The client interacts with the director to construct the product, making it easy to construct complex objects without exposing the construction details.

Builder pattern has the advantages of constructing complex objects step by step, allowing for flexibility in product creation and enhancing code readability. It separates the construction process from the object representation, ensuring that different representations of the same product can be easily created. By using builder classes, you can avoid a large number of constructor parameters, improve maintainability, and keep the client code simple. Next, we will introduce the details of refactoring using builder pattern in our project.



2.2 Reasons for Refactoring

In our project, the enemy class (such as `Enemy1`) inherits from `EnemyBase` to implement the creation and attribute setting of different enemy types. Although different types of enemies share the same behavior and basic attributes, there are differences in appearance (e.g., image resources) and specific values (e.g., health, damage). In the original implementation, we placed the enemy creation logic inside the `EnemyBase` class and customized the properties by overriding creation methods in different subclasses. However, this design had the following issues:

1. **Violation of the Single Responsibility Principle (SRP):** The `EnemyBase` class was responsible for both creating and initializing enemy objects and managing their behaviors and combat logic. This made the class overly complex and violated SRP. According to SRP, a class should have only one reason to change, but in the current design, `EnemyBase` had multiple reasons to change—handling enemy creation and managing enemy behavior, which were not directly related.
2. **Code Complexity:** In the original code, each enemy's initialization process was quite complex, and the initialization logic was scattered across several methods. For example, the `init` function not only called the parent class's initialization method but also invoked the `initial` method, which in turn called the `setDefaultData` method. This resulted in redundant and meaningless logic scattered across methods.

   Original Code:

```
// air
bool Enemy1::init()
{
    if (!EnemyBase::init("Pictures/enemy_air.png"))
    {
        return false;
    }
    initial();
    return true;
}

Enemy1* Enemy1::create()
{
    Enemy1* e1 = new Enemy1();
    if (e1 && e1->init())
    {
        e1->autorelease();
        return e1;
```

```
    }
    CC_SAFE_DELETE(e1);
    return nullptr;
}

void Enemy1::setDefaultData() {
    setType(ENEMY1_TYPE);
    scope = Enemy1Scope;
    setLethality(Enemy1Lethality);
    setHp(Enemy1Hp);
    setHealth(Enemy1Hp);
    setDefence(Enemy1Defence);
    setRunSpeed(Enemy1RunSpeed);
    setAlive(true);
    setIntervalTime(ShieldIntervalTime);

    setLastAttackTime(GetCurrentTime() / 1000.f);
    setIsGround(true);

}
void Enemy1::initial()
{
    setDefaultData();
}
```

3. **Poor Code Maintainability:** Due to the differences in the attributes and behaviors of the enemies mainly revolving around resources (like image paths) and values (like health, attack power, etc.), the current design required duplicating similar code in multiple places every time a new enemy type was added. If any attribute needed modification, changes had to be made in multiple classes and methods, leading to poor maintainability and increased chances of introducing bugs.
4. **Not Ideal for Extension and Reusability:** The creation logic for enemies was tightly coupled, making it cumbersome to extend with new enemy types. Every time a new enemy was added, it was necessary to inherit from `EnemyBase` and override the initialization and attribute-setting methods, many of which were repetitive, leading to low code reuse and poor extensibility.

   To address the above issues, we decided to refactor the design using the Builder pattern to decouple the construction of enemy objects from their complex behaviors and combat logic. This approach makes the object construction process clearer, more flexible, and aligns with the **Single Responsibility Principle**.

2.3 Refactoring Details

1. Define Generic Construction Steps

In the `EnemyCreateBase` class, we declare all possible construction steps. Based on the characteristics of enemies in the game, we divide the steps into: creating the enemy, initializing unique attributes, performing common initialization, and post-creation steps. We also declare the `getResult` method to return the final constructed enemy object.

```
class EnemyCreateBase
{
private:
    EnemyBase *e;
```

```
public:
    EnemyCreateBase();

    // Step 1: Create a new object
    virtual void new_enemy();

    // Step 2: Type-specific initialization (to be implemented by subclasses)
    virtual bool type_init() = 0;

    // Step 3: Common initialization
    virtual void general_init();

    // Step 4: After initialization (memory management)
    virtual void after_init();

    // Get the resulting enemy object
    virtual EnemyBase *get_result();
};
```

Functions to implement:

```
void EnemyCreateBase::new_enemy()
{
    e = new EnemyBase();
}

void EnemyCreateBase::general_init()
{
    e->setAlive(true);
    setLastAttackTime(GetCurrentTime() / 1000.f);
    setIsGround(true);

    // Initialize health bar
    healthBar = Bar::create(EStateType::Health, e->getHealth());
    auto size = e->getBoundingBox().size;
    healthBar->setScaleX(0.5);
    healthBar->setScaleY(0.7);
    healthBar->setAnchorPoint(Vec2::ANCHOR_MIDDLE);
    healthBar->setPosition(Vec2(200, 450));
    e->addChild(healthBar);
}

void EnemyCreateBase::after_init()
{
    e->autorelease();
}

EnemyBase *EnemyCreateBase::get_result()
{
    return e;
}
```

2. Create Specific Builder Classes for Each Product Representation

In the first step, we introduced the `type_init` pure virtual function, which implements the unique property settings for each enemy. This method will be implemented in concrete builder classes, such as `AirEnemyCreate` for air enemies. Below is an example of how the `AirEnemyCreate` class is declared and implemented.

```
class AirEnemyCreate : public EnemyCreateBase
{
public:
    virtual bool type_init() override;
};
```

Implementation:

```
bool AirEnemyCreate::type_init()
{
    if (!e->Sprite::initWithFile("Pictures/enemy_air.png"))
    {
        return false;
    }

    e->setType(ENEMY1_TYPE);
    e->scope = Enemy1Scope;
    e->setLethality(Enemy1Lethality); // Killing power
    e->setHp(Enemy1Hp);               // Max health
    e->setHealth(Enemy1Hp);           // Current health
    e->setDefence(Enemy1Defence);     // Defense
    e->setRunSpeed(Enemy1RunSpeed);   // Movement speed
    e->setIntervalTime(ShieldIntervalTime); // Attack interval time
    return true;
}
```

Similarly, we implement other types of enemies, such as `GroundEnemyCreate1` and `GroundEnemyCreate2`:

```
bool GroundEnemyCreate1::type_init()
{
    if (!e->Sprite::initWithFile("Pictures/enemy_ground1.png"))
    {
        return false;
    }

    e->setType(ENEMY2_TYPE);
    e->scope = Enemy2Scope;
    e->setLethality(Enemy2Lethality);
    e->setHp(Enemy2Hp);
    e->setHealth(Enemy2Hp);
    e->setDefence(Enemy2Defence);
    e->setIntervalTime(Enemy2IntervalTime);
    e->setRunSpeed(Enemy2RunSpeed);
    e->setIsBlock(false);
    return true;
}

bool GroundEnemyCreate2::type_init()
{
```

```
    if (!e->Sprite::initWithFile("Pictures/enemy_ground2.png"))
    {
        return false;
    }

    e->setType(ENEMY3_TYPE);
    e->scope = Enemy3Scope;
    e->setLethality(Enemy3Lethality);
    e->setHp(Enemy3Hp);
    e->setHealth(Enemy3Hp);
    e->setDefence(Enemy3Defence);
    e->setIntervalTime(Enemy3IntervalTime);
    e->setRunSpeed(Enemy3RunSpeed);
    e->setIsBlock(false);
    return true;
}
```

3. Create the Director Class

The `EnemyDirector` class is responsible for orchestrating the construction process by passing the appropriate builder. This class encapsulates all construction logic so that the client only needs to focus on the construction result without worrying about the details.

Declaration:

```
class EnemyDirector
{
private:
    EnemyCreateBase *enemyCreate;

public:
    EnemyDirector(EnemyCreateBase *enemyCreate);

    EnemyBase *construct();
};
```

Implementation:

```
EnemyDirector::EnemyDirector(EnemyCreateBase *enemyCreate) :
enemyCreate(enemyCreate) {}

EnemyBase *EnemyDirector::construct()
{
    enemyCreate->new_enemy();
    enemyCreate->type_init();
    enemyCreate->general_init();
    enemyCreate->after_init();
    return enemyCreate->get_result();
}
```

4. Remove Enemy Creation Methods from EnemyBase

At this point, we have implemented the Builder-based approach for creating enemies. The creation and initialization functions in the `EnemyBase` class (`create` and `init`) should be removed to achieve proper responsibility separation.

5. Modify Client Code

Finally, we refactor the game logic responsible for creating enemies. Instead of directly calling the initialization methods for different enemy types, we use the Director and the specific Builders.

Original client code example:

```
void GameLayer::addSceneEnemy(float dt)
{
    GameManager* instance = GameManager::getInstance();

    for (int i = 0; i < waveQ.size(); i++)
    {
        auto groupEnemy = waveQ.at(i);

        EnemyType et;

        if (groupEnemy == NULL)
        {
            return;
        }
        else if (!groupEnemy->enemies.empty())
        {
            et = groupEnemy->enemies.front();

            std::vector<Vec2> rd = instance->roadsPosition.at(et.road - 1);

            if (et.type == ENEMY1_TYPE) {
                auto enemy = Enemy1::create();
                enemy->setEntered(false);
                enemy->setLastAttackTime(this->getNowTime());
                enemy->setRoad(et.road);
                enemy->setFirstPose(rd.front());
                enemy->setLastPose(rd.back());
                enemy->setCurPose(rd.front());
                enemy->setNextPose(rd.at(1));
                enemy->setPtr(0);
                enemy->setPosition(rd.front());
                enemy->setScale(0.125);
                this->addChild(enemy, 10);
                instance->enemyVector.insert(0, enemy);

                this->addChild(enemy, 10);
                instance->enemyVector.pushBack(enemy);
            }
            else if (et.type == ENEMY2_TYPE) {
                instance->enemyVector.pushBack(enemy);

                auto enemy = Enemy2::create();
                enemy->setEntered(false);
                enemy->setLastAttackTime(this->getNowTime());
                enemy->setRoad(et.road);
                enemy->setFirstPose(rd.front());
                enemy->setLastPose(rd.back());
                enemy->setCurPose(rd.front());
                enemy->setNextPose(rd.at(1));
                enemy->setPtr(0);
                enemy->setPosition(rd.front());
                enemy->setScale(0.125);
```

```
                this->addChild(enemy, 10);
                instance->enemyVector.insert(0, enemy);
            }
            else if (et.type == ENEMY3_TYPE) {
                auto enemy = Enemy3::create();
                enemy->setEntered(false);
                enemy->setLastAttackTime(this->getNowTime());
                enemy->setRoad(et.road);
                enemy->setFirstPose(rd.front());
                enemy->setLastPose(rd.back());
                enemy->setCurPose(rd.front());
                enemy->setNextPose(rd.at(1));
                enemy->setPtr(0);
                enemy->setPosition(rd.front());
                enemy->setScale(0.125);
                this->addChild(enemy, 10);
                instance->enemyVector.insert(0, enemy);
            }

            groupEnemy->enemies.erase(groupEnemy->enemies.begin());

        }
        else
        {
            waveQ.eraseObject(groupEnemy);
        }
    }
}
```

As you can see, the core logic of this method is as follows: traverse the enemy wave queue waveQ, create the corresponding enemy according to the enemy type and initialise its attributes, add the created enemy to the game scene and the enemy list, update the state of the enemy in the enemy group, remove the processed enemy from the queue, and remove the empty enemy group that has been processed.

The idea of refactoring is as follows: after getting the enemy type, choose the appropriate creator according to the enemy type:

Enemy type	Concrete Builder
ENEMY1_TYPE (Air Enemy)	AirEnemyCreate
ENEMY2_TYPE (first ground enemy)	GroundEnemyCreate1
ENEMY3_TYPE (second ground enemy)	GroundEnemyCreate2

After obtaining the enemy type, the Director class is used to construct the enemy. Subsequent initialization, adding to the scene, and other operations are handled as follows:

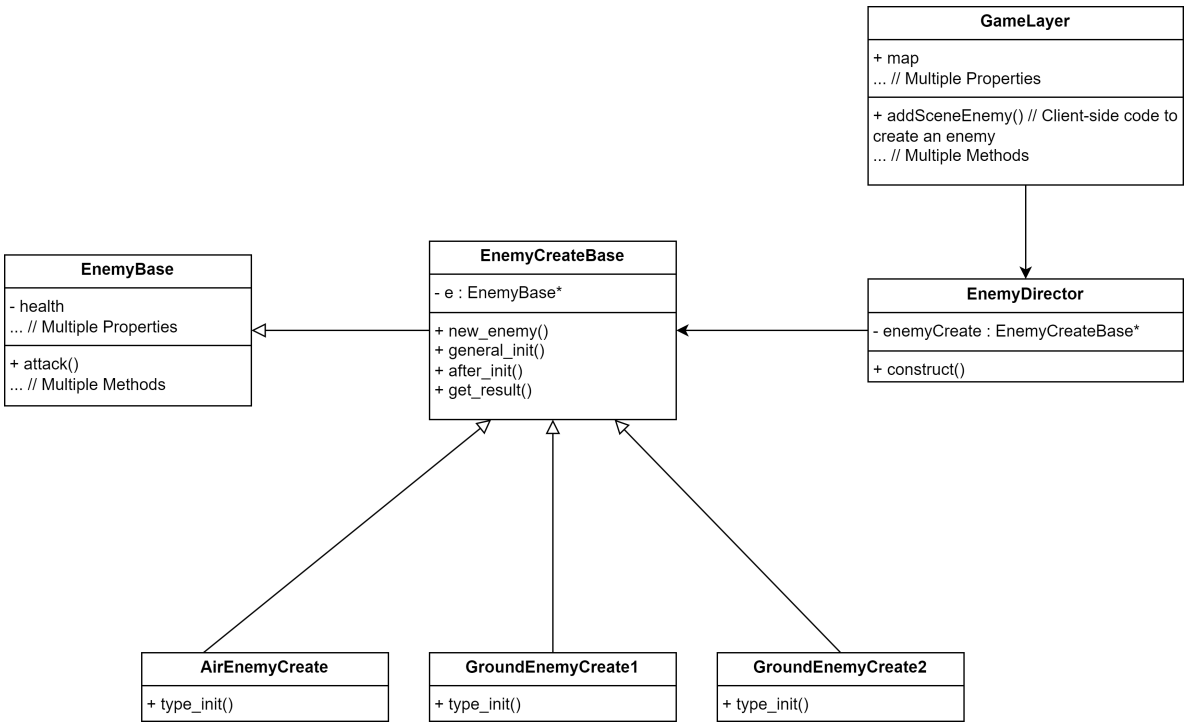
```
void GameLayer::addSceneEnemy(float dt)
{
    GameManager* instance = GameManager::getInstance();

    for (int i = 0; i < waveQ.size(); i++)
    {
        auto groupEnemy = waveQ.at(i);
```

```
        if (groupEnemy == NULL || groupEnemy->enemies.empty())
        {
            waveQ.eraseObject(groupEnemy);
            continue;
        }

        auto et = groupEnemy->enemies.front();
        std::vector<Vec2> rd = instance->roadsPosition.at(et.road - 1);

        EnemyCreateBase* enemyCreator = nullptr;

        // Choose a director class based on the enemy type
        switch (et.type)
        {
        case ENEMY1_TYPE:
            enemyCreator = new AirEnemyCreate();
            break;
        case ENEMY2_TYPE:
            enemyCreator = new GroundEnemyCreate1();
            break;
        case ENEMY3_TYPE:
            enemyCreator = new GroundEnemyCreate2();
            break;
        default:
            break;
        }

        if (enemyCreator != nullptr)
        {
            EnemyDirector enemyDirector(enemyCreator);  // Use the Director
class to create enemies
            EnemyBase* enemy = enemyDirector.construct();  // Spawn enemies

            // Set the enemy's initial position and status
            enemy->setEntered(false);
            enemy->setLastAttackTime(this->getNowTime());
            enemy->setRoad(et.road);
            enemy->setFirstPose(rd.front());
            enemy->setLastPose(rd.back());
            enemy->setCurPose(rd.front());
            enemy->setNextPose(rd.at(1));
            enemy->setPtr(0);
            enemy->setPosition(rd.front());
            enemy->setScale(0.125);

            this->addChild(enemy, 10);
            instance->enemyVector.insert(0, enemy);

            groupEnemy->enemies.erase(groupEnemy->enemies.begin());
        }
        else
        {
            CC_SAFE_DELETE(enemyCreator);
        }
    }
}
```

Through this approach, the `Builder pattern` is introduced, and the main logic of `addSceneEnemy` no longer handles the specific creation process of enemy types. The code becomes simpler, easier to maintain, and significantly improves readability, extensibility, and maintainability.

