Teamfight Tactics Project Documentation

Contents:

Project Name: Teamfight Tactics (2023 Tongji University **Programming Paradigms** Course Project)

Team Number: Group 16

Team Members:

Full name	Matriculation numbers	Contact number	Email address
林继申 Jishen LIN	2250758	15143305542	2250758@tongji.edu.cn
刘淑仪 Shuyi LIU	2251730	15256633616	2251730@tongji.edu.cn
杨宇琨 Yukun YANG	2252843	18186206130	2242843@tongji.edu.cn

中谷天音 AMANE NAKATANI	2256225	18202190215	2256225@tongji.edu.cn

> **Notes:** This project is developed based on the Cocos2d-x 3.17.2 engine, and the source code has a large overall size. To facilitate transmission and storage, the Cocos2d library function code and resources files (including audio, video, and other design materials) have been removed from the compressed package, **retaining only the core logic code of the project and the refactored code sections**. The refactored code has been clearly marked with comments to distinguish it from the original code. For instructions on running or building this project, please refer to the relevant documentation in the original GitHub repository.

Project Overview

Project Background

With the rise of Auto Chess-style games such as *Dota Auto Chess*, *Teamfight Tactics*, and *Chess Rush*, these games have attracted a large number of players due to their unique combination of strategy and randomness. The core gameplay of Auto Chess games revolves around collecting, upgrading, and combining different hero cards to create powerful lineups and battle against other players or AI. These games test players' strategic thinking while providing a rich gaming experience and social interaction.

In response to this trend, this project aims to develop a similar Auto Chess game using the Cocos2d-x 3.17.2 engine. By drawing inspiration from classic games like *Dota Auto Chess*, *Teamfight Tactics*, and *Chess Rush*, the project not only implements the basic Auto Chess mechanics, such as card collection, upgrades, and lineup building, but also expands on the game’ s strategic depth by introducing various synergy and rune enhancement features. These additions increase the strategic complexity and replayability of the game. Additionally, the game supports both single-player practice modes and multiplayer online battles, allowing players to face off against AI or other players for an enriched experience.

The project also emphasizes the audiovisual experience, with captivating initial and settings menus, background music, and battle sound effects, enhancing immersion. Through this project, players can enjoy the fun of collecting, upgrading, and combining cards in a strategic world while competing with friends or other players.

Project Introduction

This project is an Auto Chess game developed using the Cocos2d-x 3.17.2 engine, inspired by classic Auto Chess games like *Dota Auto Chess*, *Teamfight Tactics*, and *Chess Rush*. The game is strategy-based, where players collect, upgrade, and combine different hero cards to build powerful lineups and battle against AI or other players. The game not only replicates the core

mechanics of traditional Auto Chess, but also introduces unique innovative elements that provide players with a rich range of strategic choices and an immersive gaming experience.

Key features of the game include:

- **Diverse Card System**: Supports a variety of card types, each with unique skills and attributes. Players can upgrade cards to improve their combat abilities.
- **Synergy and Rune Systems**: Expands on synergy effects and rune enhancement, allowing players to leverage lineup synergies and rune selection to maximize their combat effectiveness.
- **Single-player and Multiplayer Modes**: Supports a practice mode where players can battle against AI, as well as a multiplayer mode where players can create or join rooms to compete against others.
- **Audiovisual Experience Optimization**: Features beautifully designed initial and settings menus, as well as background and battle sound effects that enhance immersion.
- **Mini Heroes and Skill Activation**: Players control mini heroes on the board, with each card having red and blue health bars. Blue health fills up over time, enabling skill activation, adding both strategy and interactivity to gameplay.

The goal of this project is to provide players with an Auto Chess game that offers both strategic depth and entertainment. Through rich features and innovative mechanics, it creates a unique Auto Chess world. Whether in single-player practice or multiplayer battles, players will enjoy the challenges and fun of Auto Chess.

Core Features of the Project

1. **Initial and Settings Menus**:
    - Provides an intuitive and attractive initial menu, allowing players to quickly start the game or enter the settings menu.
    - The settings menu supports adjustments for audio, graphics, and key mappings to enhance user experience.
2. **Background Music and Sound Effects**:
    - Dynamic background music that changes based on game scenes to enhance immersion.
    - An option to toggle sound effects, allowing players to customize their audio preferences.
3. **Card System**:
    - Supports a variety of card types, each with unique attributes, skills, and synergy effects.
    - Cards are classified by level and class, and players must strategically combine them to create the strongest lineup.
4. **Card Upgrade Feature**:

- Players can upgrade cards by consuming identical cards or resources, enhancing their attributes and skill effects.
- Upgraded cards will visually change and gain special effects, improving visual appeal.

