Refactoring Arknight Project

TEAM 18

2252551 徐俊逸

2253744 林觉凯

2153085 马立欣

2154284 杨骏昊

2151422 武芷朵

CONTENTS

01. Introduction
—— Background

01. Introduction—Background

Originally the course project for *Programming Paradigms* in the fall 2023 semester, aiming to replicate the gameplay of Arknights

A strategy tower defense mobile game, involving deploying Ganyuan on various maps to defend against enemy attacks

Operators are divided into multiple classes (such as Defence, Shooter, Medic), and players need to strategically deploy them based on enemy types.

01. Introduction—Funtionality

01 Ganyuan Features

- Ganyuan Deployment
- Choose Ganyuan
- Ganyuan Skills

02 Enemy Features

- Enemy Types
- Enemy Skill
- Wave-Based Spawning

03 Other Features

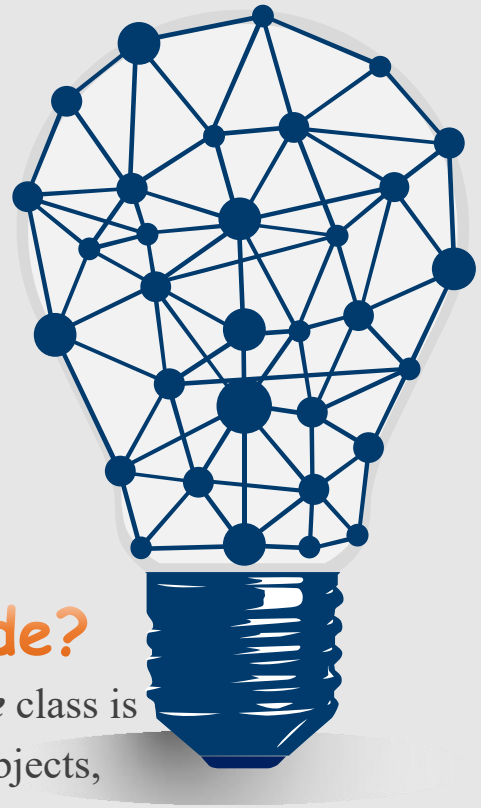
- Game Levels and Difficulty
- Victory and Defeat Screens
- Pause and Restart

02. Creational Pattern
—— Builder

02. Creational Pattern—Builder

Before refactory

```
void Enemy1::setDefaultData() {
    setType(ENEMY1_TYPE);
    scope =Enemy1Scope;
    setLethality(Enemy1Lethality);// Killability
    setHp(Enemy1Hp);  //  Maximum blood count
    setHealth(Enemy1Hp);  // Current blood level
    setDefence(Enemy1Defence);  //  Defence
    setRunSpeed(Enemy1RunSpeed);
    setAlive(true);//If or not you are still alive.
    setIntervalTime(ShieldIntervalTime);//Attack
Interval Time
    setLastAttackTime(GetCurrentTime() /1000.f);
    setIsGround(true);
}
```



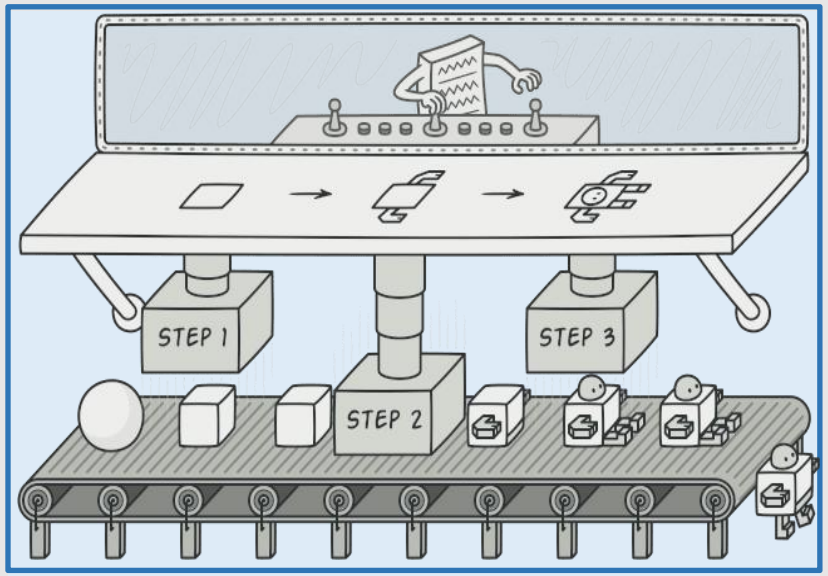
What is the problem with the code?

- **Violate of Single Responsibility Principle**: The ***EnemyBase*** class is responsible for **both** the creation and initialisation of enemy objects, **as well as** the implementation of behavioural control and battle logic.
- **Complex and poorly maintainable**: adding new enemy types requires rewriting many of the same methods, which is not conducive to extension and reuse.

Our Solution

Builder Pattern

The **Builder pattern** allows step-by-step construction of complex objects, permitting flexible creation of products and improving code readability. It separates the construction process from the object representation, ensuring that different representations of the same product can be easily created.



02. Creational Pattern—Builder

1

Define Generic Construction Steps

- Declare all possible build steps in the EnemyCreateBase class.
  - ✓ **new_enemy** —— creation of the Enemy
  - ✓ **type_init** —— initialisation of type-specific attributes
  - ✓ **general_init** —— generic initialisation
  - ✓ **after_init** —— post-creation processing steps (automatic memory management)
  - ✓ **get_result** —— be used to return the final created enemy object.

```
// Enemy.h
class EnemyCreateBase
{
private:
    EnemyBase *e;
public:
    EnemyCreateBase();
    virtual void new_enemy();
    virtual bool type_init() =0;
    virtual void general_init();
    virtual void after_init();
    virtual EnemyBase* get_result();
};
```

```
// Enemy.cpp
bool AirEnemyCreate::type_init(){
    if (!e->Sprite::initWithFile("Pictures/enemy_air.png")){
        returnfalse;
    }
    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);     // Defence
    e -> setRunSpeed(Enemy1RunSpeed);   // Movement speed
    e -> setIntervalTime(ShieldIntervalTime); // Attack interval time
    return true;
}
```

Create concrete builder

- Implement type_specific attribute settings for each enemy to enable the creation of different types of enemies.

2

02. Creational Pattern—Builder

3 Create Director Class

- The **CreateDirector** class encapsulates the construction logic, allowing the user to focus only on the result of the construction and not worry about the details.
- The **EnemyDirector** class is able to pass in the appropriate builder to create different enemys

```
// Enemy.cpp
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.
- Remove the create and initialise functions (‘create’ and ‘init’) from the ‘EnemyBase’ class for proper segregation of duties.

```
// EnemyBase.cpp : remove this two function
EnemyBase* EnemyBase::create(const std::string& filename)
{
    EnemyBase* Base = new(std::nothrow) EnemyBase;
    ...
    return nullptr;
}
bool EnemyBase::init(conststd::string& filename)
{
    if (!Sprite::initWithFile(filename))
    {
        returnfalse;
    }
    ...
    return true;
}
```

02. Creational Pattern—Builder

5 Modify Client Code

- Refactor the addSceneEnemy function in GameLayer so that it calls the director class to create the EnemyBase.
- The idea of the refactoring is that after obtaining the enemy type, the appropriate creator is selected based on the enemy type.

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

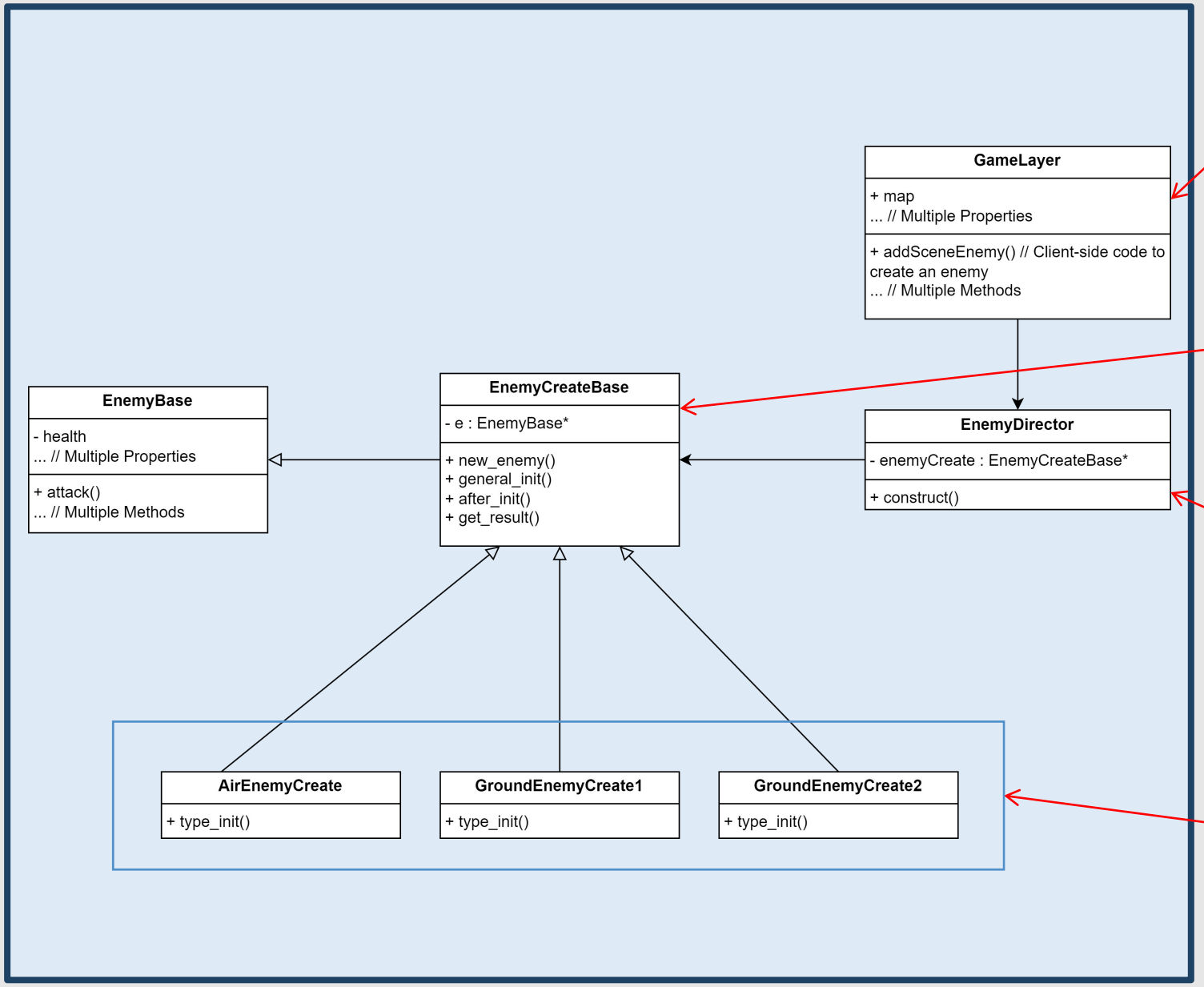
```
// GameLayer.cpp : void GameLayer::addSceneEnemy(float
dt)
...
if (et.type==ENEMY1_TYPE) {

    auto enemy = Enemy1::create();
    enemy->setEntered(false);
    enemy->setLastAttackTime(this->getNowTime());
    ...
    this->addChild(enemy, 10);
    instance->enemyVector.insert(0, enemy);
}
else if (et.type == ENEMY2_TYPE) {
    ...
}
else if (et.type == ENEMY3_TYPE) {
    ...
}
```

Before