2.4 UML Class Diagram

`EnemyCreateBase` serves as a Builder interface, defining the standard steps for constructing an `Enemy` object. Each enemy-specific Builder class inherits from `EnemyCreateBase` and implements its own construction steps. This class is responsible for breaking down complex construction logic into multiple steps, making the object creation process more flexible and configurable.



Steps:

- `new_enemy()`: Creates a new `EnemyBase` object.
- `type_init()`: Initializes the enemy's type-specific attributes (implemented by subclasses).
- `general_init()`: Executes common initialization logic (e.g., setting basic attributes, initializing health bars).
- `after_init()`: Handles post-construction tasks (e.g., memory management).
- `get_result()`: Returns the fully constructed enemy object.

Each concrete Builder class (e.g., `AirEnemyCreate`, `GroundEnemyCreate1`, `GroundEnemyCreate2`) is responsible for constructing a specific type of enemy. These classes inherit from `EnemyCreateBase` and implement the `type_init()` method to initialize attributes like type, speed, health, etc.

The `EnemyDirector` class is the Director in the Builder pattern, responsible for coordinating the construction process and ensuring that enemy objects are created in the correct order. It utilizes specific implementations of `EnemyCreateBase` to create different types of enemies, simplifying the client's task. With `EnemyDirector`, the client doesn't need to know the details of the construction process but only needs to pass in the appropriate `Concrete Builder` to obtain a fully configured enemy object.

The `GameLayer` class acts as the client, calling the `EnemyDirector` to create enemies. By providing enemy type information, it uses the `EnemyDirector` to get the constructed enemy object and adds it to the scene.

2.5 Benefits of Refactoring

1. Clear Responsibility Separation

Through refactoring, the system achieves clear separation of responsibilities.

- The `EnemyBase` class focuses solely on handling the behavior logic of enemies, such as attacking, defending, and moving, and no longer takes on the responsibility of creating and initializing enemy objects. This responsibility has been delegated to the `EnemyCreateBase` class, which serves as a Builder interface.
- `EnemyCreateBase` is responsible for the creation process, with specific Builder classes (e.g., `AirEnemyCreate` and `GroundEnemyCreate1`) implementing the initialization of specific enemy types.

This ensures that each class has a single, well-defined responsibility: `EnemyBase` focuses on behavior, while `EnemyCreateBase` handles object creation. This eliminates mixed responsibilities and adheres to the **Single Responsibility Principle (SRP)**, improving code readability, maintainability, and extensibility.

2. Strong Extensibility

Adding new enemy types becomes very easy. For example:

1. Create a new Builder class for the new enemy type.
2. Add a corresponding case to the `switch` statement in the `addSceneEnemy` method.

This extension approach complies with the **Open/Closed Principle**: the system is open for extension but closed for modification. Existing code doesn't need to be modified; instead, new functionality is added by creating new Builder classes.

3. Enhanced Readability

- Using the Builder and Director patterns, the code becomes clearer and easier to understand. Each class and method has a specific purpose, avoiding the complex logic and repeated initialization steps seen in the original code.
- The logic of the `addSceneEnemy` method is straightforward, with its main responsibility being to fetch enemy groups, select the appropriate Builder and Director for construction, and set enemy states.

4. Flexible Extension and Customization

- Through the Builder pattern, each type of enemy can have a customizable construction process. Different enemy types can be initialized with unique methods, attribute settings, and behaviors. When adding a new enemy type, there's no need to modify existing enemy creation code; simply create a new Builder for the new type.
- The Director pattern allows flexible combinations of Builders, making it easy to switch enemy construction methods by adjusting the conditions in the `addSceneEnemy` method.

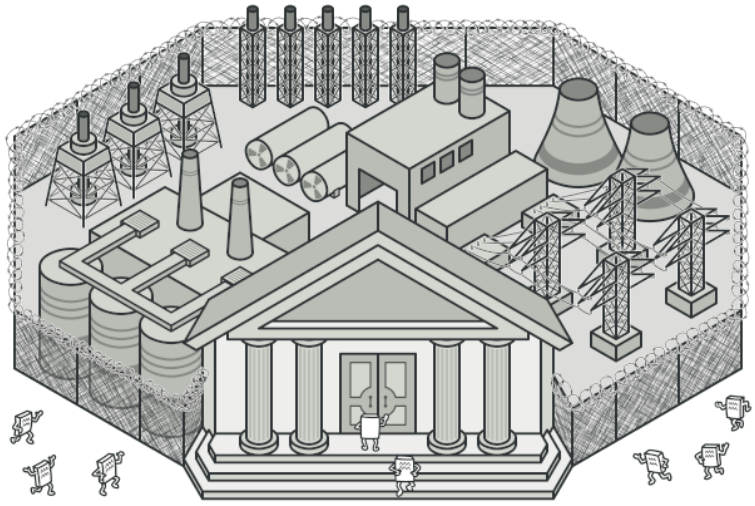
3. Structural Design Pattern

3.1 Facade Design Pattern

3.1.1 Brief of Facade Pattern

The Facade Pattern is a structural design pattern that provides a unified and simplified interface to a complex subsystem. It serves as a high-level entry point, encapsulating the intricate details of the subsystem and reducing the dependency between clients and the subsystem components.

The Facade Pattern typically involves creating a facade class that interacts with various subsystems, consolidating their functionalities into a streamlined interface. This pattern simplifies the interaction for the client while ensuring the internal complexity of the subsystem remains hidden.



To implement the Facade Pattern, the first step is identifying the subsystems and their interactions. Then, a facade class is designed to act as a mediator, delegating client requests to the appropriate subsystem methods. The facade provides only the necessary functionality, masking the detailed implementation. Finally, the client interacts solely with the facade, eliminating the need to work with individual subsystem components directly.

The Facade Pattern offers several advantages, including improving code readability, reducing coupling between the client and subsystems, and simplifying complex operations into a single, cohesive interface. In the following sections, we will explore how the Facade Pattern has been applied to refactor the `GameLayer` class in our project : the `GameFacade` class serves as a Facade that encapsulates the interactions between `EnemyManager`, `GameLogicManager`, and `UIManager`, providing a unified interface for `GameLayer`.

3.1.2 Reason for Refactoring

In our game project, the `GameLayer` serves as the primary manager for coordinating various game elements, including player characters, enemies, bullets, and environmental interactions. Now take a look at our `GameLayer` class first:

```
// Before refactory
// GameLayer class
class GameLayer : public Layer
{
public:
    int mapType;

    float startTime;
    float getNowTime() const;
    float getInterval(float a, float b) const;

    virtual void update(float dt) override;

    int totalEnemies;
```

```
    Vector<Wave*> waveQ;
    void addSceneEnemy(float dt);
    bool allWavesSuccessful();
    void addWaveEnemy(float showtime, std::initializer_list<EnemyType> il);
    void initWave();
    Wave* findEarliestWave();
    void logic();
    void runEnemies();
    void runmedical();
    void updatemoney(float dt);

    void bulletFlying(float dt);
    void removeBullet(Bullet* pSender);
protected:
    float interval;

    SpriteBatchNode* spriteSheet;
    TMXTiledMap* map;
    TMXLayer* bgLayer;
    TMXObjectGroup* towers;
    TMXObjectGroup* grounds;
    std::vector<Vec2> towers_path;
    std::vector<Vec2> grounds_path;

    float offX;
    int waveCounter;
    int moneyL;
    int star;
    Label* moneyLabel;
    Label* groupLabel;
    Label* groupTotalLabel;
    Layer* toolLayer;

    Sprite* sprite_money;
    Sprite* star3;
    Sprite* star2;
    Sprite* star1;
    Sprite* star0;
    void initToolLayer();

    bool isTouchEnable;
    void addTowerChoosePanel(Point position);
    Point convertTotileCoord(Point position);
    Point convertToMatrixCoord(Point position);
    void checkAndAddTowerPanle(Point position);
    void CollisionDetection();

    void menuBackCallback(Ref* pSender);

    Point towerPos;

    void win();
    void lose();
};
```

However, the current implementation of `GameLayer` suffers from several critical issues:

1. **High Complexity and Coupling**
   The `GameLayer` contains a vast amount of logic to handle multiple aspects of the game, such as collision detection, object spawning, animations, and score management. This results in a highly complex and tightly coupled class, making it challenging to maintain, debug, and extend the code.
2. **Violation of the Single Responsibility Principle**
   Currently, the `GameLayer` takes on too many responsibilities, including managing game objects, handling user input, controlling game state transitions, and managing resources. This not only bloats the class but also violates the Single Responsibility Principle, leading to potential instability when changes are required in any specific functionality.
3. **Difficulty in Testing and Reuse**
   Due to its monolithic nature, the `GameLayer` cannot be easily tested in isolation. Any changes to the game mechanics require thorough regression testing, increasing the development time and risk of introducing bugs. Moreover, it is hard to reuse parts of the `GameLayer` logic in other parts of the project or future projects.
4. **Lack of Abstraction**
   The `GameLayer` directly interacts with various subsystems, such as physics, rendering, and audio. This lack of abstraction exposes the internal workings of these subsystems to the `GameLayer`, resulting in code that is harder to understand and modify.
5. **Limited Scalability**
   As the game grows in complexity, adding new features or modifying existing ones becomes increasingly cumbersome. Without a clear separation of concerns, expanding the game logic often requires modifying multiple parts of the `GameLayer`, leading to errors and unanticipated side effects.

To address these challenges, we propose refactoring the `GameLayer` using the **Facade Pattern**. By introducing a facade, we can encapsulate the interactions between the `GameLayer` and the subsystems into a unified interface. This refactoring will reduce complexity, improve modularity, and provide a clear structure for future extensions and maintenance.

3.1.3 Refactoring Details

The `GameLayer` currently has a monolithic structure, directly managing various game subsystems such as player interactions, enemy behavior, bullet management, and game state updates. This refactoring introduces the **Facade Pattern** to decouple and streamline these subsystems, making the code easier to maintain, extend, and test.

1. Decompose management class and generation subsystem

The facade pattern provides a unified interface for a set of interfaces within a subsystem, making it easier to use. We can decompose the functionality in `GameLayer` into several management classes, such as `EnemyMnager`, `GameLogicManager`, and `UIManager`, and then call the corresponding functionality through these management classes in `GameLayer`. `GameLayer` can then utilize these management classes to encapsulate the detailed operations of each subsystem, delegating tasks to the appropriate manager through a simplified and consistent interface. This not only promotes modularity and separation of concerns but also enhances code readability, maintainability, and scalability.

```
// Define EnemyManager
class EnemyManager {
public:
    EnemyManager(GameManager* gameManager);
    void addSceneEnemy(float dt);
    void runEnemies();
```

```
    void removeBullet(Bullet* pSender);
    void bulletFlying(float dt);

private:
    GameManager* instance;
};

// Define GameLogicManager
class GameLogicManager {
public:
    GameLogicManager(GameManager* gameManager);
    void initWave(int mapType);
    void logic(float startTime, float nowTime, cocos2d::Vector<Wave*>& waveQ);
    bool allWavesSuccessful();
    Wave* findEarliestWave();
    void win(int star, cocos2d::Vector<Wave*>& waveQ);
    void lose(int& star);

private:
    GameManager* instance;
};

// Define UIManager
class UIManager {
public:
    UIManager(GameManager* gameManager, cocos2d::Layer* parentLayer);
    void initToolLayer();
    void updateMoney(int& moneyL, cocos2d::Label* moneyLabel);

private:
    GameManager* instance;
    cocos2d::Layer* toolLayer;
    cocos2d::Layer* parentLayer;
};
```

In `GameLayer`, we transfer the logic related to enemies to `EnemyMnager` and create an instance of `EnemyMnager` in `GameLayer`. Similarly, create `GameLogicManager` and `UIManager` to manage game logic and UI related functions. This can make the code clearer and responsibilities more defined.

2. Provide simpler interface based on existing subsystems

Create a `GameFacade` class to encapsulate the interaction between `EnemyManager`, `GameLogicManager`, and `UIManager`, providing a unified interface for `GameLayer`. This will make the interaction between `GameLayer` and subsystems more concise and in line with the definition of the Facade pattern.

```
// Define GameFacade
class GameFacade {
public:
    GameFacade(GameManager* gameManager, cocos2d::Layer* parentLayer);

    void initToolLayer();

    void addSceneEnemy(float dt);

    void bulletFlying(float dt);
```

```
    void runEnemies();

    void updateMoney(int& moneyL, cocos2d::Label* moneyLabel);

    void processGameLogic(float startTime, float nowTime,
cocos2d::Vector<Wave*>& waveQ);

    void checkWin(int& star, cocos2d::Vector<Wave*>& waveQ);

    void checkLose(int& star);

    void update(float startTime, float nowTime, int& star,
cocos2d::Vector<Wave*>& waveQ, int& moneyL, cocos2d::Label* moneyLabel);

private:
    EnemyManager* enemyManager;
    GameLogicManager* gameLogicManager;
    UIManager* uiManager;
};
```

This class acts as a central point that coordinates and simplifies the communication between the different subsystems, ensuring that `GameLayer` only interacts with the `GameFacade` rather than directly with each individual manager. The implementation of the core functionality, such as handling enemy behaviors through `EnemyManager`, managing game rules with `GameLogicManager`, and updating the UI via `UIManager`, can be included in the respective manager classes. The `GameFacade` exposes high-level methods, which internally invoke the appropriate actions in the respective subsystems, abstracting away the complexity and providing a cleaner, more maintainable interface for `GameLayer` to interact with. This approach ensures the extensibility and flexibility of the code, making future modifications and additions to any subsystem easier without affecting the other parts of the system.