5. **Mini Hero System**:
   - Players control mini heroes on the board, each with unique skins and animations.
   - The mini hero serves as a representation of the player and adds to the game's interactive fun.
6. **Combat System**:
   - Cards on the board have red and blue health bars; when the blue bar is full, the card can activate its skill. The skill effect varies depending on the card's type and level.
   - Combat is real-time with calculated damage and skill effects, providing a smooth gameplay experience.
7. **Room System**:
   - Supports creating and joining rooms where players can invite friends or other players for battles.
   - In-room chat functionality allows players to communicate strategies or interact with each other.
8. **Practice Mode**:
   - Supports single-player practice mode where players can face off against N ($N \geqslant 2$) AI opponents to familiarize themselves with the game rules and test their lineups.
   - AI difficulty levels are adjustable to suit different player skill levels.
9. **Multiplayer Mode**:
   - Supports multiplayer battles where players can compete with N ($N \geqslant 2$) human opponents in real-time.
   - The multiplayer mode uses real-time synchronization to ensure fair and smooth combat.
10. **Synergy System**:
    - Supports multiple synergy effects, where players can activate synergies by combining specific types of cards to gain attribute bonuses or special effects.
    - The synergy system adds depth to the game’ s strategy, requiring players to carefully plan their lineups to maximize synergy benefits.
11. **Rune Enhancement System**:
    - Offers various rune types that players can select during the game to enhance the power of their cards or lineup.

- Rune effects are diverse, including attribute boosts, skill enhancements, and synergy additions, increasing the game’ s randomness and replayability.

12. **Battle Sound Effects and Visual Effects**:
    - Includes sound effects for skill activation, attacks, and victory/defeat, heightening the intensity and immersion of battles.
    - Skill visual effects are meticulously designed to provide stunning and thematic visuals that align with the card types.

Refactoring with Design Patterns

Refactoring Using Creational Design Patterns

Refactoring with the Builder Pattern

Original Problem

In the pre-refactor code, the initialization design of the `Champion` class has the following issues:

1. **Maintainability Issues**

- **Scattered Constant Definitions**: The constants defined in `CHAMPION_ATTR_MAP` are spread across multiple locations, making maintenance difficult. Every time a new champion is added, changes need to be made in multiple places in the code, which increases the chances of errors.

```
enum ChampionCategory {
    NoChampion,          // 无战斗英雄
    Champion1,           // 劫（一星）
    Champion2,           // 劫（二星）
    Champion3,           // 永恩（一星）
    Champion4,           // 永恩（二星）
    Champion5,           // 奥拉夫（一星）
    ...
};
```

```
typedef struct {
    ChampionCategory championCategory; // 战斗英雄种类
    std::string championName;          // 战斗英雄名称
    std::string championImagePath;     // 战斗英雄图片路径
    int price;                         // 价格
```

```
    int level;                          // 等级
    int healthPoints;                   // 生命值
    int magicPoints;                    // 魔法值
    int attackDamage;                   // 攻击伤害
    int skillTriggerThreshold;          // 技能触发阈值
    int attackRange;                    // 攻击范围
    float attackSpeed;                  // 攻击速度
    float movementSpeed;                // 移动速度
    float defenseCoefficient;           // 防御系数
    Bond bond;                          // 羁绊效果
} ChampionAttributes;
```

```
const std::map<ChampionCategory, ChampionAttributes> CHAMPION_ATTR_MAP = {
    {Champion1, CHAMPION_1_ATTR},   // 劫（一星）
    {Champion2, CHAMPION_2_ATTR},   // 劫（二星）
    {Champion3, CHAMPION_3_ATTR},   // 永恩（一星）
    {Champion4, CHAMPION_4_ATTR},   // 永恩（二星）
    {Champion5, CHAMPION_5_ATTR},   // 奥拉夫（一星）
    ...
};
```