```
// Select director class according to 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){
// Calling the director class
    EnemyDirector enemyDirector(enemyCreator);
EnemyBase*enemy=enemyDirector.construct();
enemy->setEntered(false);
    ...}
```

After

02. Creational Pattern—Builder



Clients

The **GameLayer** class acts as the client, calling **EnemyDirector** to create enemies.

The Builder

EnemyCreateBase serves as a **Builder interface**, defining the standard steps for constructing an **Enemy** object.

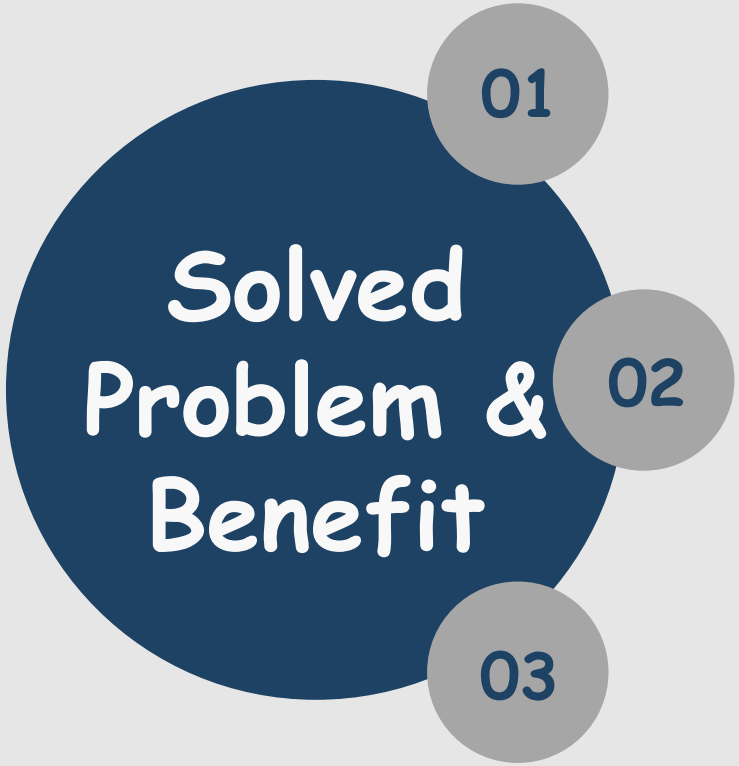
The Director

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.

The Concrete Builder

Each concrete Builder class (e.g., AirEnemyCreate) is responsible for constructing a specific type of enemy.

02. Creational Pattern—Builder



■ Clear Responsibility Separation

Ensures that each class has a single, well-defined responsibility: ***EnemyBase*** focuses on behavior, while ***EnemyCreateBase*** handles object creation. Eliminates mixed responsibilities and adheres to the ***Single Responsibility Principle.***

■ Enhanced Readability

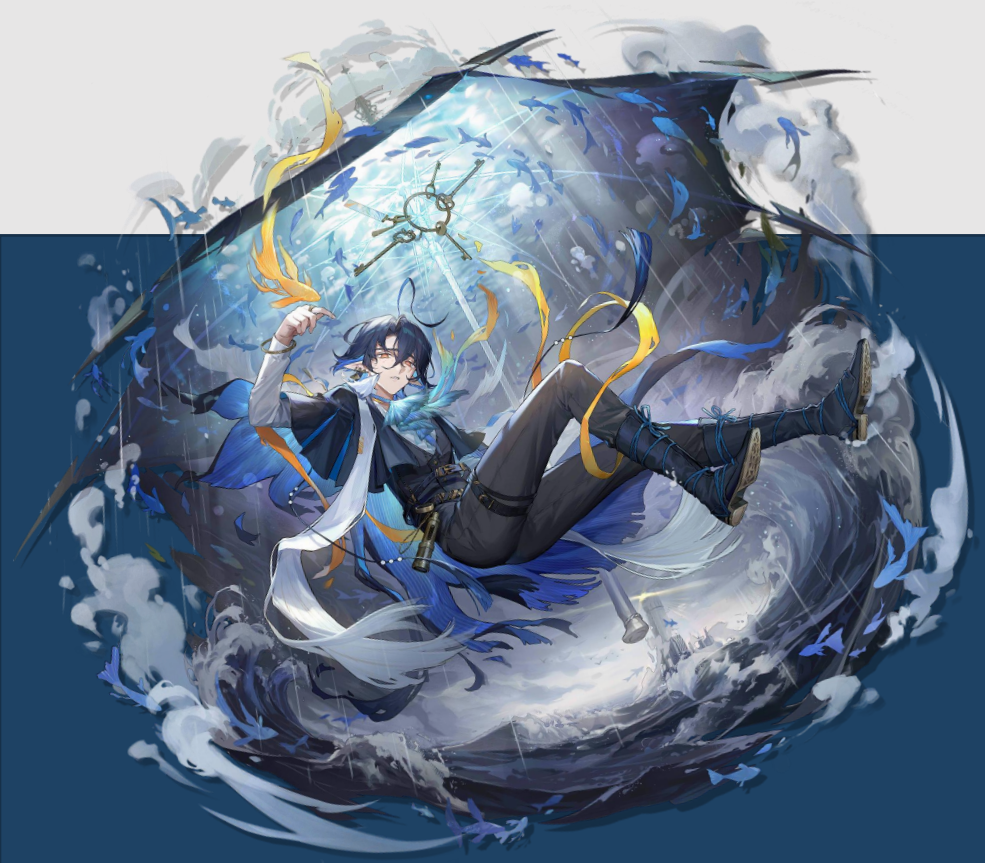
Using the Builder patterns, the code becomes ***clearer and easier to understand***. Each class and method has a specific purpose, avoiding the complex logic and repeated initialization steps seen in the original code.

■ 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.

03. Structural Pattern

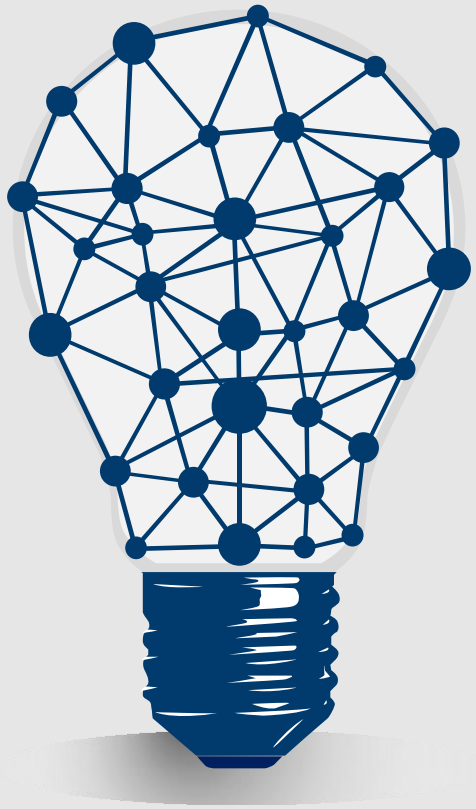
—— Facade



03. Structral Pattern—Facade

Before refactory

```
class GameLayer:public Layer{
public:
    int mapType;
    // Other attributes are omitted here.
    void bulletFlying(float dt);
    void removeBullet(Bullet* pSender);

protected:
    int waveCounter;
    int moneyL;
    int star;
    // Other methods and attributes are omitted
here.
    void menuBackCallback(Ref*pSender);
    void win();
    void lose();
};
```

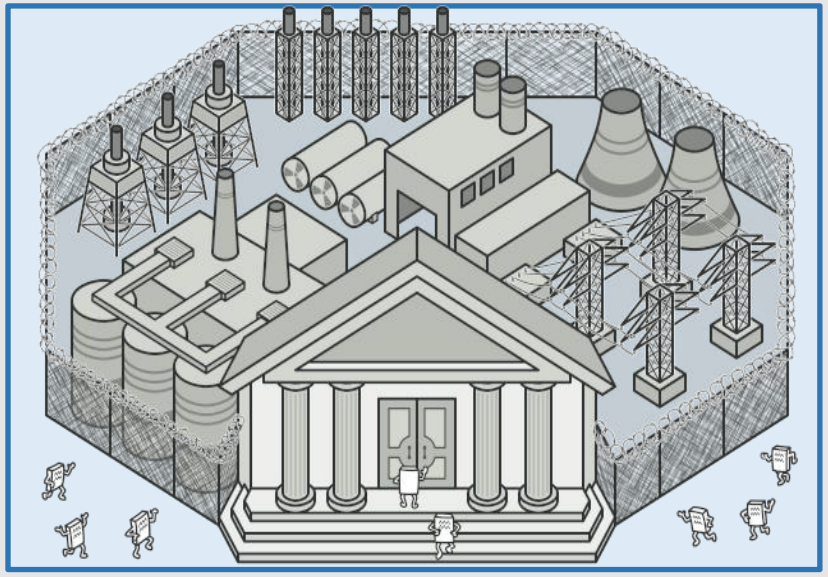


- takes on too many responsibilities
- serves as the primary manager for coordinating various elements
- lacks the Abstraction
- violates the single responsibility principle

Our Solution

Facade Pattern

The Facade Pattern typically involves creating a facade class that interacts with various subsystems, consolidating their functionalities into a streamlined interface.



03. Structral Pattern—Facade

1

Decompose management class and generation subsystem

- Decompose the functionality in GameLayer into several **management classes**
- Utilize these management classes to encapsulate the detailed operations of each subsystem, delegating tasks to the appropriate manager through a simplified and consistent interface.
- Promote modularity and separation of concerns and enhances code readability, maintainability, and scalability.

```
// 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 EnemyManager
class EnemyManager {
public:
EnemyManager(GameManager*gameManager);
    void addSceneEnemy(floatdt);
    void runEnemies();
    void removeBullet(Bullet*pSender);
    void bulletFlying(floatdt);
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;
};
```

03. Structral Pattern—Facade

2

Provide simpler interface based on existing subsystems

- Create a ***GameFacade*** class to encapsulate the interaction between ***EnemyManager***, ***GameLogicManager***, and ***UIManager***, 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.
- Expose 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.