3. Declare and implement interfaces in the new Facade class

We created the class and declared all required methods within it, such as `initToolLayer`, `addSceneEnemy`, `bulletFlying`, `runEnemies`, `updateMoney`, `processGameLogic`, `checkWin`, `checkLose`, and `update`. These methods will encapsulate calls to various subsystems and provide a simple interface for `GameLayer` to use.

```
// GameFacade.cpp
void GameFacade::initToolLayer() {
    uiManager->initToolLayer();
}

void GameFacade::addSceneEnemy(float dt) {
    enemyManager->addSceneEnemy(dt);
}

void GameFacade::bulletFlying(float dt) {
    enemyManager->bulletFlying(dt);
}

void GameFacade::runEnemies() {
    enemyManager->runEnemies();
}
```

```
void GameFacade::updateMoney(int& moneyL, cocos2d::Label* moneyLabel) {
    uiManager->updateMoney(moneyL, moneyLabel);
}

void GameFacade::processGameLogic(float startTime, float nowTime,
cocos2d::Vector<Wave*>& waveQ) {
    gameLogicManager->logic(startTime, nowTime, waveQ);
}

void GameFacade::checkWin(int& star, cocos2d::Vector<Wave*>& waveQ) {
    gameLogicManager->win(star, waveQ);
}

void GameFacade::checkLose(int& star) {
    gameLogicManager->lose(star);
}

void GameFacade::update(float startTime, float nowTime, int& star,
cocos2d::Vector<Wave*>& waveQ, int& moneyL, cocos2d::Label* moneyLabel) {
    processGameLogic(startTime, nowTime, waveQ);
    updateMoney(moneyL, moneyLabel);
    runEnemies();
    checkLose(star);
    checkWin(star, waveQ);
}
```

4. Redirect client code calls to the corresponding objects in the subsystem

In the implementation of `GameFacade`, we redirect each method call to the corresponding subsystem. For example, the `addSceneEnemy` method calls `enemyManager ->addSceneEnemy (dt)`, the `processGameLogic` method calls `gameLogicManager ->logic (startTime, nowTime, waveQ)`, and so on. In this way, `GameLayer` only needs to call the methods of `GameFacade` without directly interacting with the subsystem.

```
// GameLayer.cpp
bool GameLayer::init()
{

    gameFacade->initToolLayer();
    gameFacade->processGameLogic(startTime, nowTime, waveQ);

    schedule(CC_SCHEDULE_SELECTOR(GameLayer::addSceneEnemy), interval);
    schedule(CC_SCHEDULE_SELECTOR(GameLayer::bulletFlying), 0.1f);
    schedule(CC_SCHEDULE_SELECTOR(GameLayer::updatemoney), 1.0f);
    scheduleUpdate();

    return true;
}

void GameLayer::addSceneEnemy(float dt)
{
    gameFacade->addSceneEnemy(dt);
}

void GameLayer::bulletFlying(float dt)
{
    gameFacade->bulletFlying(dt);
```

```
}

void GameLayer::runEnemies()
{
    gameFacade->runEnemies();
}

void GameLayer::removeBullet(Bullet* pSender)
{
    gameFacade->removeBullet(pSender);
}

void GameLayer::updatemoney(float dt)
{
    gameFacade->updateMoney(moneyL, moneyLabel);
}

void GameLayer::update(float dt)
{
    float nowTime = GetCurrentTime() / 1000.f;
    gameFacade->update(startTime, nowTime, star, waveQ, moneyL, moneyLabel);
}
```

5. Initialize the subsystem based on the Facade class

In the constructor of `GameFacade`, we initialize instances of all subsystems. In this way, `GameLayer` does not need to worry about subsystem initialization and lifecycle management, and only needs to use the interfaces provided by `GameFacade`.

```
// GameFacade.cpp
GameFacade::GameFacade(GameManager* gameManager, cocos2d::Layer* parentLayer) {
    enemyManager = new EnemyManager(gameManager);
    gameLogicManager = new GameLogicManager(gameManager);
    uiManager = new UIManager(gameManager, parentLayer);
}
```

6. Extract new specialized Facade class

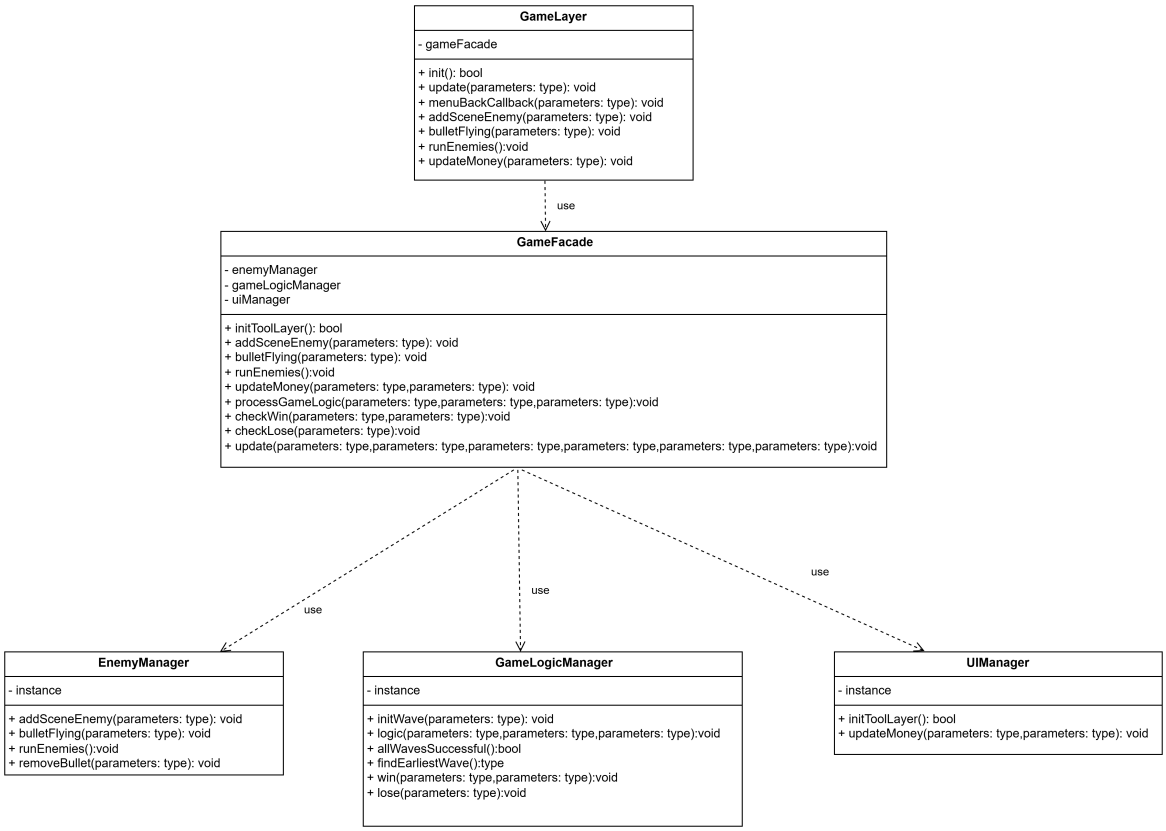
If the appearance becomes too bulky, you can consider extracting some of its behaviors as a new specialized appearance class.

In the current design, `GameFacade` has encapsulated all subsystem interactions in a unified interface. If the functionality of `GameFacade` is found to be too complex in the future, it may be considered to extract certain features into a new dedicated appearance class to further simplify the responsibilities of `GameFacade`.

3.1.4 UML Class Diagram

- `GameLayer`: As the main control layer of the game, responsible for game initialization, updates, and user interaction. It interacts with other subsystems through `GameFacade`.
- `GameFacade`: As a appearance class, it encapsulates calls to `EnemyManager`, `GameLogicManager`, and `UIManager`, providing a unified interface for `GameLayer` to use.
- `EnemyManager`: Responsible for managing enemies in the game, including their generation, behavior, and status updates.
- `GameLogicManager`: Responsible for handling the logic of the game, including wave management, outcome judgment, etc.

- `UIManager`: Responsible for managing the user interface of the game, including money display, buttons, etc.



3.1.5 Benefits of Refactoring

- **Solved Problem**
  - **Complexity management**
    Before refactoring, the `GameLayer` code was complex and difficult to maintain. By introducing multiple subsystems such as `EnemyManager`, `GameLogicManager`, and `UIManager`, as well as `GameFacade`, we have encapsulated these complex interactions into a unified interface, simplifying the logic of `GameLayer`.
  - **Separation of Responsibilities**
    The original `GameLayer` class took on too many responsibilities, including enemy management, game logic processing, and UI updates. After refactoring, `GameLayer` is only responsible for calling the `GameFacade` interface, while the specific logic processing and coordination of interactions between various subsystems are entrusted to `GameFacade` and various management classes, in accordance with the principle of single responsibility.
  - **Reduce coupling degree**
    By using the Facade mode, `GameLayer` no longer relies directly on the implementation details of multiple subsystems, but interacts through `GameFacade`. This reduces the coupling between classes, making the system easier to expand and modify.
  - **Improve readability**
    The restructured code structure is clearer and the logic of `GameLayer` becomes more intuitive. Developers can more easily understand the functionality of `GameLayer` without the need to delve into the implementation details of each subsystem.
- **Benefits Refactor Brings**

- **Enhance maintainability**
  By encapsulating complex logic in `GameFacade`, subsequent modifications to subsystems will not affect the implementation of `GameLayer`. Simply make the necessary adjustments in `GameFacade` to maintain system stability.
- **Simplify the interface**
  `GameFacade` provides a simple interface for `GameLayer` to use, reducing the complexity of method calls. Developers only need to focus on the interface of `GameFacade`, without having to deal with the details of multiple subsystems.
- **Improve scalability**
  If new features or subsystems need to be added in the future, simply add the corresponding methods in `GameFacade` without modifying the implementation of `GameLayer`. This makes the system more scalable and able to adapt to future demand changes.

  For example, to add a new bullet type only need to create a new decorator, you can:

  ```
  // add new function
  class GameFacade {
  public:
      // Add a method to call a new management class
  private:
      EnemyManager* enemyManager;
      GameLogicManager* gameLogicManager;
      UIManager* uiManager;
      // Add a new management class
  };
  ```

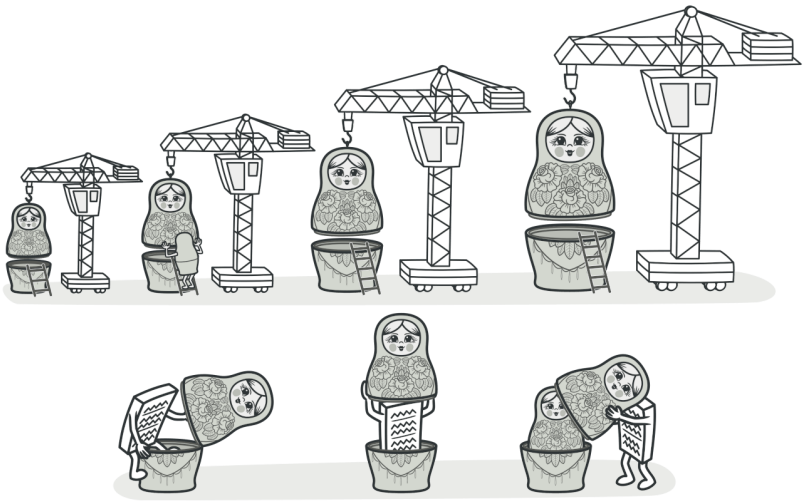
- **Principle of Single Responsibility**

  After refactoring, `GameLayer` is only responsible for calling the `GameFacade` interface, while the specific logic processing and coordination of interactions between various subsystems are entrusted to `GameFacade` and various management classes, in accordance with the principle of single responsibility.

3.2 Decorator Design Pattern

3.2.1 Brief of Decorator Pattern

Decorator Pattern is a structural design pattern that allows new functionality to be added dynamically without changing the structure of the object. The decorator pattern wraps the original object by creating a set of decorator classes, each of which adds its own functionality, capable of combining multiple behaviors without modifying the original object's code.



The decorative pattern first requires creating component interfaces and declaring common methods, and then implementing concrete component classes to define basic behavior. Next, you create a base decorator class that holds the wrapper object and delegates its work. Extend functionality by inheriting the base decorator and creating a concrete decorator. Finally, the client is responsible for assembling the decorator, combining functions on demand.

Decorative mode has the advantages of enhancing the functionality without changing the original class, dynamically adding behavior and multiple decorations, etc. Next, we will introduce the details of refactoring using decorative mode in our project.

3.2.2 Reason for Refactoring

In our game project, GanYuan can fire attack bullets at the enemy, and the enemy can also fire attack bullets at GanYuan. But let's think about a problem, **a bullet in a game can't have exactly the same damage effect**, such as "plants vs. zombies", bullets have damage, slow down, freeze and so on. At the same time, there are also "wet nurses" among our GanYuans, and the therapeutic bullets fired by them should also have therapeutic effects on friends. We may need to **expand on different bullet effects** in the future. So, now take a look at our bullet class first:

```
// Before refactory
// bullet class
class Bullet : public Sprite
{
    // bullet damage
    CC_SYNTHESIZE(float, damage, Damage);
    // bullet velocity
    CC_SYNTHESIZE(float, velo, Velo);
    // bullet from
    CC_SYNTHESIZE(Actor *, from, From);
    // bullet target
    CC_SYNTHESIZE(Actor *, target, Target);

public:
    // create bullet
    virtual bool init(const std::string &filename, float damage, float velo,
Actor *from, Actor *target);

    static Bullet *create(const std::string &filename, float damage, float velo,
Actor *from, Actor *target);

    // calculate bullet position
```

```
    void calculatePosition();

    // calculate bullet distance
    float calculateDistance() const;
};
```

We can see that the current bullet class has only one damage attribute, which **cannot represent different bullet effects** (healing, deceleration, etc.), and **lacks the scalability of bullet effects**. Also, if we were to expand new bullet types, all bullet logic would be in one class, which would result in **a bloated class** and would appear to be **highly coupled between different bullet behaviors**, violating the **single responsibility principle**.

To solve this problem, we can introduce decorator mode, which implements different bullet effects by creating a base bullet class and multiple decorator classes, but also allows us to dynamically add new behavior to the bullet to extend new bullets.