- **Hardcoding:** The attributes of the champions (such as attack power, defense coefficient, etc.) are hardcoded in `CHAMPION_ATTR_MAP`, which reduces the readability and maintainability of the code.

```
const ChampionAttributes CHAMPION_1_ATTR = { // 劫（一星）
    Champion1, u8"劫", "../Resources/Champions/Champion1.png", 1, 1, 500, 0, 50, 200, 1, 0.75f, 1.0f, 1.0f, NoBond
};
const ChampionAttributes CHAMPION_2_ATTR = { // 劫（二星）
    Champion2, u8"劫", "../Resources/Champions/Champion2.png", 3, 2, 900, 0, 75, 200, 1, 0.75f, 1.0f, 1.0f, NoBond
};
const ChampionAttributes CHAMPION_3_ATTR = { // 永恩（一星）
    Champion3, u8"永恩", "../Resources/Champions/Champion3.png", 1, 1, 550, 0, 45, 200, 1, 1.0f, 1.0f, 1.2f, Brotherhood
};
const ChampionAttributes CHAMPION_4_ATTR = { // 永恩（二星）
    Champion4, u8"永恩", "../Resources/Champions/Champion4.png", 3, 2, 990, 0, 68, 200, 1, 1.0f, 1.0f, 1.2f, Brotherhood
};
const ChampionAttributes CHAMPION_5_ATTR = { // 奥拉夫（一星）
```

```
    Champion5, u8"奥拉夫", "../Resources/Champions/Champion5.png", 1, 1, 700, 0, 55, 200, 1, 0.6f, 1.0f, 1.4f, Lout
};
...
```

- In the current code, the constructor of the `Champion` class directly initializes attributes through `CHAMPION_ATTR_MAP`, which makes the code less maintainable and less scalable.

```
Champion::Champion(const ChampionCategory championCategory) :
    currentBattle(nullptr),
    attributes(CHAMPION_ATTR_MAP.at(championCategory)),
    currentLocation({ 0, 0 }),
    currentDestination({ 0, 0 }),
    currentEnemy(nullptr),
    sword(nullptr),
    healthBar(nullptr),
    manaBar(nullptr),
    currentMove(nullptr),
    isMyFlag(false),
    isMoving(false),
    isAttaking(false),
    attackIntervalTimer(0.0f),
    moveIntervalTimer(0.0f)
{
    sprite = Sprite::create(attributes.championImagePath);
    maxHealthPoints = attributes.healthPoints;
    maxMagicPoints = attributes.skillTriggerThreshold;
}
```

2. **Scalability Issues**

- **Difficult to Add New Attributes**: If new attributes (such as critical hit rate, dodge rate, etc.) need to be added for champions in the future, new fields must be added to the `ChampionAttributes` structure, and each hero must have these attributes configured in `CHAMPION_ATTR_MAP`. This can lead to a sharp increase in code size and the risk of missing attributes for some heroes.
- **Attribute redundancy:** certain attributes may only be valid for some heroes, but in the current design, all heroes must have them, leading to redundancy.For example, a ranged hero may not need certain attributes (e.g. attack range) of a melee hero, and vice versa.

3. **High Initialization Complexity**

- **Too Many Constructor Parameters**: The `Champion` class constructor requires a large number of parameters to initialize, which makes the code lengthy and difficult to read. If more attributes are added in the future, the constructor will become even more complex.
- **Default Value Issues**: Some attributes may have default values, but in the current design, all attributes must be explicitly initialized, which leads to redundant code.