```
// Define GameFacade
class GameFacade{
public:
    GameFacade(GameManager* gameManager,
cocos2d::Layer* parentLayer);
    void initToolLayer();
    void addSceneEnemy(float dt);
    void bulletFlying(float dt);
    void removeBullet(Bullet* pSender);
    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;
};
```

03. Structral Pattern—Facade

3 Declare and implement interfaces in the new Facade class

- Creat the class and declared all required methods.
- Encapsulate calls to various subsystems and provide a simple interface for ***GameLayer*** .

```
// GameFacade.cpp
// Other methods are omitted here.
// Take the initToolLayer and update method as an example
void GameFacade::initToolLayer() {
    uiManager->initToolLayer();
}
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

For example PenetrateBullet

- In the implementation of ***GameFacade***, we redirect each method call to the corresponding subsystem.

```
// GameLayer.cpp
// Other methods are omitted here.
// Take the initToolLayer and update method as an example
bool GameLayer::init(){
    gameFacade->initToolLayer();
    schedule(CC_SCHEDULE_SELECTOR(GameLayer::addSceneEnemy), interval);
    schedule(CC_SCHEDULE_SELECTOR(GameLayer::bulletFlying), 0.1f);
    schedule(CC_SCHEDULE_SELECTOR(GameLayer::updatemoney), 1.0f);
    scheduleUpdate();
}
void GameLayer::update(float dt){
    float nowTime=GetCurrentTime() /1000.f;
    gameFacade->updateGame(startTime, nowTime, star, waveQ, moneyL,
moneyLabel);
}
```

03. Structral Pattern—Facade

5 Initialize the subsystem based on the Facade class

- ***GameLayer*** does not need to worry about subsystem initialization and lifecycle management, and only needs to use the interfaces provided by ***GameFacade***.

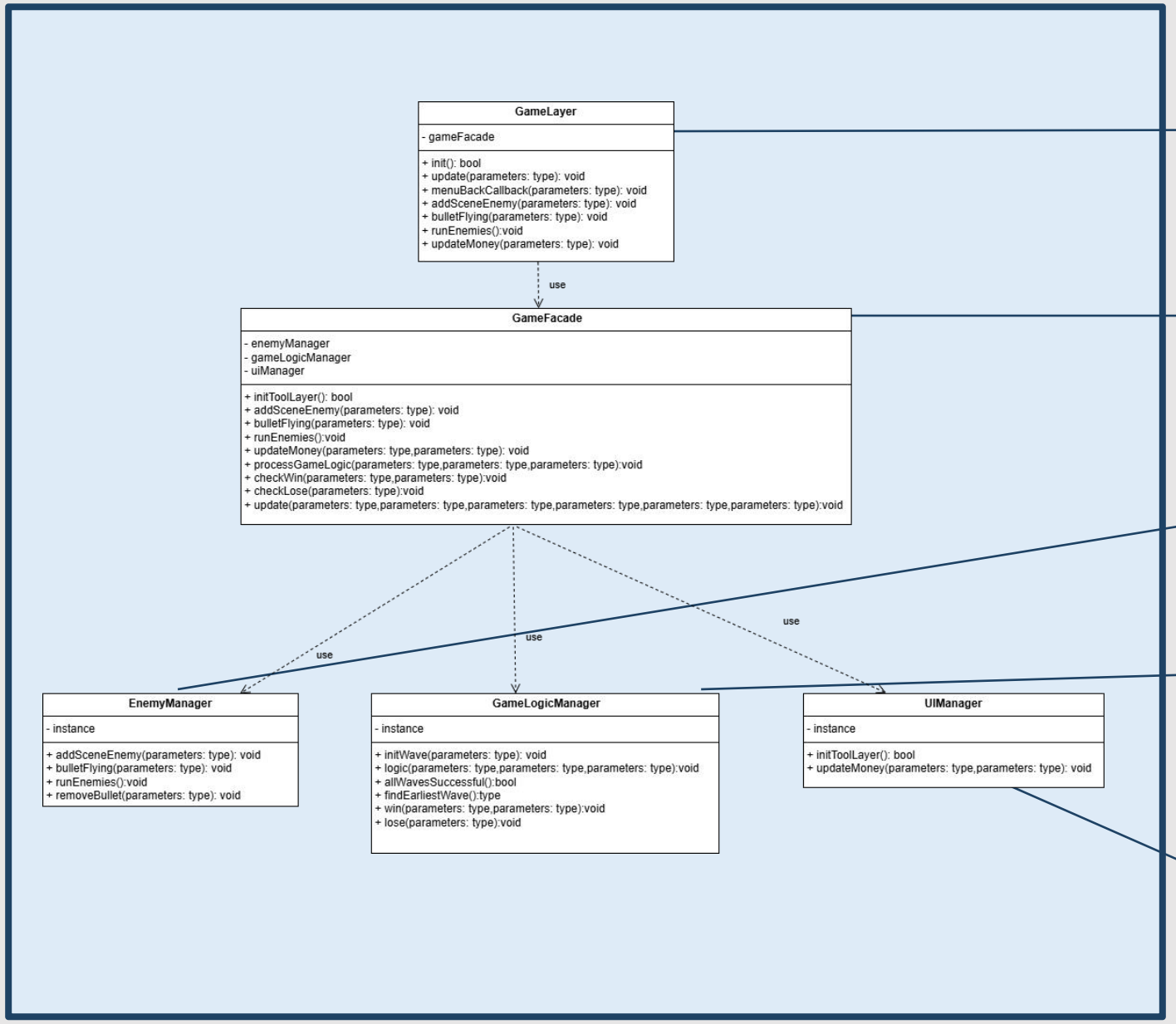
```
// GameFacade.cpp

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*** .

03. Structral Pattern—Facade



GameLayer
the main control layer of the game

GameFacade
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.

03. Structral Pattern—Facade

Solved Problem

Complexity management

Encapsulat these complex interactions into a unified interface, simplifying the logic of GameLayer.

Separation of Responsibilities

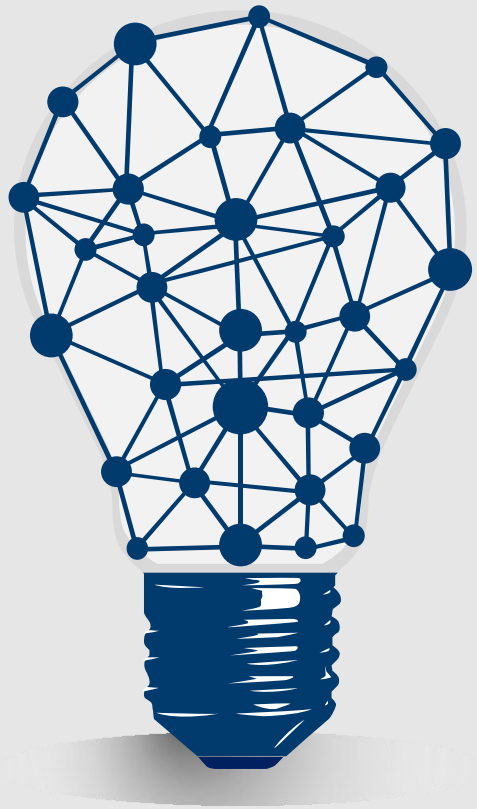
Responsible for calling the GameFacade interface, while the specific logic processing and coordination of interactions between various subsystems are entrusted to GameFacade and various management classes, in accordance with the principle of single responsibility.

Reduce coupling degree

Reduce the coupling between classes, making the system easier to expand and modify.

Improve readability

Restructure code structure is clearer and the logic of *GameLayer* becomes more intuitive.



Benefits Refactor Brings

Enhance maintainability

- Encapsulate 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

- Provide a simple interface for *GameLayer* to use, reducing the complexity of method calls.

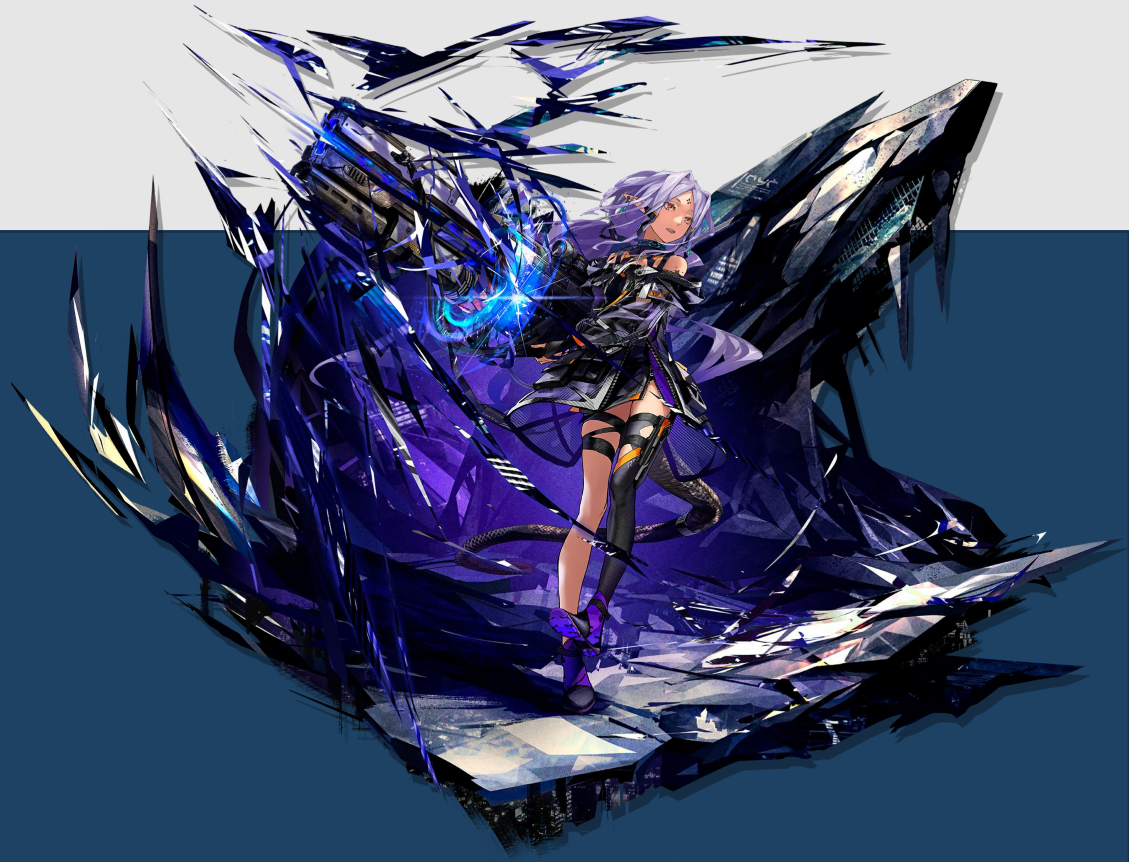
Improve scalability

- 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* .

Principle of Single Responsibility

- *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.

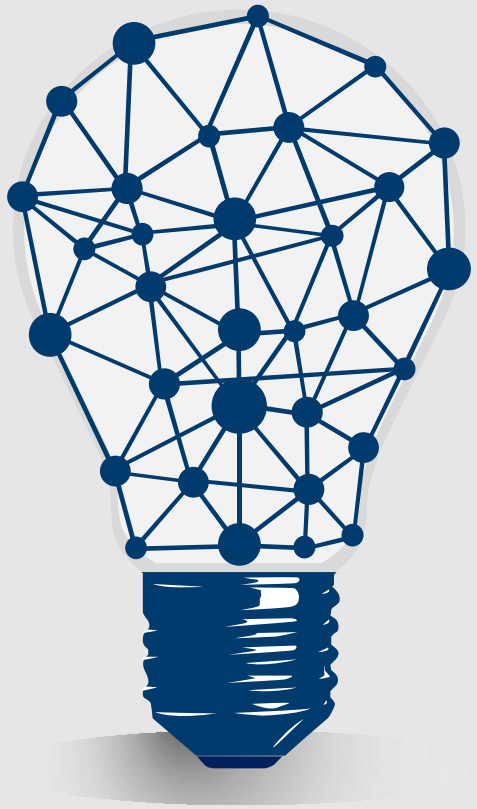
04. Structural Pattern
—— Decorator



04. Structral Pattern—Decorator

Before refactory