3.2.3 Refactoring Details

- **Define Component Interface**

In decorative mode, defining a public interface is the first step. Through this interface, we specify the basic behavior that all bullet types must implement. The key to this step is to ensure that all bullet classes share the same interface, and to force subclasses to implement these methods using pure virtual functions. Through the pure virtual function, the subclass is forced to realize the concrete behavior, thus achieving the unity and extensibility of the code.

```
// Define component interface
class Bullet : public Sprite
{
public:
    virtual bool init(const std::string &filename, float damage, float velo,
Actor *from, Actor *target) = 0;
    virtual void calculatePosition() = 0;
    virtual float calculateDistance() const = 0;
    virtual void applyEffect() = 0;

    CC_SYNTHESIZE(float, damage, Damage);
    CC_SYNTHESIZE(float, velo, Velo);
    CC_SYNTHESIZE(Actor *, from, From);
    CC_SYNTHESIZE(Actor *, target, Target);
};
```

In this step, we define a common interface class called Bullet that forces all concrete bullet classes to implement functions such as initialization, position calculation, distance calculation, and effect application to ensure consistent bullet behavior.

- **Concrete Components**

In this step, we implement the BaseBullet class, which serves as the base bullet class and provides the basic bullet functionality. This class implements all the methods defined in the Bullet interface, including initialization, calculating position, calculating distance, and applying effects. The BaseBullet class provides the base functionality that can be extended for subsequent decorators.

```
// Concrete Component - Basebullet realization
class BaseBullet : public Bullet
{
public:
    static BaseBullet *create(const std::string &filename, float damage, float
velo, Actor *from, Actor *target);
    virtual bool init(const std::string &filename, float damage, float velo,
Actor *from, Actor *target) override;
    virtual void calculatePosition() override;
    virtual float calculateDistance() const override;
    virtual void applyEffect() override;
};
```

The implementation of the base functions defined in the BaseBullet class can be found in the bullet.cpp file. It covers the basic behavior of bullets before refactoring code, such as create, init, calculatePosition, calculateDistance, applyEffect, and so on. A Concrete Component is a base type of the object being decorated. It implements a common interface that defines basic behaviors and functionality that we can then extend or modify through decorators.

```
// BaseBullet Implementation
// basic bullet creation
BaseBullet *BaseBullet::create(const std::string &filename, float damage, float
velo, Actor *from, Actor *target)
{
    BaseBullet *bullet = new (std::nothrow) BaseBullet;
    if (bullet && bullet->init(filename, damage, velo, from, target)){
        bullet->autorelease();
        return bullet;
    }
    CC_SAFE_DELETE(bullet);
    return nullptr;
}
// basic bullet initialization
bool BaseBullet::init(const std::string &filename, float damage, float velo,
Actor *from, Actor *target)
{
    if (!Sprite::init()){
        return false;
    }
    setTexture(filename);
    setPosition(from->getPosition());
    setScale(0.5);

    this->setDamage(damage);
    this->setVelo(velo);
    this->setFrom(from);
    this->setTarget(target);

    return true;
}
// basic bullet position calculation
void BaseBullet::calculatePosition()
{
    if (target != NULL){
        auto delta = target->getPosition() - this->getPosition();
        auto distance = delta.length();
```

```
        auto dx = delta.x;
        auto dy = delta.y;

        auto rotateRadians = delta.getAngle();
        auto rotateDegrees = CC_RADIANS_TO_DEGREES(-1 * rotateRadians);

        setRotation(rotateDegrees);
        setPosition(getPosition() + Vec2(dx / distance * velo /
FRAMES_PER_SECOND,
                                         dy / distance * velo /
FRAMES_PER_SECOND));
    }
}
// basic bullet distance calculation
float BaseBullet::calculateDistance() const
{
    if (target != NULL){
        auto delta = target->getPosition() - this->getPosition();
        return delta.getLength();
    }
    return 0;
}
// basic bullet effect application
void BaseBullet::applyEffect()
{
    if (target){
        target->takeDamage(damage);
    }
}
```

With the BaseBullet class, we provide a bullet type that can be used as a base for the decorator. Later we can add decorators to BaseBullet to add more functionality to the bullet without modifying the BaseBullet core code.

- **Create the Base Decorator**

Next, we need to create the base decorator, which is responsible for maintaining references to the decorated object and delegating all operations to this decorated object. The base decorator itself usually does not implement a specific function, but rather passes the function to the decorated object by way of delegation, thus achieving the purpose of extending and modifying the behavior.

Each method in the base decorator is delegated to the decorated object to perform the actual operation. Specifically, for each method in the BulletDecorator class, the decorator calls the corresponding implementation of the wrapped object, so you can implement a specific wrapper for each method.

```
// Base Decorator
class BulletDecorator : public Bullet
{
protected:
    Bullet *m_bullet;

public:
    BulletDecorator(Bullet *bullet) : m_bullet(bullet) {}
    // Decorator initialization delegate
```

```
    virtual bool init(const std::string &filename, float damage, float velo,
Actor *from, Actor *target) override{
        return m_bullet->init(filename, damage, velo, from, target);
    }
    // Decorator position delegate
    virtual void calculatePosition() override{
        m_bullet->calculatePosition();
    }
    // Decorator distance delegate
    virtual float calculateDistance() const override{
        return m_bullet->calculateDistance();
    }
    // Decorator effect delegate
    virtual void applyEffect() override{
        m_bullet->applyEffect();
    }
};
```

- **Implement Concrete Decorators**

In Step 4, we add a specific behavior or additional effect to the bullet by creating a concrete decorator class. Each concrete decorator expands on the base bullet functionality with additional features.

For example, in our project, we expand five kinds of bullets, including normal bullets, healing bullets, penetrating bullets, decelerating bullets and close-in bullets. The performance of different bullets is mainly due to the difference of applyAffect() function, that is, the attack effect of different bullets is different, and we use different bullet decorators to modify it. Next is an introduction to the different bullet decorators.

1. Normal bullets: Same as base bullets, dealing a certain amount of damage.

```
// Bullet.h
// Normal bullet decorator
class NormalBullet : public BulletDecorator
{
public:
    static NormalBullet *create(Bullet *bullet);
    virtual void applyEffect() override;
};
```

```
// Bullet.cpp
// Normal bullet decorator creation
NormalBullet *NormalBullet::create(Bullet *bullet)
{
    auto normalBullet = new (std::nothrow) NormalBullet(bullet);
    if (normalBullet)
    {
        normalBullet->autorelease();
        return normalBullet;
    }
    CC_SAFE_DELETE(normalBullet);
    return nullptr;
}

// Normal bullet decorator effect application
```

```
void NormalBullet::applyEffect(){
    BulletDecorator::applyEffect();
}
```

2. Healing Bullets: A medic fires at friendly forces, restoring health, i.e. healing bullets deal negative damage.

```
// Bullet.h
// Healing bullet decorator
class HealingBullet : public BulletDecorator
{
public:
    static HealingBullet *create(Bullet *bullet);
    virtual void applyEffect() override;
};
```

```
// Bullet.cpp
// Healing bullet decorator creation
HealingBullet *HealingBullet::create(Bullet *bullet)
{
    auto healingBullet = new (std::nothrow) HealingBullet(bullet);
    if (healingBullet){
        healingBullet->autorelease();
        return healingBullet;
    }
    CC_SAFE_DELETE(healingBullet);
    return nullptr;
}

// Healing bullet decorator effect application
void HealingBullet::applyEffect()
{
    if (target){
        target->takeDamage(-damage);
    }
}
```

3. Penetrate bullets: Ignore the enemy's defense when hitting them, and restore the enemy's defense after dealing damage.

```
// Bullet.h
// Penetrate bullet decorator
class PenetrateBullet : public BulletDecorator
{
public:
    static PenetrateBullet *create(Bullet *bullet);
    virtual void applyEffect() override;
};
```

```
// Bullet.cpp
// Penetrate bullet decorator creation
PenetrateBullet *PenetrateBullet::create(Bullet *bullet)
{
    auto penetrateBullet = new (std::nothrow) PenetrateBullet(bullet);
    if (penetrateBullet)
```

```
    {
        penetrateBullet->autorelease();
        return penetrateBullet;
    }
    CC_SAFE_DELETE(penetrateBullet);
    return nullptr;
}
// Penetrate bullet decorator effect application
void PenetrateBullet::applyEffect()
{
    if (target){
        float originalDefense = target->getDefence();
        target->setDefence(0);
        target->takeDamage(damage);
        target->setDefence(originalDefense);
    }
}
```

4. Slow down bullet: Deals a small amount of damage while slowing the movement of the attacked enemy for 3 seconds.

```
// Bullet.h
// Penetrate bullet decorator
class PenetrateBullet : public BulletDecorator
{
public:
    static PenetrateBullet *create(Bullet *bullet);
    virtual void applyEffect() override;
};
```

```
// Bullet.cpp
// Slow bullet decorator creation
SlowBullet *SlowBullet::create(Bullet *bullet)
{
    auto slowBullet = new (std::nothrow) SlowBullet(bullet);
    if (slowBullet){
        slowBullet->autorelease();
        return slowBullet;
    }
    CC_SAFE_DELETE(slowBullet);
    return nullptr;
}
// Slow bullet decorator effect application
void SlowBullet::applyEffect()
{
    if (target){
        // Less damage than a regular bullet
        target->takeDamage(damage * 0.5f);
        if (auto enemy = dynamic_cast<EnemyBase *>(target)){
            // Slow down the enemy
            enemy->setRunSpeed(enemy->getRunSpeed() * 0.7f);
            // Reset the enemy's speed after 3 seconds
            enemy->scheduleOnce([enemy](float)
                        { enemy->setRunSpeed(enemy->getOriginalRunSpeed()); },
3.0f, "reset_speed_key");
        }
```

```
    }
}
```

5. Melee bullet: Fired by shield soldiers, it deals more damage to the enemy due to its close range.

```
// Bullet.h
// Melee bullet decorator
class MeleeBullet : public BulletDecorator
{
public:
    static MeleeBullet *create(Bullet *bullet);
    virtual void applyEffect() override;
};
```

```
// Bullet.cpp
// Melee bullet decorator creation
MeleeBullet *MeleeBullet::create(Bullet *bullet)
{
    auto meleeBullet = new (std::nothrow) MeleeBullet(bullet);
    if (meleeBullet){
        meleeBullet->autorelease();
        return meleeBullet;
    }
    CC_SAFE_DELETE(meleeBullet);
    return nullptr;
}

// Melee bullet decorator effect application
void MeleeBullet::applyEffect()
{
    if (target){
        target->takeDamage(damage * 1.5f);
    }
}
```

- **Client Usage**

The next step is to use these bullets on the client side. In our game, normal bullets, penetrating bullets and decelerating bullets are used to attack ENEMY1, ENEMY2 and ENEMY3 from SHOOTER_TYPE. Healing bullets are fired by MEDICAL_TYPE to heal SHIELD_TYPE, MEDICAL_TYPE and SHOOTER_TYPE. MSM bullets are fired by SHIELD_TYPE to attack ENEMY2 and ENEMY3. ENEMY1 fires only normal bullets to attack SHOOTER_TYPE and MEDICAL_TYPE. ENEMY2 and ENEMY3 fire only normal bullets to attack SHOOTER_TYPE, MEDICAL_TYPE and SHIELD_TYPE.

We use these bullets judiciously in the attack function of EnemyBase.cpp and GanYuanBase.cpp (classes that are subclasses of actors), respectively.

```
// EnemyBase.cpp
bool EnemyBase::attack(Actor *target)
{
    if (!Actor::attack(target)){
        return false;
    }
```

```
    GameManager *instance = GameManager::getInstance();
    std::string tmpPath;
    // Choose bullet path based on enemy type
    switch (this->getType()){
    case ENEMY1_TYPE:
        tmpPath = ENEMY1_PATH;
        break;
    case ENEMY2_TYPE:
        tmpPath = ENEMY2_PATH;
        break;
    case ENEMY3_TYPE:
        tmpPath = ENEMY3_PATH;
        break;
    default:
        return false;
    }

    // Create base bullet
    auto baseBullet = BaseBullet::create(tmpPath, this->getLethality(), 3000,
this, target);
    if (!baseBullet)
        return false;

    Bullet *bullet = baseBullet;

    // ENEMY1 can only attack SHOOTER and MEDICAL
    if (this->getType() == ENEMY1_TYPE){
        if (target->getType() == SHOOTER_TYPE ||
            target->getType() == MEDICAL_TYPE){
            bullet = NormalBullet::create(baseBullet);
        }
    }
    // ENEMY2 and ENEMY3 can attack SHOOTER, MEDICAL, and SHIELD
    else if (this->getType() == ENEMY2_TYPE ||
             this->getType() == ENEMY3_TYPE){
        if (target->getType() == SHOOTER_TYPE ||
            target->getType() == MEDICAL_TYPE ||
            target->getType() == SHIELD_TYPE){
            bullet = NormalBullet::create(baseBullet);
        }
    }

    if (bullet == baseBullet){
        CC_SAFE_DELETE(baseBullet);
        return false;
    }
    // Set bullet position and scale
    bullet->setPosition(this->getPosition());
    bullet->setScale(0.12);
    instance->gameScene->addChild(bullet);
    instance->bulletVector.pushBack(dynamic_cast<Sprite *>(bullet));

    return true;
}
```

```
// GanYuanBase.cpp
bool GanYuanBase::attack(Actor *target)
```

```
{
    if (!Actor::attack(target)){
        return false;
    }

    GameManager *instance = GameManager::getInstance();
    std::string tmpPath;
    // Choose bullet path based on ganyuan type
    switch (this->getType()){
    case SHIELD_TYPE:
        tmpPath = SHIELD_PATH;
        break;
    case SHOOTER_TYPE:
        tmpPath = SHOOTER_PATH;
        break;
    case MEDICAL_TYPE:
        tmpPath = MEDICAL_PATH;
        break;
    default:
        return false;
    }

    // Create base bullet
    auto baseBullet = BaseBullet::create(tmpPath, this->getLethality(), 3000,
this, target);
    if (!baseBullet)
        return false;

    Bullet *bullet = baseBullet;
    // Create different bullets based on ganyuan type
    switch (this->getType()){
    // SHOOTER can attack ENEMY1, ENEMY2, and ENEMY3
    case SHOOTER_TYPE:
        if (target->getType() == ENEMY1_TYPE ||
            target->getType() == ENEMY2_TYPE ||
            target->getType() == ENEMY3_TYPE){
            // Randomly choose bullet type
            int bulletType = rand() % 3;
            switch (bulletType){
            case 0:
                bullet = NormalBullet::create(baseBullet);
                break;
            case 1:
                bullet = PenetrateBullet::create(baseBullet);
                break;
            case 2:
                bullet = SlowBullet::create(baseBullet);
                break;
            }
        }
        break;
    // MEDICAL can heal SHIELD, MEDICAL, and SHOOTER
    case MEDICAL_TYPE:
        if (target->getType() == SHIELD_TYPE ||
            target->getType() == MEDICAL_TYPE ||
            target->getType() == SHOOTER_TYPE){
            bullet = HealingBullet::create(baseBullet);
        }
```

```
            break;
        // SHIELD can attack ENEMY2 and ENEMY3
        case SHIELD_TYPE:
            if (target->getType() == ENEMY2_TYPE ||
                target->getType() == ENEMY3_TYPE){
                bullet = MeleeBullet::create(baseBullet);
            }
            break;
        }

        if (bullet == baseBullet){
            CC_SAFE_DELETE(baseBullet);
            return false;
        }
        // Set bullet position and scale
        bullet->setPosition(this->getPosition());
        bullet->setScale(0.12);
        instance->gameScene->addChild(bullet);
        instance->bulletVector.pushBack(dynamic_cast<Sprite *>(bullet));

        return true;
    }
```

At this point, refactoring the bullet class using decorator mode is complete, We have shown above the implementation of normal bullet, healing bullet, PenetrateBullet, slow bullet and MeleeBullet five bullet types, The bullet effect can be combined flexibly, and it is easy to expand new bullet types later.

3.2.4 UML Class Diagram

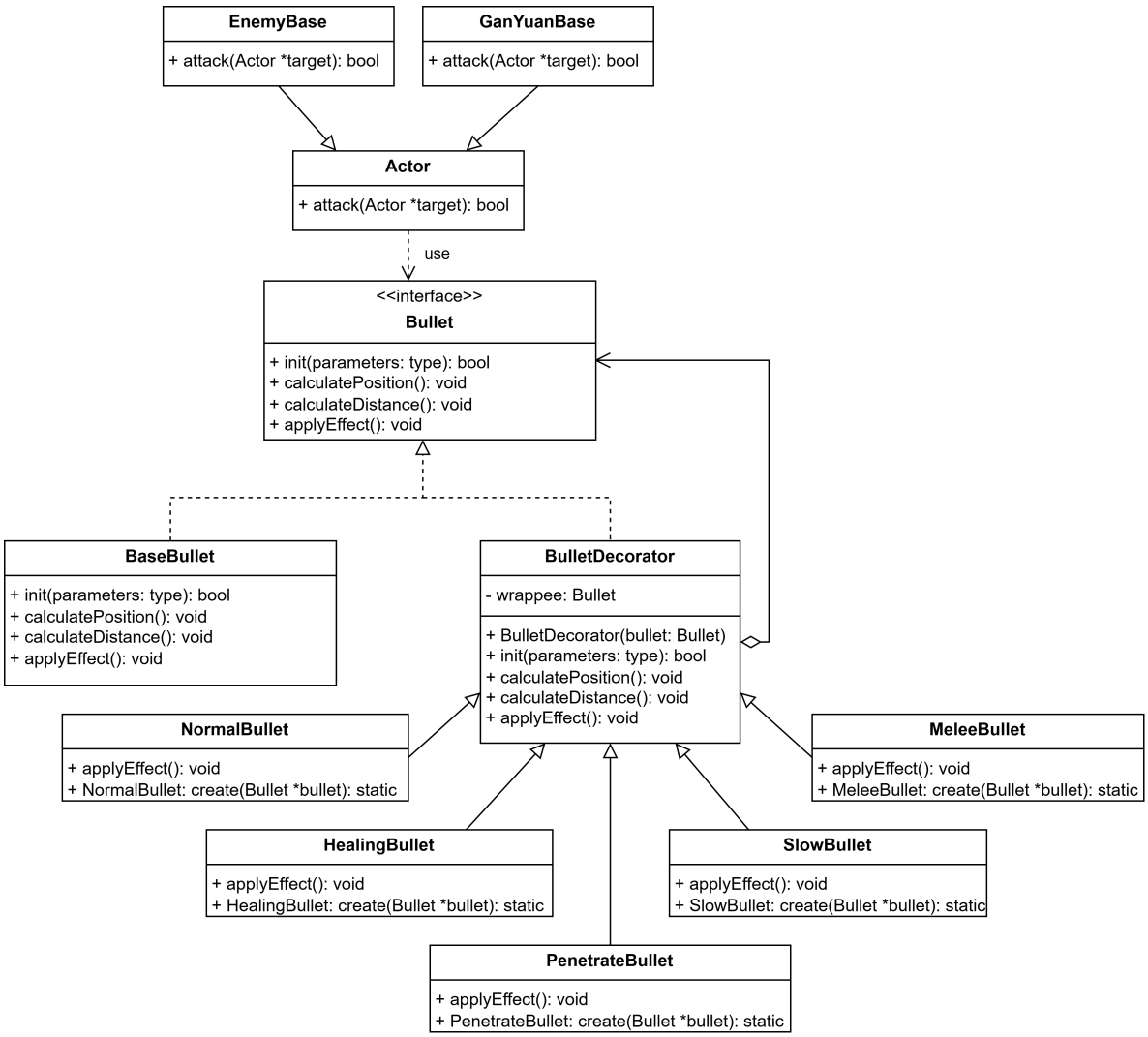
The Component (Bullet) interface defines the common methods that all bullet classes must implement. All Bullet classes (including the base bullet class and the decorator class) must implement the Bullet interface to ensure that they follow the same code of behavior.

The Concrete Component (BaseBullet) class implements the Bullet interface and provides the basic functionality of bullets. It is the basic component in the decorator pattern that provides a basic bullet to which the decorator can attach behavior, and these methods provide the default bullet behavior.BaseBullet is the basis for building a chain of decorators that will add or modify behavior based on this class. BaseBullet implements the basic applyEffect() method, but concrete decorators can extend this method with new functions such as healing, penetration, and deceleration.

The Base Decorator (BulletDecorator) is the base decorator class in the decorator pattern that inherits from the Bullet interface and holds a wrappee member that points to the decorated Bullet object. The base decorator implements the extension of bullet behavior by delegating all method calls to wrappee.

Concrete decorator classes (such as NormalBullet, HealingBullet, PenetrateBullet, SlowBullet, etc.) extend the BulletDecorator class. Each decorator adds specific behavior to the applyEffect() method. The decorator can maintain the basic behavior by calling super.applyEffect() before adding additional features to the bullet.

The Actor class is the client, which represents the user of the bullet. The Actor class can interact with any Bullet that implements the Bullet interface. The client achieves flexible bullet behavior by combining different decorators.



3.2.5 Benefits of Refactoring

- **Solved Problem**
  - **The Potential Class Explosion Problem**

    Before refactoring, if we need to add a new bullet effect, we need to create a new bullet class, resulting in a sharp increase in the number of classes, increasing the complexity and maintenance difficulty of the system. After refactoring, through the decorator mode, different bullet effects can be combined, avoiding the creation of a large number of subclasses, thus solving the potential class explosion problem.
  - **The Potential Code Duplication**

    Before refactoring, if there are many different bullet classes, there will be a lot of duplicate code, especially some basic behaviors need to be implemented in multiple classes. After refactoring, the common behavior is extracted into BaseBullet and BulletDecorator, reducing the number of duplicate code and improving the reusability of code.
  - **Poor Scalability**

    Before refactoring, adding new effects requires modifying existing code, increasing the risk of modifying existing classes.
    After refactoring, adding new effects only needs to create new decorator classes, without modifying existing code, thus improving the extensibility of the system. The convenient extensibility also makes the game content more colorful.
- **Benefits Refactor Brings**

- **Increased Flexibility**

  The decorator mode provides the ability to dynamically add different effects to bullets. Different decorators can be combined flexibly, making the combination of bullet effects more flexible and adaptable to more needs.

  For example, if I want to achieve a combination effect, I can use the following code:

```
// SHOOTER_TYPE realizes combination effect
case SHOOTER_TYPE:
    bullet = new SlowBullet(
                new PenetrateBullet(
                    new NormalBullet(baseBullet)
                )
            );
```

- **Increased Maintainability**

  Each decorator class has a single responsibility, focusing on achieving a bullet effect that makes decorator code easy to understand and maintain. At the same time, the decorators are independent of each other, and the modification of one decorator does not affect the other decorators.

- **Open and Close Principle**

  With the decorator mode, new bullet effects can be implemented by adding new decorator classes without modifying existing code. This is in line with the open-closed principle, that is, open to expansion and closed to modification.

  For example, to add a new bullet type only need to create a new decorator, you can:

```
// add new bullet decorator "FireBullet"
class FireBullet : public BulletDecorator {
    void applyEffect() override {
        // realize FireBullet effects
    }
};
```

- **Principle of Single Responsibility**

  Each decorator is responsible for only one bullet effect, which makes the functionality of each decorator highly centralized, which is easy for developers to test and modify.

4. Behavioral Design Pattern

4.1 Strategy Pattern

4.1.1 Brief of Strategy Pattern

The **Strategy Pattern** is a behavioral design pattern that enables selecting an algorithm's behavior at runtime. By defining a family of algorithms, encapsulating each one, and making them interchangeable, the Strategy Pattern allows the algorithm to vary independently from the clients that use it.

The Strategy Pattern begins by creating a **strategy interface** that declares the methods each concrete strategy must implement. Next, **concrete strategy classes** are developed to provide specific implementations of these algorithms. A **context class** is then created to maintain a reference to a strategy object and delegate the execution of the algorithm to the chosen strategy. The **client** is responsible for selecting and setting the appropriate strategy at runtime, enabling dynamic behavior changes without altering the context class's code.

The Strategy Pattern offers several advantages:

- **Flexibility and Reusability**: Algorithms can be easily swapped and reused across different contexts without modifying the client code.
- **Adherence to the Open/Closed Principle**: New strategies can be introduced without changing existing code, promoting system scalability and maintainability.
- **Reduced Complexity**: By encapsulating algorithms within separate classes, the pattern minimizes conditional statements and simplifies the codebase.
- **Enhanced Testability**: Individual strategies can be tested in isolation, ensuring each algorithm functions correctly independently of others.

By implementing the Strategy Pattern, developers can create more maintainable, flexible, and scalable systems. The following sections will delve into the specifics of refactoring our project using the Strategy Pattern.

4.1.2 Reason for Refactoring

In our game project, a ***GanYuan*** represents an operator deployed in the battle that can be located in certain areas. There are many kinds of ***GanYuan***, including ***GanYuanMedical***, ***GanYuanShield*** and ***GanYuanShooter*** for now in our game, all inheriting from there common parent class, ***GanYuanBase***.

Clearly, a ***GanYuan*** can not be deployed at any random location. Originally, We checked the legality of the position (and also formatting the position) by implementing and utilizing `positionLegal` method defined in each of the ***GanYuan*** subclasses. The reason why we don't reuse one single implementation defined in the ***GanYuanBase*** is that for different types of ***GanYuan***, the definition of 'legal' for a certain position may be different and require special treatment. Let's take a look at how different subclass implemented their own version of `positionLegal` method originally.

```
// GanYuanMedical.cpp
void GanYuanMedical::positionLegal(bool& state, Vec2& p) {
    GameManager* instance = GameManager::getInstance();
    for (int i = 0; i < instance->towersPosition.size(); i++) {
        //(road_path[i - 1] - road_path[i]).getLength()
```

```
        if ((this->getPosition()).distance(instance->towersPosition[i]) < 50.f)
        {
            state = true;
            p = instance->towersPosition[i];
            listener1->setEnabled(0);
            return;
        }
    }
    return;
}
```

```
// GanYuanMedical.cpp
void GanYuanMedical::positionLegal(bool& state, Vec2& p) {
    GameManager* instance = GameManager::getInstance();
    for (int i = 0; i < instance->towersPosition.size(); i++) {
        //(road_path[i - 1] - road_path[i]).getLength()
        if ((this->getPosition()).distance(instance->towersPosition[i]) < 50.f)
        {
            state = true;
            p = instance->towersPosition[i];
            listener1->setEnabled(0);
            return;
        }
    }
    return;
}
```

```
// GanYuanShield.cpp
void GanYuanShield:: positionLegal(bool& state, Vec2& p) {
    GameManager* instance = GameManager::getInstance();
    for (int i = 0; i < instance->groundsPosition.size(); i++) {
        //(road_path[i - 1] - road_path[i]).getLength()
        if ((this->getPosition()).distance(instance->groundsPosition[i]) < 50.f)
        {
            state = true;
            p = instance->groundsPosition[i];
            listener1->setEnabled(0);
            return;
        }
    }
    return;
}
```

```
// GanYuanShooter.cpp
void GanYuanShooter::positionLegal(bool& state, Vec2& p) {
    GameManager* instance = GameManager::getInstance();
    for (int i = 0; i < instance->towersPosition.size(); i++) {
        //(road_path[i - 1] - road_path[i]).getLength()
        if ((this->getPosition()).distance(instance->towersPosition[i]) < 50.f)
        {
            state = true;
            p = instance->towersPosition[i];
            listener1->setEnabled(0);
            return;
        }
```

```
    }
    return;
}
```

As you can clearly see, the logic for the legality validation for different ***GanYuan*** types has plenty of similarities. In fact, you may have already realized that they only differ in the valid position they choose from, for example, like this:

```
instance->towersPosition
```

To reuse the rest of the logic that stays the same, we decide to use **Strategy Pattern** to isolate the logic of choosing the specific set of valid positions and extract the rest of the logic common for every type into a common superclass.

4.1.3 Refactoring Details

- **Identify the strategy and declare the strategy interface**

We start by identifying and isolating the only different part in the original implementation, which is the **legal position set selection logic**. We isolate this part into a nested class of ***GanyuanBase***, like this:

```
// GanYuanBase.h
class GanYuanBase : public Actor
{
    //...
    public: class GetPosition {
        public: virtual std::vector<Vec2> &get(GameManager *);
    };
    //...
}
```

- **Implement the strategy one by one**

We then implement the different strategies required. For now, we stay consistent with the original implementation functionalities by creating one selection logic for each type of ***GanYuan*** type. This can actually be different in the future, for example, when multiple logic should be used based on the extended game mechanism.

```
// GanYuanMedical.h
class GanYuanMedical :public GanYuanBase
{
    public: class GetPositionMedical: public GanYuanBase::GetPosition {
        public: virtual std::vector<Vec2> &get(GameManager * instance) {
            return instance->towersPosition[i];
        }
    };
    //...
}
```

```
// GanYuanShield.h
class GanYuanShield : public GanYuanBase
{
    public: class GetPositionShield: public GanYuanBase::GetPosition {
        public: virtual std::vector<Vec2> &get(GameManager * instance) {
            return instance->towersPosition[i];
        }
    };
    //...
}
```

```
// GanYuanShooter.h
class GanYuanShooter : public GanYuanBase
{
    public: class GetPositionShooter: public GanYuanBase::GetPosition {
        public: virtual std::vector<Vec2> &get(GameManager * instance) {
            return instance->towersPosition[i];
        }
    };
    //...
}
```

- **Add a reference to the strategy object and the corresponding initialization**

This is a simple step. We simply add a line like this:

```
// GanYuanBase.h
class GanYuanBase : public Actor {
    //...
    GetPosition * get_position;
    //...
}
```

And some initialization code for each subclass:

```
// GanYuanMedical.cpp
void GanYuanMedical::initial()
{
    setDefaultData();
    this->get_position = new GanYuanMedical::GetPosition(); // Newly added.
    firstInteract();
}
```

```
// GanYuanShield.cpp
void GanYuanShield::initial()
{
    setDefaultData();
    this->get_position = new GanYuanShield::GetPosition(); // Newly added.
    firstInteract();
}
```

```
// GanYuanShooter.cpp
void GanYuanShooter::initial()
{
    setDefaultData();
    this->get_position = new GanYuanShooter::GetPosition(); //Newly added.
    firstInteract();
}
```

- **Extract the common logic**

Finally, we implement the `positionLegal` in ***GanYuanBase*** instead of its subclass, and all jobs are done.