```
class Bullet : public Sprite
{
    CC_SYNTHESIZE(float, damage, Damage);
    CC_SYNTHESIZE(float, velo, Velo);
    CC_SYNTHESIZE(Actor*, from, From);
    CC_SYNTHESIZE(Actor*, target, Target);
public:
    virtual bool init(conststd::string&filename,
float damage, float velo, Actor*from,
Actor*target);
    static Bullet*create(conststd::string&filename,
float damage, float velo, Actor*from,
Actor*target);
    void calculatePosition();
    float calculateDistance() const;
};
```



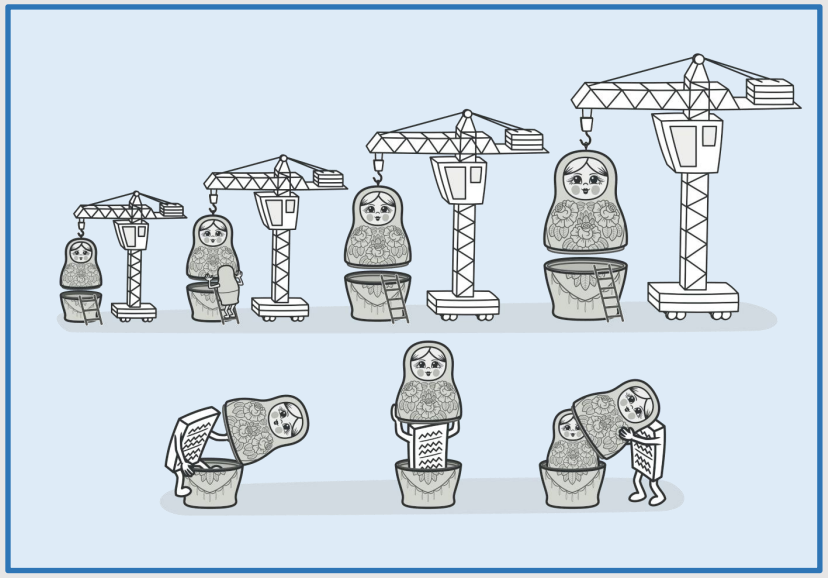
If we need more bullet effect?

- The current bullet class has only one damage attribute
- Lacks the scalability of bullet effects
- All bullet logic would be in one class would result in a bloated class
- violating the single responsibility principle

Our Solution

Decorator Pattern

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.



04. Structral Pattern—Decorator

1

Define Component Interface

- Specifies **the basic behavior** that all bulleted types must implement.
- Ensure that all bulleted classes **share the same interface**, and force subclasses to use purely virtual functions to implement these methods.
- Through pure virtual function, the subclass is forced to **realize the concrete behavior**, so as to achieve the unity and extensibility of the code.

```
// Define component interface
class Bullet : public Sprite
{
public:
    virtual bool init(const std::string&filename,
float damage, floatvelo, 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);
};
```

```
// bullet.h
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, floa tvelo, Actor*from, Actor*target) override;

    virtual void calculatePosition() override;
    virtual float calculateDistance() const override;
    virtual void applyEffect() override;
};
```

Implement Concrete Components

- **The basic bullet class** is implemented, which serves as the basic bullet class and provides basic bullet functions.
- The implementation of the basic functions defined in the basic bullet symbol class can be included in the **bullet.cpp** file, including **the basic behavior of bullets** before refactoring the code.

2

04. Structral Pattern—Decorator

3 Create Base Decorator

- Responsible for maintaining references to **decorative objects** and **delegating all operations** to this decorative object.
- The methods in the base decorator are all delegated to **the decorator object** to perform the actual operation.

```
class BulletDecorator : public Bullet
{
protected:
    Bullet *m_bullet;
public:
    BulletDecorator(Bullet*bullet):m_bullet(bullet) {}
      // Other methods are omitted here.
     // Take the applyEffects method as an example
     // Decorator effect delegate
    virtual void applyEffect() override{
        m_bullet->applyEffect();
    }
};
```

The decorator calls the corresponding implementation of the wrapped object.

4 Implement Concrete Decorators

Add specific behavior or additional effects to your project by creating concrete decoration classes.

NormalBullet
Same as base bullets, dealing a certain amount of damage.

HealingBullet
A medic fires at friendly forces, restoring health, i.e. healing bullets deal negative damage.

PenetrateBullet
Ignore the enemy's defense when hitting them, and restore the enemy's defense after dealing damage.

SlowBullet
Deals a small amount of damage while slowing the movement of the attacked enemy for 3 seconds.

MeleeBullet
Fired by shield soldiers, it deals more damage to the enemy due to its close range.

For example PenetrateBullet

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

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

04. Structral Pattern—Decorator

```cpp
// EnemyBase.cpp
bool EnemyBase::attack(Actor*target)
{
    ...
    // Create base bullet
    auto baseBullet =BaseBullet::create(tmpPath, this->getLethality(),
3000, this, target);
    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
    elseif (this->getType() == ENEMY2_TYPE ||
             this->getType() == ENEMY3_TYPE){
        if (target->getType() == SHOOTER_TYPE ||
            target->getType() == MEDICAL_TYPE ||
            target->getType() == SHIELD_TYPE){
            bullet =PenetrateBullet::create(baseBullet);
        }
    }
    if (bullet == baseBullet){
        CC_SAFE_DELETE(baseBullet);
        returnfalse;
    }
    // Set bullet position and scale
    bullet->setPosition(this->getPosition());
    bullet->setScale(0.12);
    instance->gameScene->addChild(bullet);
    instance->bulletVector.pushBack(dynamic_cast<Sprite *>(bullet));
    returntrue;
}
```

5 Client Usage

First you need to create the base bullet.

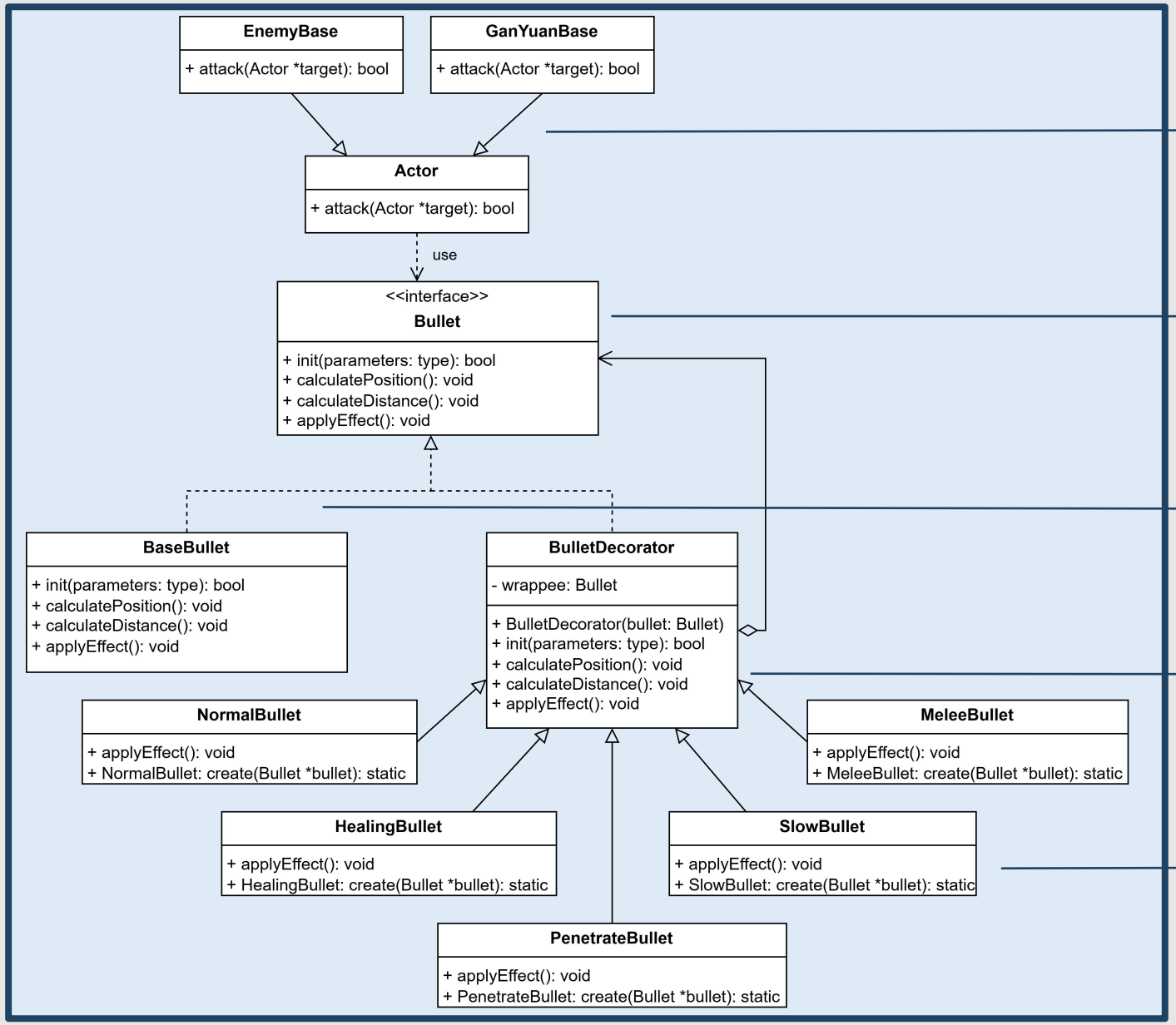
```cpp
auto baseBullet =BasicBullet::create(tmpPath, this->getLethality(), 2000, this, target);
```

Then wrap the base bullet with a concrete decorator.