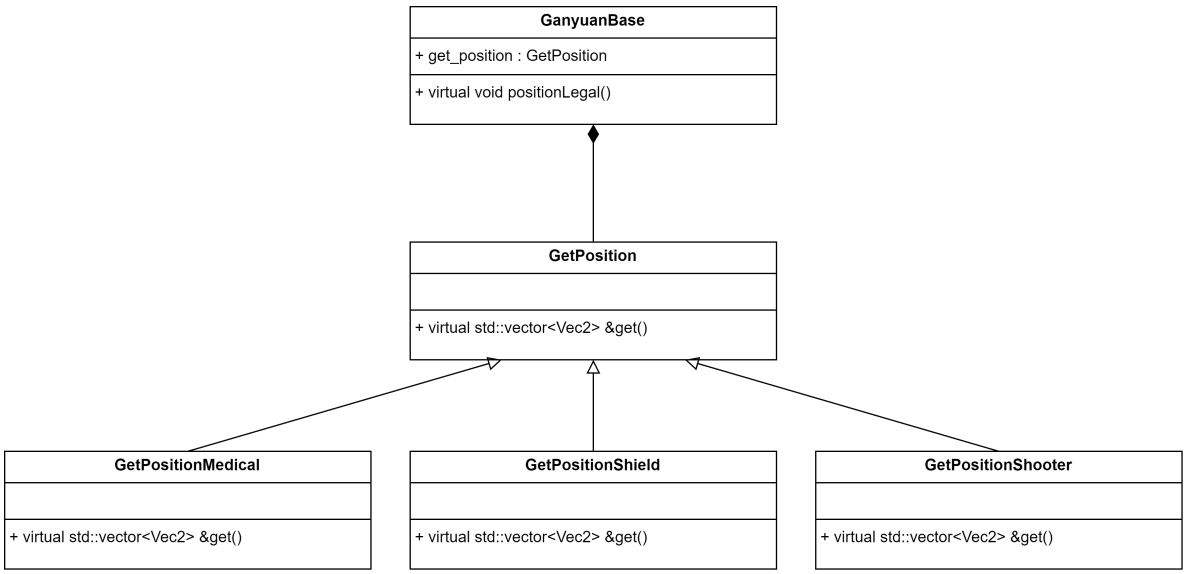
```
//GanYuanBase.cpp
void GanYuanBase::positionLegal(bool& state, Vec2& p) {
    GameManager* instance = GameManager::getInstance();
    auto pos = this->get_position->get(instance);
    for (int i = 0; i < pos.size(); i++) {
        //(road_path[i - 1] - road_path[i]).getLength()
        if ((this->getPosition()).distance(pos[i]) < 50.f)
        {
            state = true;
            p = pos[i];
            listener1->setEnabled(0);
            return;
        }
    }
    return;
}
```

4.1.4 UML Class Diagram

The **Strategy Interface (GetPosition)** defines the common method that all strategy classes must implement. All strategy classes (including the base strategy and its concrete implementations) must adhere to this interface to ensure they follow the same behavior contract. The interface includes the `get(GameManager* instance)` method, which returns the positions used for different purposes (e.g., tower placement, shooter positioning, etc.).

The **concrete strategy** (**GetPositionMedical**, **GetPositionShield**, **GetPositionShooter**) classes implement the GetPosition interface and define the logic for retrieving medically relevant positions. The strategy returns the position from the GameManager instance by overriding the get() method

The **Context Class (GanYuanBase)** is the user of the strategy pattern. It holds a reference to a `GetPosition` strategy instance and interacts with it to retrieve the required positions. The context class delegates the execution of the `get()` method to the currently assigned strategy object.



4.1.5 Benefits of Refactoring

- **Solved Problems**
  - **The Potential Code Duplication**

    Prior to refactoring, the `positionLegal` method was duplicated across multiple **GanYuan** subclasses, with only slight variations in the position selection criteria. This duplication not only violated the DRY (Don't Repeat Yourself) principle but also increased the risk of inconsistencies and bugs. By extracting the position selection logic into distinct strategy classes, the common code resides in the **GanYuanBase** class, significantly reducing code duplication and enhancing code reusability.
- **Benefits Refactor Brings**
  - **Increased Flexibility**

    The Strategy Pattern provides the ability to dynamically select and switch position selection strategies at runtime. This flexibility allows the system to adapt to different game scenarios and mechanics without requiring significant code changes. For instance, if a new game mode requires a different set of valid positions for deploying ***GanYuan***, a new strategy can be introduced and applied seamlessly.

    ```
    // Example of switching strategies at runtime
    GanYuanBase* ganYuan = new GanYuanMedical();
    ganYuan->setPositionStrategy(new NewPositionStrategy());
    ganYuan->positionLegal(state, p);
    ```

  - **Adherence to the Open/Closed Principle**

    The refactored system complies with the Open/Closed Principle by allowing the addition of new position selection strategies without modifying existing code. This adherence ensures that the system can grow and evolve without compromising the stability of existing functionalities.

    For example, to introduce a new strategy for ***GanYuan*** deployment:

```
// Add new position strategy "AdvancedPositionStrategy"
class AdvancedPositionStrategy : public GanYuanBase::GetPosition {
  public:
    virtual std::vector<Vec2> &get(GameManager* instance) override {
        // Implement advanced position selection logic
    }
};
```

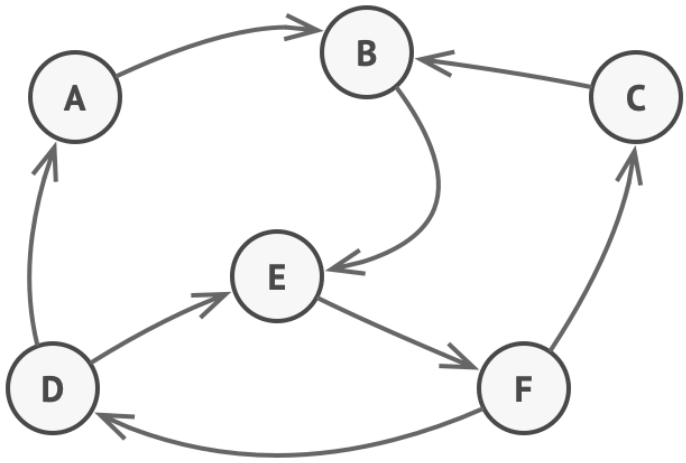
- **Principle of Single Responsibility**

  Each strategy class is responsible for a specific aspect of position selection, ensuring that classes remain focused and cohesive. This clear delineation of responsibilities simplifies testing and future modifications, as changes to one strategy do not impact others.

4.2 State Pattern

4.2.1 Brief of State Pattern

The **State Strategy**, often compared to a **Finite State Machine (FSM)**, is a behavioral design pattern that models an object's behavior based on its internal state. Like FSMs, it involves defined states and transitions, but instead of representing the logic in a table or diagram, it encapsulates state-specific behaviors into classes. This approach simplifies the management of state transitions and associated behaviors in code.



Finite-State Machine.

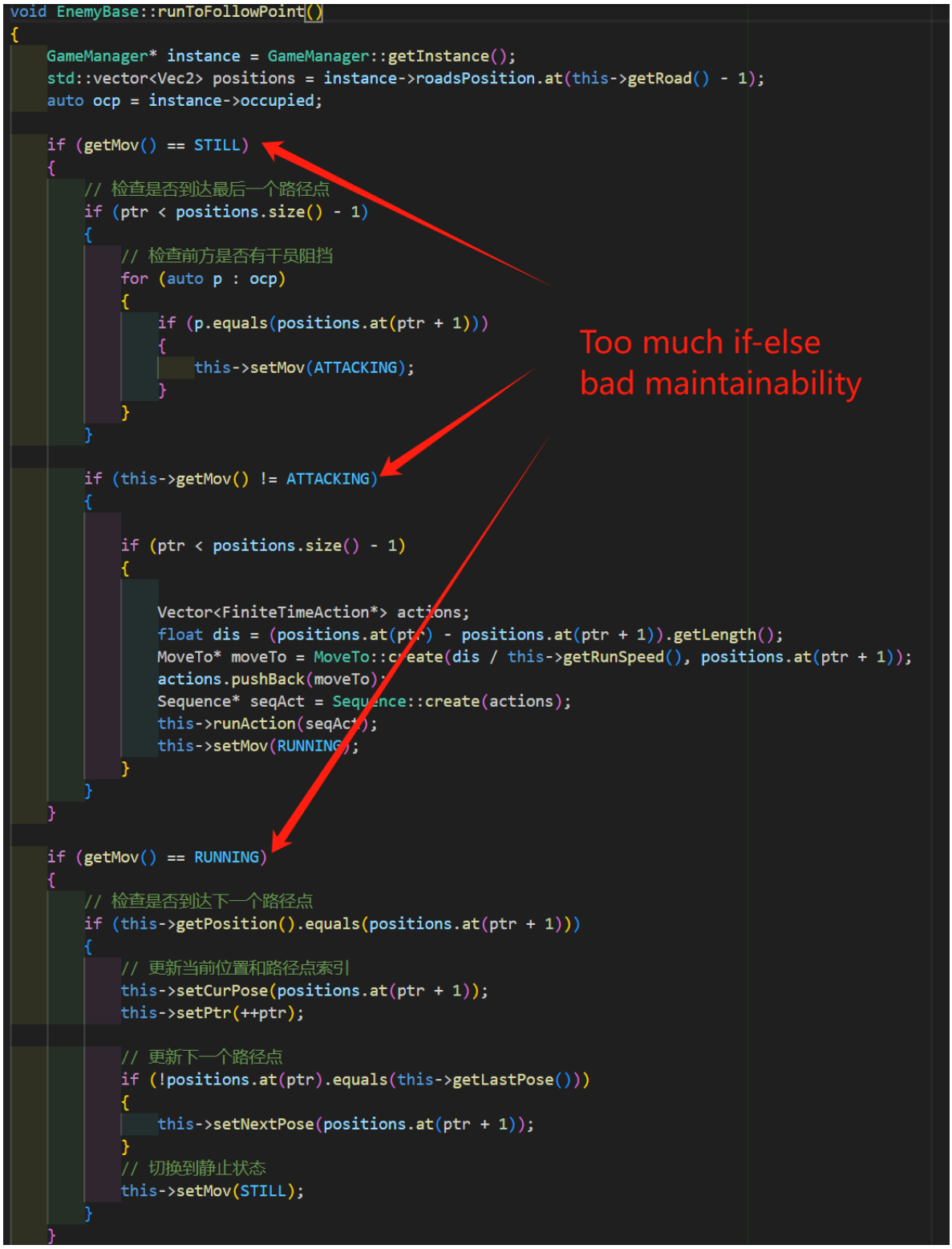
The **Context** is an object that holds the current state and dynamically delegates behavior to the state object. It manages state transitions and maintains a reference to the current state while providing a public interface for clients to interact with, delegating specific behaviors to the current state. The **State** serves as an abstract class or interface that defines the behaviors each state must implement. It establishes a uniform interface for all states, ensuring consistent behavior definitions and simplifying the logic within the context. Specific implementations of the state interface, referred to as **ConcreteState**, represent distinct states of the context. These concrete states encapsulate state-specific behavior and can notify the context to transition to a different state when necessary.

4.2.2 Reason for Refactoring

The `runToFollowPoint` function in EnemyBase class handles the behavior of an enemy object based on its state (`STILL`, `ATTACKING`, `RUNNING`). The current implementation has several critical issues that can lead to long-term maintenance and scalability problems.

- high Complexity and Poor Readability

The current function uses multiple nested `if` statements to handle state-specific behavior, making the logic hard to read and understand. Each state (`STILL`, `RUNNING`, `ATTACKING`) has its behavior mixed into a single method, which obscures the purpose and flow of the code. This makes debugging and understanding state behavior challenging.



- Strong Coupling Between States and Behavior

The logic for different states is tightly coupled within the function. This means any change to one state's behavior can inadvertently impact the others. For instance, modifying `ATTACKING` logic might require careful navigation through the entire function to ensure it doesn't conflict with `STILL` or `RUNNING`.

- Difficulty in Adding New States or Modifying Existing Ones

If we want to add a new state (e.g., `FLEEING`) or modify an existing state's behavior, we'll have to revisit the already complex `runToFollowPoint` function and add more conditions, further increasing complexity. This violates the **Open/Closed Principle** of software design, where code should be open to extension but closed to modification.

- Testing Challenges

With all state-related logic in one function, it becomes harder to test each state's behavior independently. A bug in one part of the function might cascade into other parts, making it harder to isolate and debug.

- **Code Bloat and Maintenance Issues**

As more states or behaviors are introduced, the `runToFollowPoint` function will grow larger and more bloated. This makes the function a bottleneck in the system—any future changes risk breaking existing functionality.

The **State Pattern** solves these issues by decoupling state logic from the `runToFollowPoint` function and encapsulating it in independent state classes. Each class handles the behavior specific to one state (e.g., `StillState`, `AttackingState`, `RunningState`), making the system more modular and easier to understand. The `EnemyBase` class (or equivalent context) will only manage state transitions, simplifying its responsibility.

4.2.3 Refactoring Details

1. **Context Class Identification**

   Firstly, in `Classes/EnemyBase.h`, the `EnemyBase` class is identified as the context class since it contains the state-dependent behavior in `runToFollowPoint`

2. **State Interface Declaration**

   Secondly, we create abstract class `EnemyState` with three core methods: `enter()`, `execute()`, and `exit()`. And then Add protected helper methods like `getRoadPositions()`, `getNextPathPoint()`, etc.

```
class EnemyBase;

class EnemyState {
protected:
    EnemyBase* enemy;

public:
    EnemyState(EnemyBase* enemy) : enemy(enemy) {}
    virtual ~EnemyState() {}

    // Core virtual methods - enemy parameters are no longer required
    virtual void enter() = 0;
    virtual void execute() = 0;
    virtual void exit() = 0;

protected:
```

```
    // Common path manipulation method - No longer requires the enemy
parameter
    std::vector<Vec2> getRoadPositions() {
        GameManager* instance = GameManager::getInstance();
        return instance->roadsPosition.at(enemy->getRoad() - 1);
    }

    Vec2 getNextPathPoint() {
        auto positions = getRoadPositions();
        return positions.at(enemy->getPtr() + 1);
    }

    bool hasNextPoint() {
        auto positions = getRoadPositions();
        return enemy->getPtr() < positions.size() - 1;
    }

    // General status check method
    bool isBlockedByGanYuan(const Vec2& position) {
        Actor* blocker = enemy->checkBlockedGanYuan(position);
        return blocker != nullptr && blocker->getAlive();
    }

    // Universal mobile computing methods
    float calculateMoveDuration(const Vec2& from, const Vec2& to) {
        float distance = enemy->getDistance(from, to);
        return distance / enemy->getRunSpeed();
    }

    // General state transition method
    void transitToStillState() {
        enemy->changeState(new StillState(enemy));
    }

    void transitToRunningState() {
        enemy->changeState(new RunningState(enemy));
    }

    void transitToAttackingState() {
        enemy->changeState(new AttackingState(enemy));
    }
};
```

3. **Concrete State Classes Creation**

Then, we create three concrete state classes: `StillState`, `RunningState`, and `AttackingState`. Each inherits from `EnemyState` and implements the three core methods

```
#include "EnemyState.h"

// StillState
class StillState : public EnemyState {
public:
    StillState(EnemyBase* enemy) : EnemyState(enemy) {}
    virtual void enter() override;
    virtual void execute() override;
    virtual void exit() override;
```

```
};

// RunningState
class RunningState : public EnemyState {
public:
    RunningState(EnemyBase* enemy) : EnemyState(enemy) {}
    virtual void enter() override;
    virtual void execute() override;
    virtual void exit() override;
};

// AttackingState
class AttackingState : public EnemyState {
public:
    AttackingState(EnemyBase* enemy) : EnemyState(enemy) {}
    virtual void enter() override;
    virtual void execute() override;
    virtual void exit() override;
};
```

4. **State-Specific Code Extraction**

   Later, we extract code from `runToFollowPoint()` into respective state classes:

- STILL state code → `StillState::execute()`
- RUNNING state code → `RunningState::execute()`
- ATTACKING state code → `AttackingState::execute()`

```
void StillState::execute(EnemyBase* enemy) {
    if (!hasNextPoint(enemy)) {
        return;
    }

    Vec2 nextPoint = getNextPathPoint(enemy);

    // Check to see if there is an agent blocking you
    if (isBlockedByGanYuan(enemy, nextPoint)) {
        transitToAttackingState(enemy);
        return;
    }

    // No obstruction. Start moving
    float duration = calculateMoveDuration(enemy, enemy-
>getCurrentPosition(), nextPoint);
    enemy->moveToPosition(nextPoint, duration);
    transitToRunningState(enemy);
}
void RunningState::execute(EnemyBase* enemy) {
    Vec2 nextPoint = getNextPathPoint(enemy);

    if (enemy->isAtPosition(nextPoint)) {
        enemy->setCurPose(nextPoint);
        enemy->setPtr(enemy->getPtr() + 1);

        auto positions = getRoadPositions(enemy);
        if (!positions.at(enemy->getPtr()).equals(enemy->getLastPose())) {
            enemy->setNextPose(positions.at(enemy->getPtr() + 1));
```

```
        }

        transitToStillState(enemy);
    }
}
void AttackingState::execute(EnemyBase* enemy) {
    if (enemy->getType() != ENEMY3_TYPE && enemy->getType() != ENEMY2_TYPE)
{
        return;
    }

    Vec2 nextPoint = getNextPathPoint(enemy);
    Actor* target = enemy->checkBlockedGanYuan(nextPoint);

    if (target && target->getAlive()) {
        enemy->setAttacking(target);
        enemy->attack(target);
    } else {
        transitToStillState(enemy);
    }
}
```

5. **State Reference Addition**

   Then, in `Classes/EnemyBase.h` we add `EnemyState* state` as a private member and add `changeState(EnemyState* newState)` method for state transitions.

```
class EnemyState;

class EnemyBase : public Actor
{
private:
    EnemyState* state;

public:

    EnemyBase();

    virtual bool init(const std::string& filename);

    static EnemyBase* create(const std::string& filename);

    // State-dependent method
    void changeState(EnemyState* newState);
    void runToFollowPoint();

    // Methods for use by state classes
    Vec2 getCurrentPosition() const { return getPosition(); }
    Vec2 getNextPosition(int index) const;
    float getDistance(const Vec2& pos1, const Vec2& pos2) const;
    void moveToPosition(const Vec2& target, float duration);
    bool isAtPosition(const Vec2& pos) const;

}
```

6. **Context Method Modification**

Original complex `runToFollowPoint()` with state conditionals replaced with:

```
void EnemyBase::runToFollowPoint() {
    if (state)
        {
            state->execute(this);
        }
}
```

7. **Initial State Setup**

   In `Classes/EnemyBase.cpp`, In constructor:

```
EnemyBase::EnemyBase() {
    state = new StillState(this);
}
```

8. **State Transitions Implementation**

   Last, in `Classes/EnemyBase.cpp`, we implement state change mechanism:

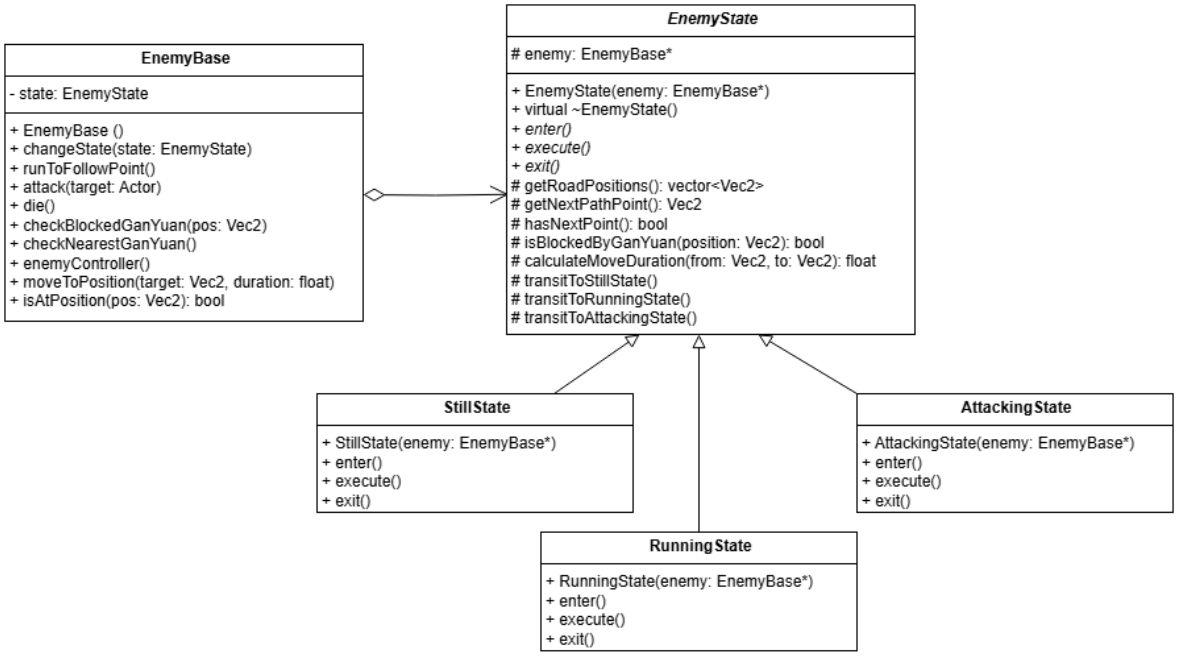
```
void EnemyBase::changeState(EnemyState* newState) {
    if (state) {
        state->exit();
        delete state;
    }
    state = newState;
    if (state) {
        state->enter();
    }
}
```

4.2.4 UML Class Diagram

This UML diagram illustrates a well-structured State Pattern implementation for enemy behavior in what appears to be a game system. The refactoring demonstrates several key design improvements: First, it extracts common state-related behaviors into an abstract EnemyState base class, which defines the interface for state transitions and core operations (enter(), execute(), exit()).

This abstraction reduces code duplication and establishes a consistent contract for all concrete states. Second, the concrete states (StillState, RunningState, and AttackingState) inherit from EnemyState, allowing for specialized implementations while maintaining a uniform interface.

The EnemyBase class acts as the context, maintaining a composition relationship with EnemyState, which enables dynamic state changes during runtime. This refactoring improves maintainability by encapsulating state-specific behavior within dedicated classes, making the system more modular and easier to extend. The use of virtual methods in EnemyState ensures proper polymorphic behavior, while the clear separation of concerns between the context (EnemyBase) and its states makes the code more organized and easier to understand.



4.2.5 Benefits of Refactoring

- Improved Code Maintainability

By separating state-specific logic into independent classes (e.g., `StillState`, `RunningState`, and `AttackingState`), the responsibilities of each state are clearly defined. The `EnemyBase` class no longer needs to handle all state logic in a single, monolithic function. This modular approach simplifies the maintenance process, as changes or fixes can be made to one state without worrying about affecting others. For example, modifying how `RUNNING` operates won't require touching `ATTACKING` or `STILL` logic buried in the same method.

- Enhanced Scalability for Adding New States

With the current implementation, adding a new state (e.g., `FLEEING` or `IDLE`) would require revisiting the `runToFollowPoint` method, introducing new conditional branches, and potentially disrupting existing functionality. After refactoring, new states can be implemented as independent classes, adhering to the **Open/Closed Principle**. The `EnemyBase` class simply needs to transition to the new state, making it easy to extend the system without risking unintended side effects.

- Reduced Code Complexity and Improved Readability

Currently, the `runToFollowPoint` function is filled with nested `if` statements that mix state checks and behaviors. This makes it difficult to follow the flow of logic, especially for someone unfamiliar with the code. Refactoring with the State Pattern ensures each state encapsulates its own behavior, allowing the `runToFollowPoint` function to delegate behavior to the current state. This makes the overall code structure cleaner and easier to understand.

- Easier Testing and Debugging

With state-specific behavior encapsulated in separate classes, testing can focus on individual state logic in isolation. For instance, you can test the `RunningState` class independently of `AttackingState`. This modular approach makes debugging more straightforward, as any issue is likely confined to a specific state rather than being buried in a shared function like `runToFollowPoint`. It also reduces the risk of introducing cascading errors when making changes to one part of the code.

- Clearer State Transitions and Responsibility Separation

Currently, the `runToFollowPoint` function handles not only the logic for each state but also the transitions between them (e.g., switching from `STILL` to `RUNNING`). This violates the **Single Responsibility Principle** by combining state behavior with state management. After refactoring, the `EnemyBase` class would only manage state transitions, while each state class handles its respective logic. This separation makes transitions more explicit and reduces the mental overhead of understanding how state changes are triggered.

5. Additional Design Pattern

5.1 Null Object Pattern

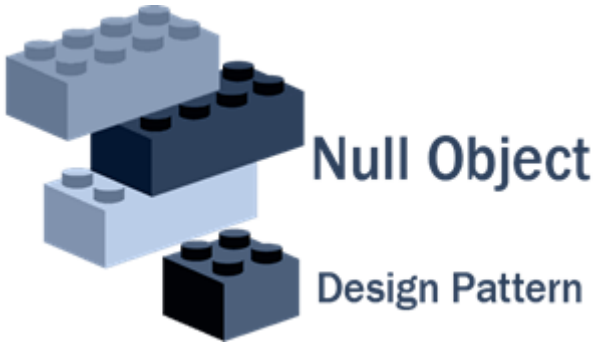
5.1.1 Brief of Null Object Pattern

The empty object pattern is a behavior design pattern used to avoid explicit checks for null values by providing an object that does "nothing". The core idea is to use a special object to represent "no action" or "default behavior", thus simplifying the code and avoiding null pointer exceptions.

- **Key concept**

1. Empty object: An object that implements a particular interface or inherits an abstract class, and whose methods do not perform any operations or return default values.
2. In place of null: Returns an empty object where null is required.
3. Behavior consistency: Empty objects and normal objects have the same interface, and the caller does not need to care which object is returned.

The advantage of the null object pattern is that it eliminates the explicit checking of null values by replacing null with an empty object of consistent behavior, thus simplifying the code logic and avoiding frequent if-else branch judgments. At the same time, it enhances the robustness of the code, reduces the risk of null pointer exceptions, and allows callers to focus on the core logic without having to worry about whether special handling of null values is required. This design also improves the readability and maintainability of the code, making the system more stable and consistent.



5.1.2 Reason for Refactoring

In the current game system, the bullet has a flight time from firing to hitting the target. During this process, the target may be destroyed and removed from the game due to the attack of other bullets. This leads to the following problems: null pointer risk, when the target disappears, the target pointer held by the bullet becomes a dangling pointer code requiring a lot of if (target! = nullptr) checks, which are scattered across methods and can easily be missed and crash. This leads to code maintainability issues, repeated null checks make the code verbose, and error handling logic is scattered and difficult to maintain uniformly, so we use the null object pattern here.

5.1.3 Refactoring Details

1. First we need to define the Actor interface, change the Actor class to a pure virtual class, define the public interface that all roles need to implement, and make sure that all methods are pure virtual functions that subclasses must implement.
2. Create a RealActor class: Move the implementation code from the original Actor class into the RealActor class.
   RealActor inherits from Actor and implements all virtual functions. Retain the original functionality and logic.
   Create the NullActor class: inherits from Actor. Implement all virtual functions, but provide empty operations or return default values. Use **singleton pattern** to ensure that there is only one NullActor instance on the system.