```cpp
bullet = new NormalBullet(baseBullet)     // normal bullet
bullet = new PenetrateBullet(baseBullet); // penetrate bullet
bullet = new SlowBullet(baseBullet);      // slow bullet
bullet = new HealingBullet(baseBullet);   // healing bullet
bullet = new MeleeBullet(baseBullet)      // melee bullet
```

There are now five bullets in the project, and if you need to expand or combine new bullets, it is much faster to use Decorator pattern.

04. Structral Pattern—Decorator



Clients

The client achieves flexible bullet behavior by combining different decorators.

The Component interface

Defines the common methods that all bullet classes must implement.

The Concrete Component

Implements the Bullet interface and provides the basic functionality of bullets.

The Base Decorator

The base decorator implements the extension of bullet behavior by delegating all method calls to wrappee.

Concrete decorator

Each decorator adds specific behavior to the applyEffect() method.

04. Structral Pattern—Decorator

Solved Problem

Potential Class Explosion Problem

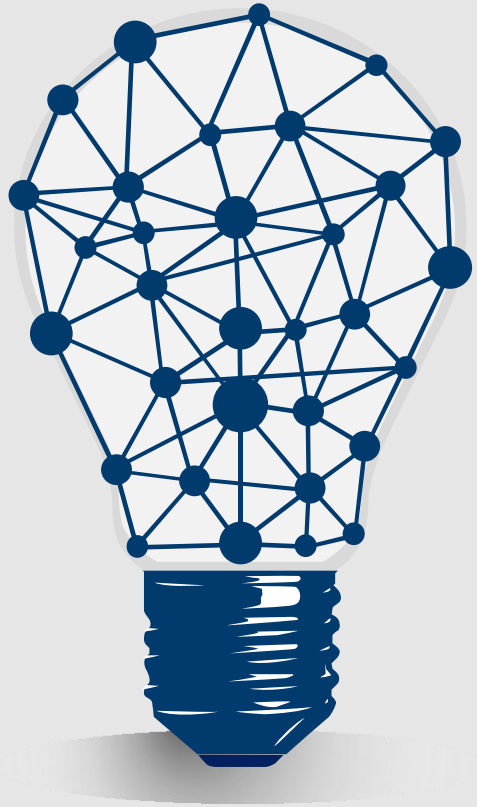
- Avoid the generation of a large number of subclasses, thus solving the potential class explosion problem.

Potential Code Duplication

- Common behavior is extracted into the base and bullet decorator, reducing the amount of duplicate code and improving code reusability.

Poor Scalability

- Before refactoring, adding new effects requires modifying existing code, increasing the risk of modifying existing classes.



Benefits Refactor Brings

Increased Flexibility

- Different decorators can be combined flexibly, making the combination of bullet effects more flexible and adaptable to more needs.

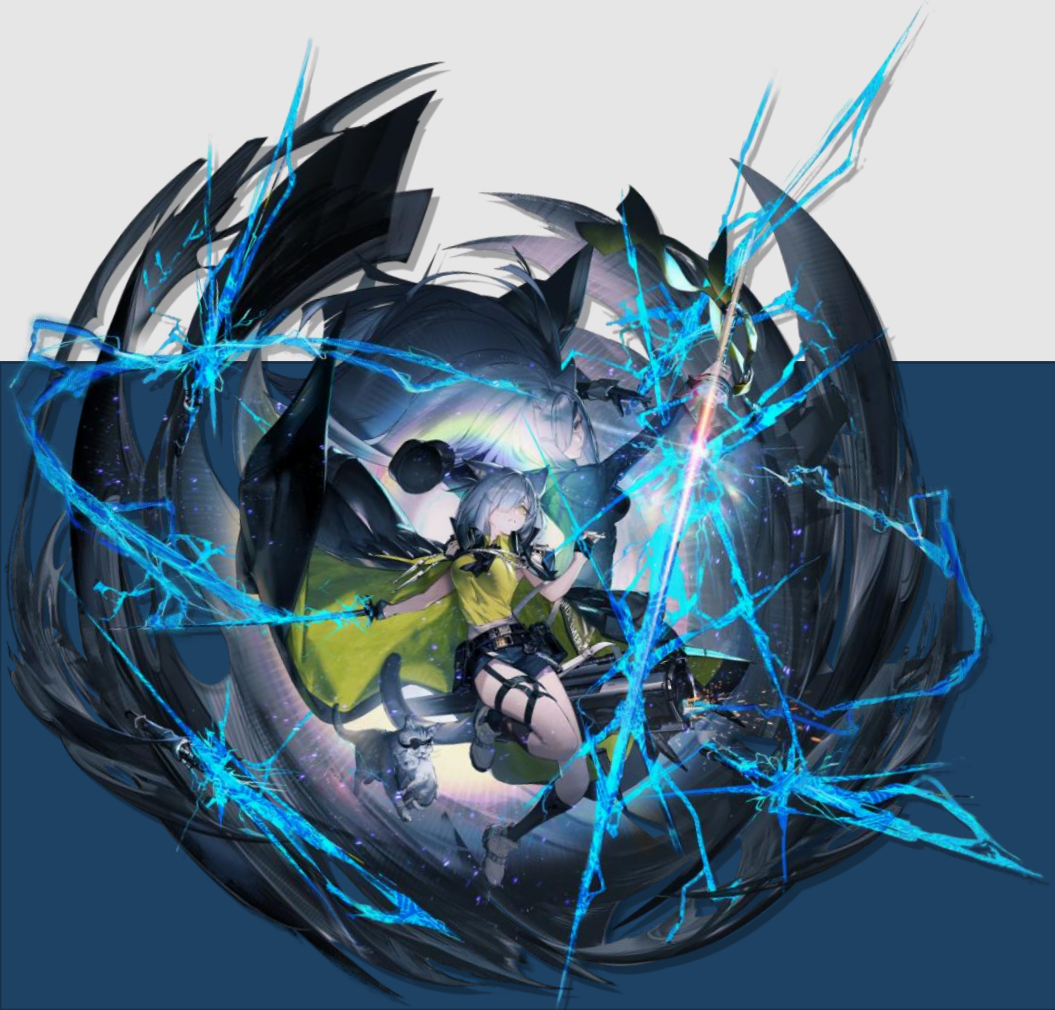
Open and Close Principle

- With the decorator mode, new bullet effects can be implemented by adding new decorator classes without modifying existing code.

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.

05. Behavioral Pattern
—— State



05. Behavioral Pattern—State

Before refactory

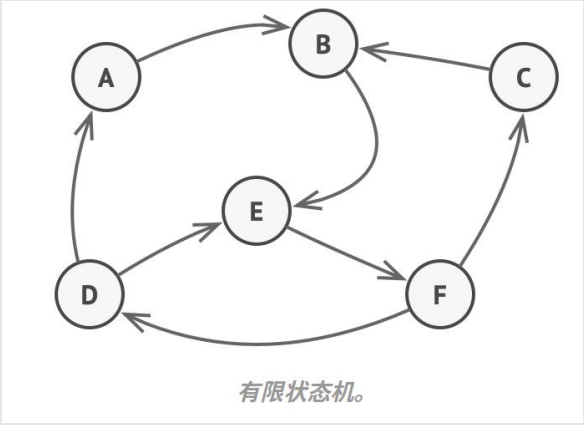
```
void EnemyBase::runToFollowPoint(){
    ...
    if (getMov() == STILL){
        if (ptr < positions.size() - 1){
            ...
            this->setMov(ATTACKING);
        }
        if (this->getMov() != ATTACKING){
            ...
            this->setMov(RUNNING);
        }
    }
    if (getMov() == RUNNING){
        ...
       this->setMov(STILL);
    }
    if (this->getMov() == ATTACKING){
        ...
        this->setMov(STILL);
    }
}
```

Problem

- **High complexity and poor readability**: Nested if statements make the logic hard to follow and maintain.
- **Strong coupling between states and behavior**: State logic is intertwined, making changes to one state risky for others.
- **Poor extensibility**: Adding or modifying states requires changes to the core function, violating the Open/Closed Principle
- **Testing challenges**: Centralized state logic makes it difficult to test each state independently.

Solution: State Pattern

State Strategy is a behavioral design pattern that models an object's behavior based on its internal state, by encapsulating state-specific behaviors into classes.



有限状态机。

05. Behavioral Pattern—State

1

State Interface Declaration

- Create **abstract class EnemyState** with three core methods: enter(), execute(), and exit().
- Add **protected helper methods** like getRoadPositions(), getNextPathPoint(), etc.

```
class EnemyState{
protected:
    EnemyBase *enemy;
public:
    EnemyState(EnemyBase *enemy) : enemy(enemy) {}
    virtual ~EnemyState() {}
    virtual void enter() = 0;
    virtual void execute() = 0;
    virtual void exit() = 0;
protected:// helper method
    std::vector<Vec2> getRoadPositions(){...}
};
```

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

Concrete State Classes Creation

- Create three **concrete state classes**: StillState, RunningState, and AttackingState.
- Each **inherits from EnemyState** and implements the three core methods.

2

05. Behavioral Pattern—State

3 State-Specific Code Extraction

- 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 if blocked
    if (isBlockedByGanYuan(enemy, nextPoint)){
        transitToAttackingState(enemy);
        return;}
    float duration = calculateMoveDuration(enemy,
enemy->getCurrentPosition(), nextPoint);
    enemy->moveToPosition(nextPoint, duration);
    transitToRunningState(enemy);}
void RunningState::execute(EnemyBase *enemy){...}
void AttackingState::execute(EnemyBase *enemy){...}
```

```
class EnemyBase : public Actor{
private:
    EnemyState *state; // 当前状态
public:
    EnemyBase();
    virtual bool init(const std::string &filename);
    static EnemyBase *create(const std::string &filename);
    // state related method
    void changeState(EnemyState *newState);
    void runToFollowPoint();
    // helper method
    Vec2 getCurrentPosition() const { return getPosition();
}
    ...
}
```

4 State Reference Addition

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

05. Behavioral Pattern—State

```
// EnemyBase.cpp
EnemyBase::EnemyBase()
    : runSpeed(0)
    , entered(false)
    , ptr(0)
    , mov(STILL)
{state = new StillState();}
EnemyBase::~EnemyBase() {delete state;}

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

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

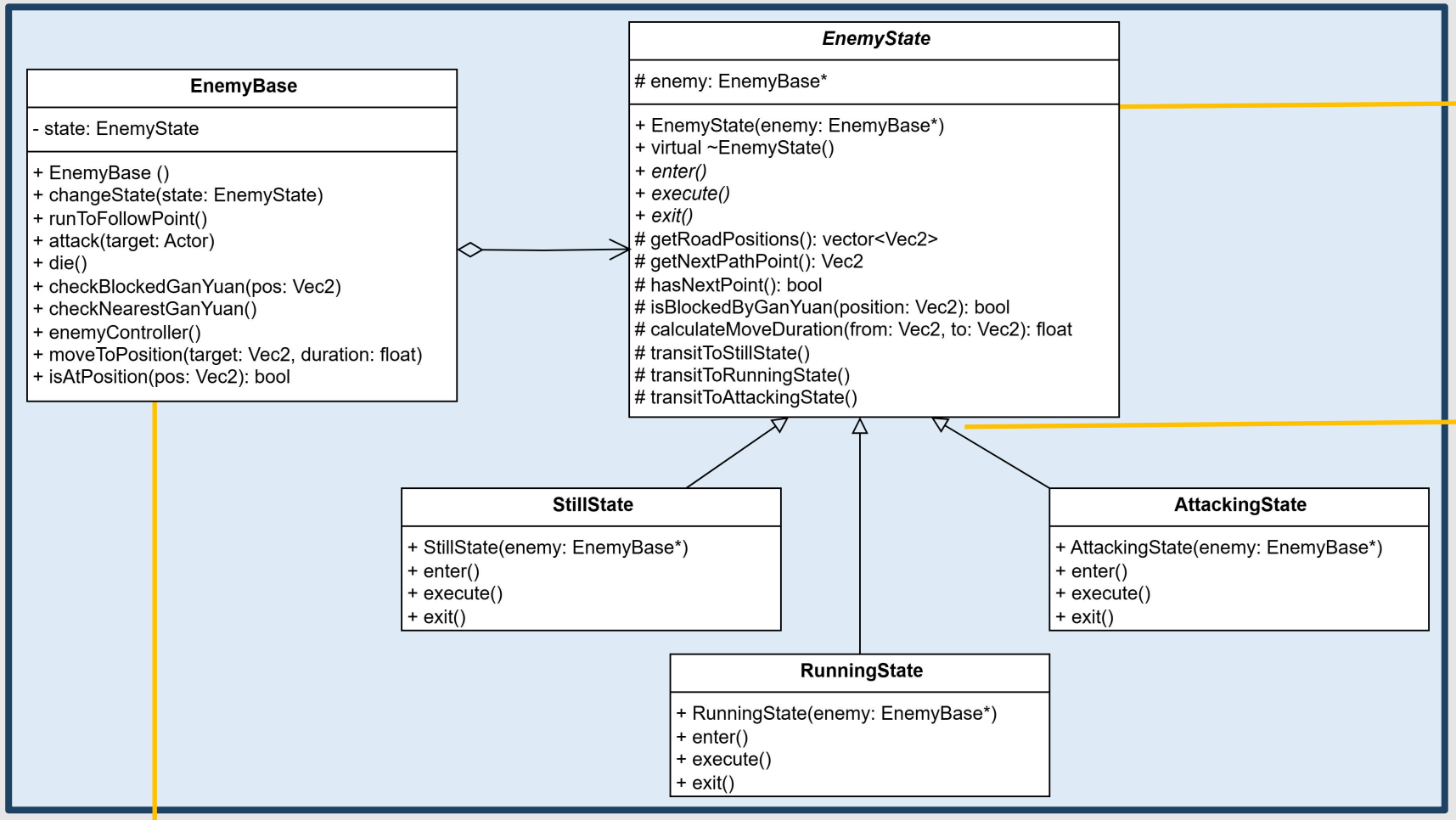
5 Change Context Accordingly

First you need to setup initial state.

Then you need to finnish state transitions implementation

Lastly, the original complex runToFollowPoint() with state conditionals replaced by this.

05. Behavioral Pattern—State



EnemyState
An abstract base class for states and contains common method such an enter(), execute(), exit().

StillState/AttackingState/ RunningState
Derived Class from base class and each implements those methods accordingly.

EnemyBase
The context class of states, and compose aggregation relationship with the states base class

05. Behavioral Pattern—State

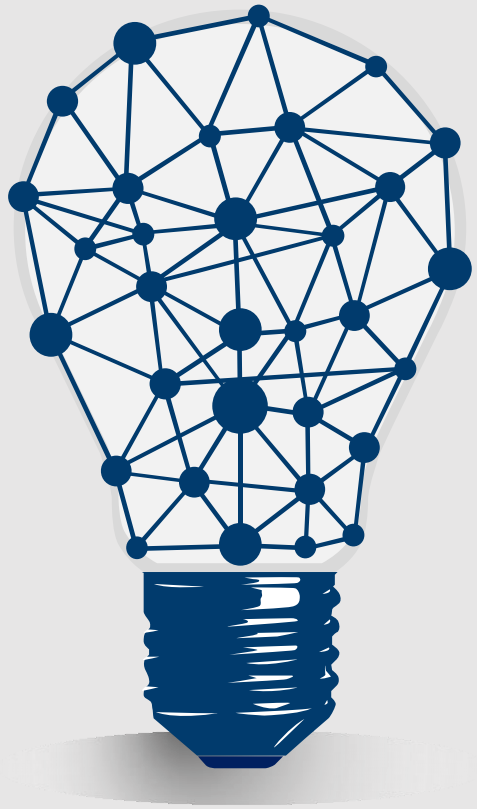
Benefits Refactor Brings

Improved Code Maintainability

- Encapsulating state logic in classes simplifies maintenance and avoids interference between states.

Easier Testing

- Isolated state logic enables independent testing and debugging, reducing risk of cascading errors.



Enhanced Scalability

- Adding new states becomes easier without modifying existing code, adhering to Open/Closed Principle.

Reduced Code Complexity

- Separating logic into classes improves readability, avoiding nested if statements in monolithic functions

06. Behavioral Pattern
—— Strategy

06. Behavioral Pattern—Strategy

Before refactory

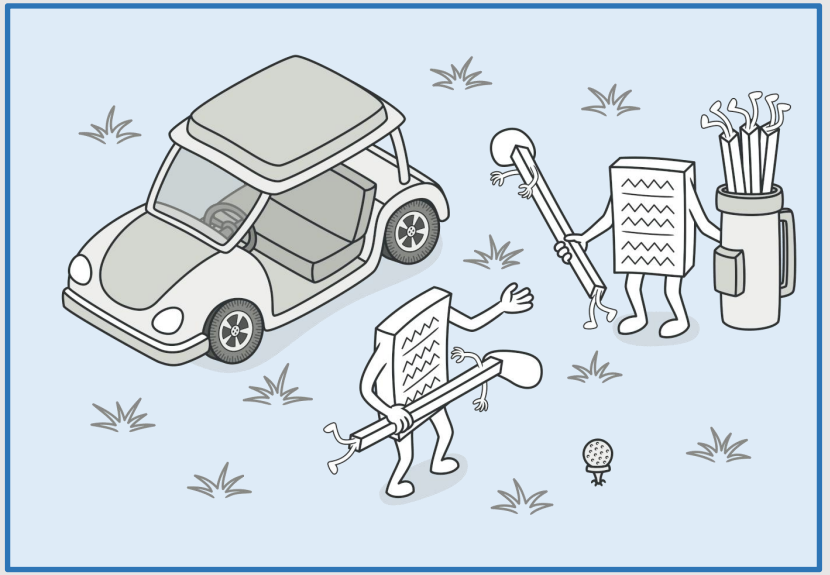
```
void GanYuanMedical::positionLegal(bool& state, Vec2& p) {
    GameManager* instance = GameManager::getInstance();
    for (int i = 0; i < instance->towersPosition.size(); i++) {
        ...
    }
    return;
}

void GanYuanShield:: positionLegal(bool& state, Vec2& p) {
    GameManager* instance = GameManager::getInstance();
    for (int i = 0; i < instance->groundsPosition.size(); i++)
{
        ...
    }
    return;
}

void GanYuanShooter::positionLegal(bool& state, Vec2& p) {
    GameManager* instance = GameManager::getInstance();
    for (int i = 0; i < instance->towersPosition.size(); i++) {
        ...
    }
    return;
}
```

Problem: duplication!

Our Solution

Stragety Pattern

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.



06. Behavioral Pattern—Strategy

1

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***.

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

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

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.

2

06. Behavioral Pattern—Strategy

3 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();
}
```