   (Here we also implement the singleton pattern in the creator pattern)

```
// Actor.cpp
// Null Object Pattern: Actor class
class Actor : public cocos2d::Sprite
{
public:
    virtual void die() = 0;
    virtual bool attack(Actor *target) = 0;
    virtual void takeDamage(INT32 damage) = 0;
    virtual bool init() = 0;
    virtual Vec2 getPosition() const = 0;
    virtual float getDefence() const = 0;
    virtual void setDefence(float defence) = 0;
    virtual ~Actor() {}
};

// Null Object Pattern: RealActor class
class RealActor : public Actor
{
protected:
    Bar *lethalityBar;
    Bar *healthBar;
    Bar *defenceBar;
    CC_SYNTHESIZE(int, type, Type);
    CC_SYNTHESIZE(int, road, Road);
    CC_SYNTHESIZE(int, scope, Scope);
    CC_SYNTHESIZE(int, lethality, Lethality);
    CC_SYNTHESIZE(float, hp, Hp);
    CC_SYNTHESIZE(float, health, Health);
    CC_SYNTHESIZE(float, defence, Defence);
    CC_SYNTHESIZE(bool, alive, Alive);
    CC_SYNTHESIZE(float, intervalTime, IntervalTime);
    CC_SYNTHESIZE(float, lastAttackTime, LastAttackTime);
    CC_SYNTHESIZE(bool, isBlock, IsBlock);
    CC_SYNTHESIZE(int, block, Block);
    CC_SYNTHESIZE(int, curBlock, CurBlock);
    CC_SYNTHESIZE(bool, isGround, IsGround);
    CC_SYNTHESIZE(Actor *, attacking, Attacking);

public:
    virtual void die() override;
    virtual bool attack(Actor *target) override;
    virtual void takeDamage(INT32 damage) override;
    virtual bool init() override;
```

```
    virtual Vec2 getPosition() const override;
    virtual void setPosition(const Vec2 &position) override;
    virtual float getDefence() const override;
    virtual void setDefence(float defence) override;
    static RealActor *create();
};

// Null Object Pattern: NullActor class
class NullActor : public Actor
{
public:
    static NullActor *getInstance();

    virtual void die() override {}
    virtual bool attack(Actor *target) override { return false; }
    virtual void takeDamage(INT32 damage) override {}
    virtual bool init() override { return true; }
    virtual Vec2 getPosition() const override { return Vec2::ZERO; }
    virtual void setPosition(const Vec2 &position) override {}
    virtual float getDefence() const override { return 0.0f; }
    virtual void setDefence(float defence) override {}

private:
    static NullActor *instance;
    NullActor() {} // private constructor
};
```

3. Because nullObject's function is relatively simple, the override function is written in the.h file. Here is the singleton schema for the empty object:

```
// Actor.h
// Null Object Pattern: NullActor class
NullActor *NullActor::getInstance()
{
    if (instance == nullptr)
    {
        instance = new NullActor();
    }
    return instance;
}
```

4. Modify Bullet.cpp to change all target! The judgment of = nullptr is replaced by using the null object mode.
   In the BaseBullet::init method, NullActor::getInstance() is used instead of nullptr. Remove null checks from other methods and call target's method directly.

   For example:

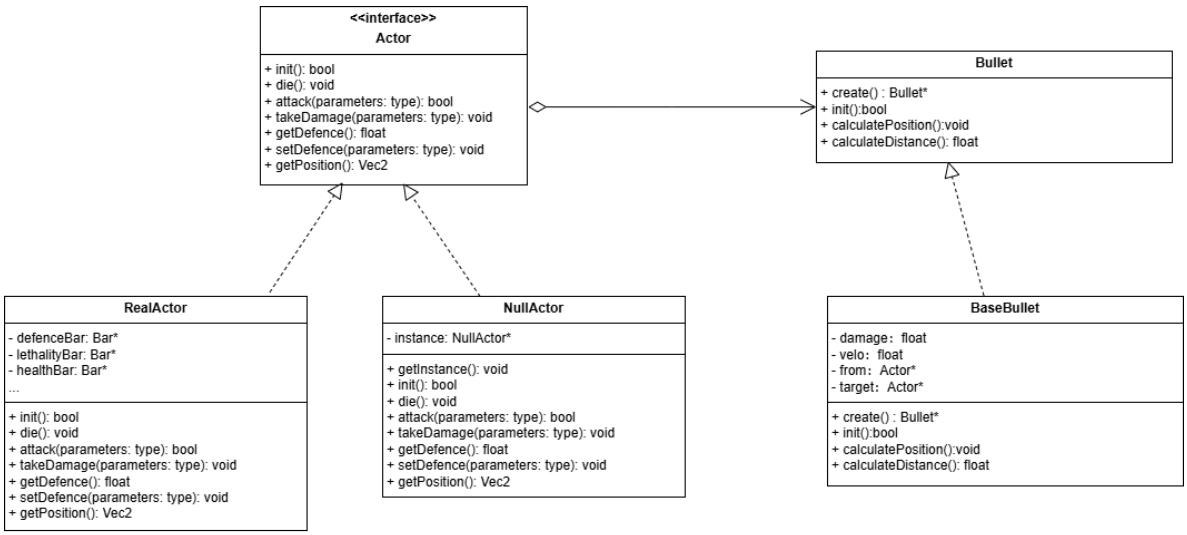
```
// Null Object Pattern: Use the empty object mode to process target
    this->setTarget(target ? target : NullActor::getInstance());
```

5.1.4 UML Class Diagram

The Actor interface defines a series of methods that must be implemented by any concrete class that represents an actor in the system. These methods represent the actor's position. The RealActor class and NullActor class both implement the Actor interface.

RealActor represents a real actor in the system, with actual implementations of all the methods defined in the Actor interface, including handling defensive actions, attacking, and receiving damage. It also has a reference to multiple bars which are used for visualizing the actor's statistics.

On the other hand, the NullActor class is a special implementation of the Actor interface, which acts as a placeholder or empty object. The NullActor class includes a static instance and a getInstance() method that ensures only one instance of the NullActor is used. Its methods provide default, non-functional behavior, such as returning default values for damage and position, to avoid null pointer exceptions in the system when no real actor is available.



5.1.5 Benefits of Refactoring

- **Solved Problem**
  - **Null pointer exception**

    In the original code, target might be nullptr, causing a null pointer exception.By using NullActor, we ensure that target is always a valid object and avoid null pointer exceptions.
  - **Repeated null checks**

    Multiple places in the original code need to check the target! = nullptr. With the empty object pattern, these checks are removed and the code is cleaner.
- **Benefits Refactor Brings**
  - **Code simplification**

    Redundant null checks are removed, making the code more readable. For example, in BaseBullet::init, NullActor::getInstance() is used directly instead of nullptr.

    ```
    this->setTarget(target ? target : NullActor::getInstance());
    ```
  - **Improve robustness**

    By ensuring that target is always active, the risk of potential crashes is reduced.In applyEffect, methods are called directly without checking whether target is null.
  - **Easy to maintain and extend**

    By defining actors as interfaces, RealActor and NullActor implement specific logic and the code structure is clear.

    When you add a role type, you only need to implement the Actor interface.

5.2 Lazy loading Pattern

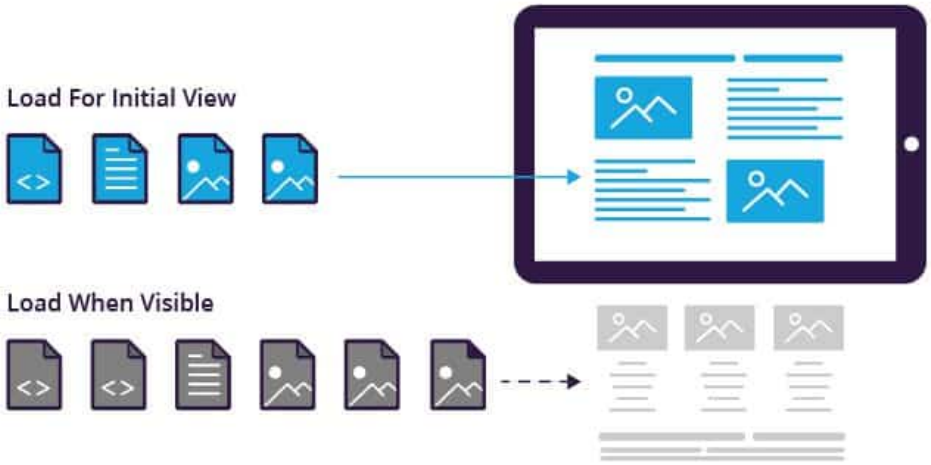
5.2.1 Brief of Lazy loading Pattern

Lazy loading is a structural design pattern that delays the initialization of an object or resource until it is actually needed. This can help optimize performance, reduce memory usage, and improve system efficiency by avoiding the unnecessary creation of objects or resources that might never be used.

- **Key concept**

1. Deferred Initialization: The core idea behind lazy loading is to delay the creation of an object or loading of a resource until it is required. Instead of initializing resources at the start, they are created or fetched only when an actual request for them is made.
2. Proxy Object: In many implementations of lazy loading, a proxy object is used. The proxy acts as a stand-in for the actual object. When the proxy is first accessed, it triggers the loading or initialization of the real object.
3. Performance Optimization: Lazy loading improves system performance by deferring heavy computations, database queries, or network calls until absolutely necessary. This helps reduce the initial load time and resource consumption, especially in cases where the resource might not be used.
4. Consistency: Despite the deferral of initialization, the real object behaves as if it were loaded in advance. This consistency allows the caller to interact with the object in the same way as they would with an eagerly loaded object, without needing to manage the loading process.

Lazy loading is an effective design pattern for improving system performance by deferring the creation and loading of resources until they are actually needed. It allows applications to be more efficient, reduce memory consumption, and provide a better user experience by avoiding unnecessary delays in startup or resource usage. With consistent behavior and deferred initialization, lazy loading is a powerful tool for optimizing performance without complicating the application logic.



5.2.2 Reason for Refactoring

This project is a game, where the map is a core component. The existing map loading code has the following problems, leading to poor performance and inefficient resource management, which necessitates refactoring.

In the original implementation, we defined separate classes such as `NormalMap1`, `NormalMap2`, `NormalMap3`, and `HardMap1` for different map paths and difficulty levels. Each map class implements the `createMap` function to load map images from the `.tmx` file. The code is as follows:

```
TMXTiledMap* NormalMap1::createMap()
{

    TMXTiledMap* map = TMXTiledMap::create("normalmap1.tmx");
    return map;
}
```

We use the `TMXTileMap`'s `create` function to load the map, which does not support lazy loading, and maps are recreated every time a level is entered. This causes the following issues:

- For levels that have not yet been entered, the map files are loaded when the class is created, but these maps might never actually be used.
- Each time a `.tmx` file is loaded, additional time and performance are consumed.
- Even for the same map, re-entering the level causes the map to be reloaded, wasting memory and performance.
- On devices with limited performance, frequent map loading may result in noticeable frame rate drops.

To improve game performance, we adopt the **Lazy Loading Pattern** to refactor the code, enabling lazy loading and resource reuse.

5.2.3 Refactoring Details

1. Static Variable Implementation

   Define the resource to be lazily loaded as a static variable and initialize it upon the first access.

   Added code (using `NormalMap1::createMap` as an example):

   ```
   static TMXTiledMap* map = nullptr;
   ```

2. Load on First Access

   Ensure that the map object is only loaded upon first access, rather than when the map class is created, and only load the map on the first access, reusing it afterward.

   Refactoring logic:

   - Add logic to check whether the map object has already been loaded.
   - If the map object has not been loaded (first access), load the map.
   - If the map object has already been loaded (not the first access), directly return the cached map.

   Code:

```
if (map == nullptr)
{
    map = TMXTiledMap::create("normalmap1.tmx");
}
```

3. Manual Memory Management

   In Cocos2d-x, to reuse the `TMXTiledMap` object, we need to manually manage its memory lifecycle. By default, objects created through the `TMXTiledMap::create()` method are added to the **Auto Release Pool** and destroyed at the end of the current frame. To reuse the map object during the game runtime, its memory lifecycle must be manually managed.

   Added code:

   ```
   map->retain();
   ```

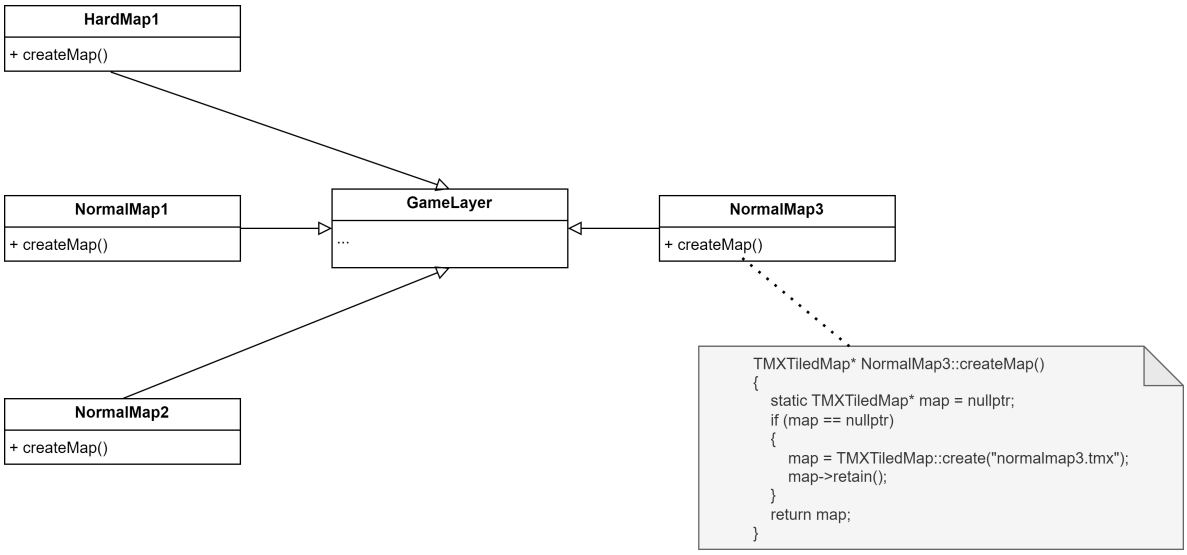
   This enables manual memory management.

At this point, we have completed the lazy loading of maps. The refactored code is as follows:

```
TMXTiledMap* NormalMap1::createMap()
{
    static TMXTiledMap* map = nullptr;
    if (map == nullptr)
    {
        map = TMXTiledMap::create("normalmap1.tmx");
        map->retain();
    }
    return map;
}
```

5.2.4 UML Class Diagram

The UML class diagram is as follows:



We modified the `createMap` function in `NormalMap1`, `NormalMap2`, `NormalMap3`, and `HardMap1`. The class diagram demonstrates the implementation of `NormalMap3::createMap()`.

5.2.5 Benefits of Refactoring

- **Performance Optimization**

    - **Reduced Resource Consumption**:

      The refactored code avoids loading unnecessary map resources. For levels that have not been entered, the maps will not be loaded in advance, reducing memory and CPU usage.

    - **Improved Frame Rate**:

      On devices with limited performance, avoiding frequent map reloads reduces disk I/O and memory allocation pressure, preventing frame drops or stuttering.

- **Resource Reuse**
    - **Caching Mechanism**:

      By using a static variable `map` to cache map objects, the same map object can be reused across multiple level entries, saving memory and load time.

- **Enhanced User Experience**
    - **Smoother Level Loading**:

      The refactored code reuses already loaded maps, avoiding the waiting time for loading maps in each level, providing a smoother gaming experience.

    - **Reduced Stuttering**:

      Optimized map loading prevents stuttering that may occur when loading large map files, especially on resource-constrained devices such as mobile platforms, significantly improving the user experience.

Reference

[1] Wikipedia. (n.d.). Lazy loading. Retrieved January 5, 2025, from https://en.wikipedia.org/wiki/Lazy loading

[2] Wikipedia. (n.d.). Null object pattern. Retrieved January 5, 2025, from https://en.wikipedia.org/i/Null_object_pattern

[3] Refactoring Guru. (n.d.). Design patterns. Retrieved January 5, 2025, from https://refactoring.guru/design-patterns