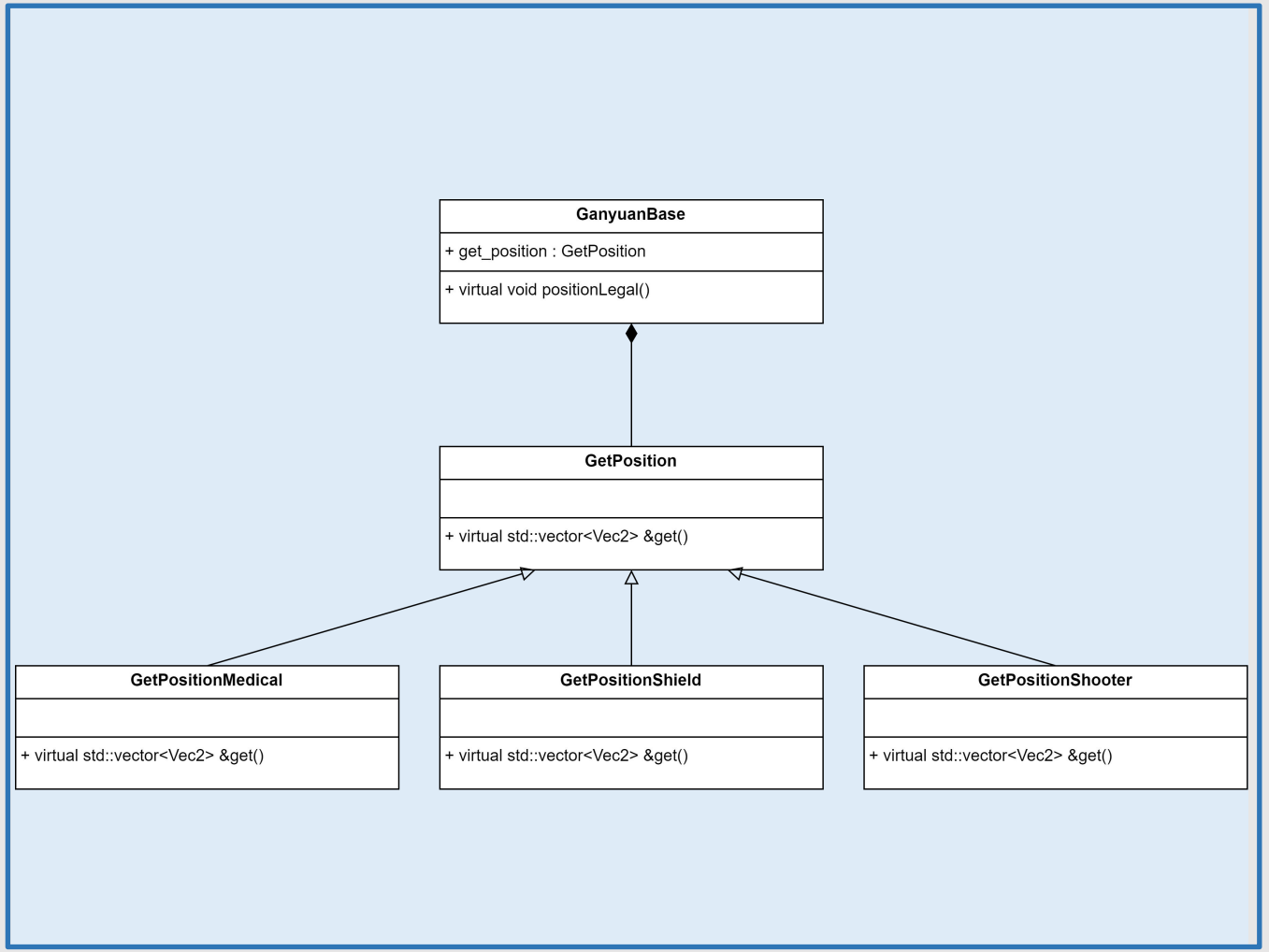
06. Behavioral Pattern—Strategy

4 Extract the common logic

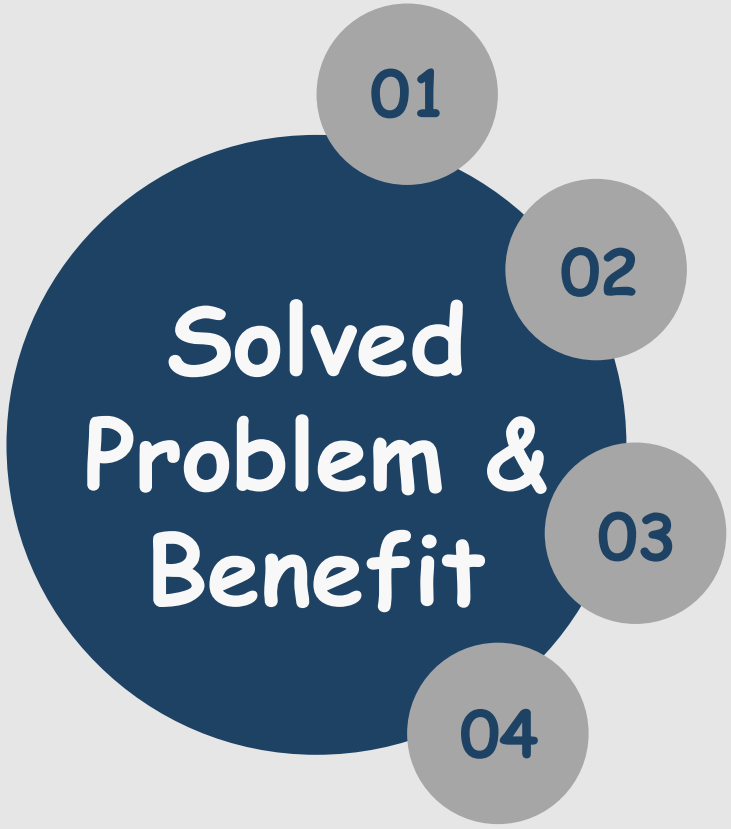
```
//GanYuanBase.cpp
void GanYuanBase::positionLegal(bool& state, Vec2&
p) {
    GameManager* instance =
GameManager::getInstance();
    auto pos = this->get_position->get(instance);
    for (int i = 0; i < pos.size(); i++) {
        //(road_path[i - 1] -
road_path[i]).getLength()
        if ((this->getPosition()).distance(pos[i])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

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

UML Class Diagram:



06. Behavioral Pattern—Strategy



■ The Potential Code Duplication

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.

■ 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.

■ 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.

■ 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.

07. Additional Pattern
—— Null Object

07. Additional Pattern — Null Object

Before refactory

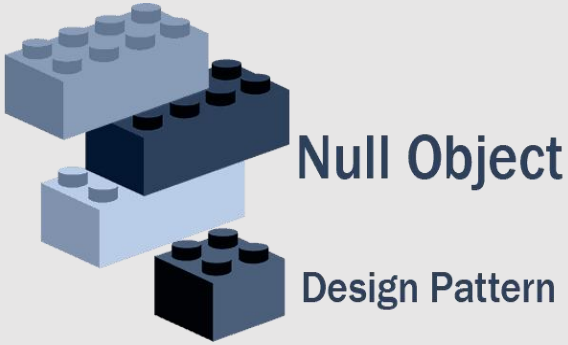
```
// Decorator Pattern: basic bullet position calculation
void BaseBullet::calculatePosition()
{
    if (target != NULL){
        ...
    }
}
// basic bullet distance calculation
float BaseBullet::calculateDistance() const
{
    if (target != NULL){
        ....
    }
    return 0;
}
// Decorator Pattern: basic bullet effect application
void BaseBullet::applyEffect()
{
    if (target != NULL){
        ....
    }
}
```

Problem

- **Dangling pointer risk**: Target objects may be destroyed while bullets are in flight, leading to potential crashes.
- **Maintenance overhead**: Duplicated null-checking logic makes code verbose and difficult to update consistently.
- Dangling pointer risk: Target objects may be destroyed while bullets are in flight, leading to potential crashes.

Solution: Null Pattern Pattern

The empty object pattern is ***a behavior design pattern*** that provides proxy objects that implement interfaces but ***do not perform operations***. This pattern returns an object that provides the default "do nothing" behavior.



07. Additional Pattern—Null Object

1

Actor Interface Declaration

- Create abstract class Actor with pure virtual methods that define the core interface all roles must implement.
- Add pure virtual methods for common behaviors like movement, state updates, and interactions that all derived classes must override.

2

State Interface Declaration

- **Create a RealActor class**: Move the implementation code from the original Actor class into the RealActor class.
- **Create the NullActor class**: inherits from Actor. Implement all virtual functions, but provide empty operations or return default values.
- Also implement **singleton pattern**

```
// Actor.cpp
// Null Object Pattern: Actor class
class Actor : public cocos2d::Sprite{
public:
    ...
};
// Null Object Pattern: RealActor class
class RealActor : public Actor{
protected:
    ...
public:
    ...
};
// 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
};
```

07. Additional Pattern—Null Object

3 State Interface Declaration (Continued)

- Because null Object's function is relatively simple, the override function is written in the.h file.

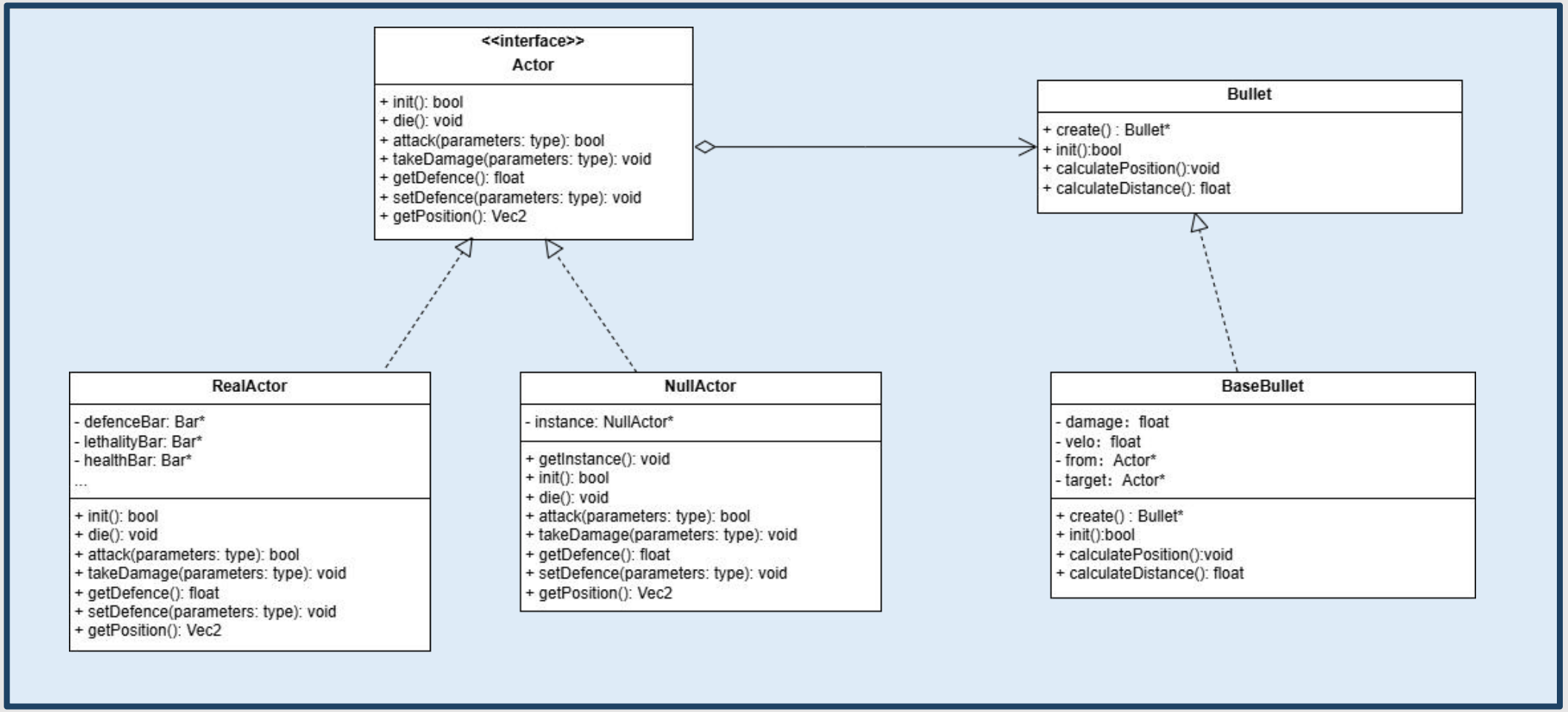
```
// Actor.cpp
// Null Object Pattern: NullActor class
NullActor *NullActor::getInstance()
{
    if (instance == nullptr)
    {
        instance = new NullActor();
    }
    return instance;
}
```

4 Modify Bullet.cpp

- In the BaseBullet::init method, NullActor::getInstance() is used instead of nullptr. Remove null checks from other methods and call target's method directly.

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

07. Additional Pattern—Null Object



Actor Interface
Defines core methods for game entities, requiring implementation of position, combat, and state management behaviors.

RealActor
Implements concrete actor behaviors with actual game logic and visual statistics through status bars.

NullActor
Serves as a singleton placeholder implementing Actor interface with default behaviors to prevent null pointer exceptions.

07. Additional Pattern—Null Object

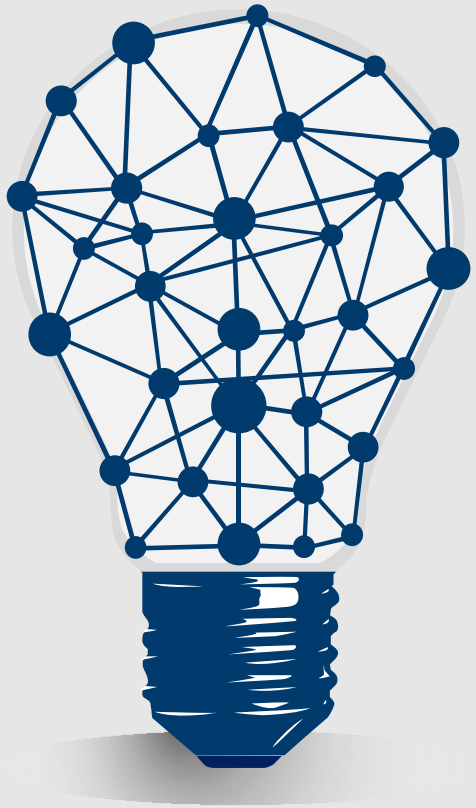
Solved Problem

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 from refactoring

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..

Easy to maintain and extend

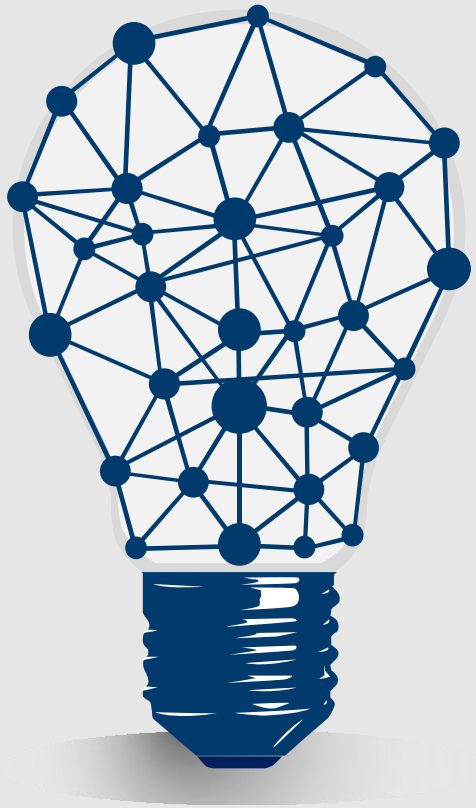
- By defining actors as interfaces, RealActor and NullActor implement specific logic and the code structure is clear.When you add a role type, you only need to implement the Actor interface.

08. Additional Pattern
—— Lazy Loading

08. Additional Pattern—Lazy loading

Before refactory

```
TMXTiledMap* NormalMap1::createMap() {
    TMXTiledMap* map =
TMXTiledMap::create("normalmap1.tmx");
    return map;
}
TMXTiledMap* NormalMap2::createMap() {
    TMXTiledMap* map =
TMXTiledMap::create("normalmap1.tmx");
    return map;
}
TMXTiledMap* NormalMap3::createMap() {
    TMXTiledMap* map =
TMXTiledMap::create("normalmap1.tmx");
    return map;
}
TMXTiledMap* HardMap1::createMap() {
    TMXTiledMap* map =
TMXTiledMap::create("normalmap1.tmx");
    return map;
}
```

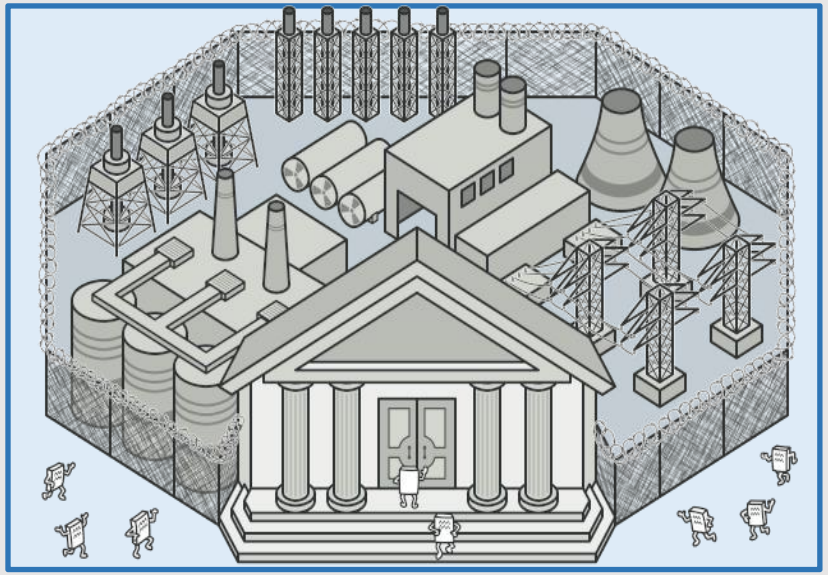


- The map files not used are loaded when the class is created.
- Each time a `.tmx` file is loaded, additional time and performance are consumed.
- Even for the same map, re-entering the level causes the map to be reloaded.
- On devices with limited performance, frequent map loading may result in noticeable frame rate drops.

Our Solution

Lazy loading

Lazy loading helps optimize performance, reduce memory usage, and improve system efficiency by avoiding the unnecessary creation of objects or resources that might never be used.



08. Additional Pattern—Lazy Loading

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

1. Static Variable Implementation

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

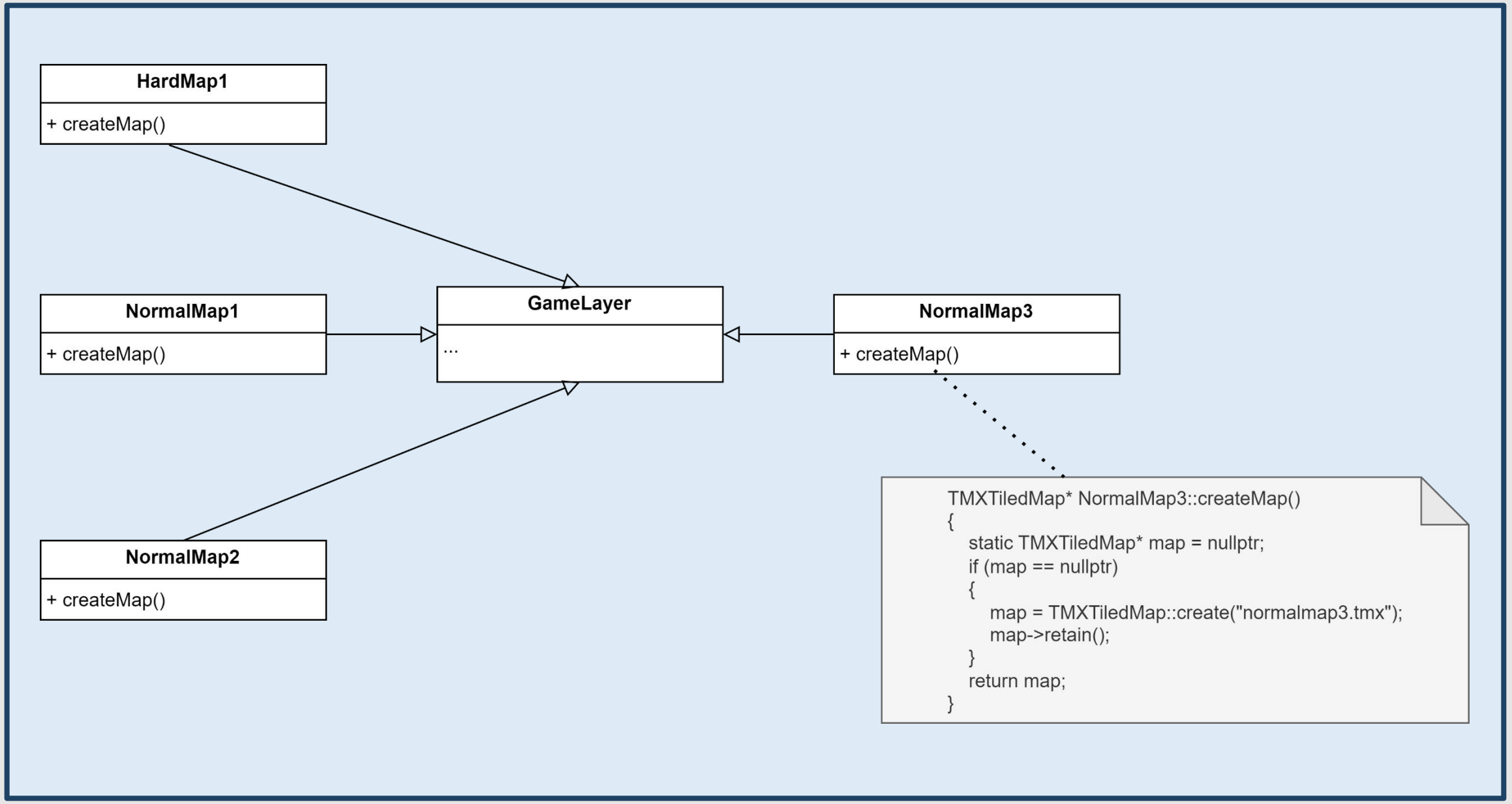
2. Load on First Access

- 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.

3. Manual Memory Management

Manage memory manually with retain to avoid autodestruction.

08. Additional Pattern—Lazy loading



08. Additional Pattern—Lazy loading

Benefits Refactor Brings

Reduced Resource Consumption

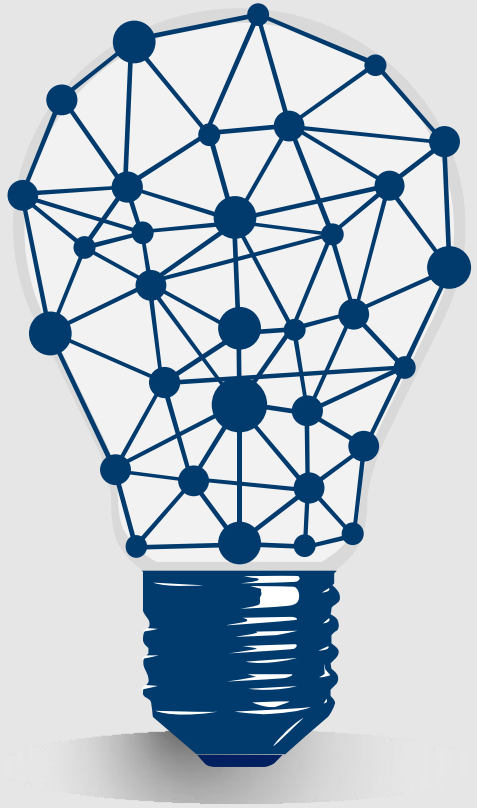
The refactored code avoids loading unnecessary map resources.

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.

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.



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.

References

	Doucument Title	Source
[1]	Lazy Loading. Wikipedia.	https://en.wikipedia.org/wiki/Lazy_loading
[2]	Null Object Pattern. Wikipedia.	https://en.wikipedia.org/wiki/Null_object_pattern
[3]	Design Patterns. Refactoring Guru.	https://refactoring.guru/design-patterns

Thanks for listening!

TEAM 18

2153085 马立欣

2252551 徐俊逸

2154284 杨骏昊

2253744 林觉凯

2151422 武芷朵