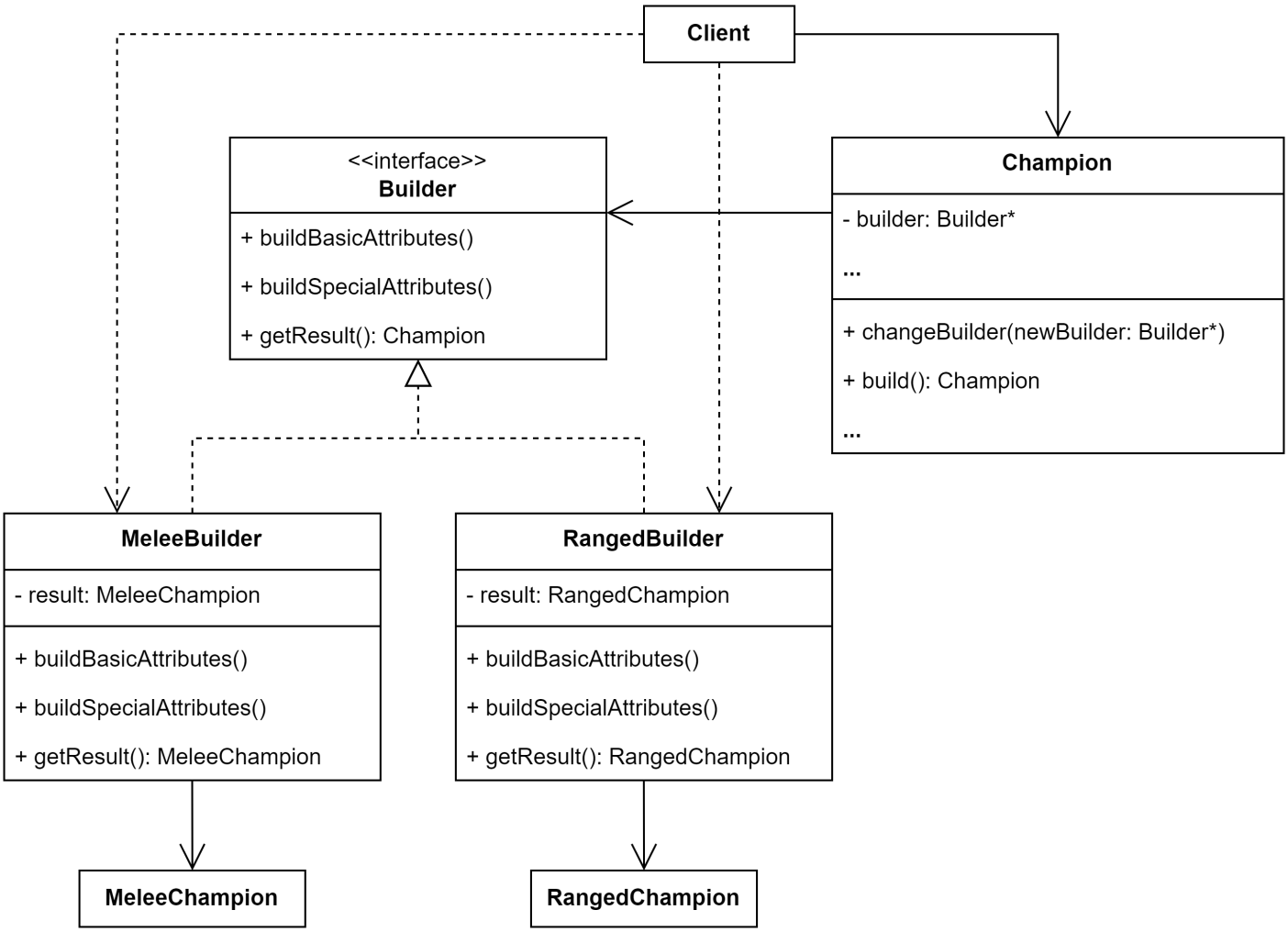
4. **Lack of Flexibility**

- **Fixed Attribute Settings**: In the current design, the hero attributes are fixed and cannot be dynamically adjusted at runtime. If the game needs to adjust hero attributes based on game progress or player choices, the current design will not meet the requirements.

Necessity of Refactoring

In the pre-refactor code, the initialization design of the `Champion` class has multiple issues, including poor maintainability, insufficient scalability, high initialization complexity, and lack of flexibility. Specifically, constants are scattered across multiple locations, making maintenance difficult; hardcoded hero attributes reduce the readability and maintainability of the code; adding new attributes requires changes in multiple places, which is prone to errors; too many constructor parameters make the code long and hard to read; attribute redundancy and default value issues increase code complexity. Additionally, the current design cannot dynamically adjust hero attributes at runtime, lacking flexibility. To address these issues, refactoring using the **Builder Pattern** is necessary. The Builder Pattern separates the construction process of an object from its representation, making the code more modular, maintainable, and scalable.

UML Class Diagram



Builder Pattern UML Class Diagram

Refactoring Steps

To solve the above issues, we will refactor the `Champion` class using the **Builder Pattern**. The specific steps for refactoring are as follows:

1. Defining the `Builder` Interface

We define a `Builder` interface that contains three purely virtual functions: `buildBasicAttributes`, `buildSpecialAttributes`, and `getResult`. `buildBasicAttributes` is used to build the hero's basic attributes, such as life value and attack power; `buildSpecialAttributes` is used to build the special attributes of the hero, such as attack range and defence coefficient; `getResult` returns the completed `Champion` object.This interface provides a unified build process for specific generator classes, ensuring that all generators follow the same build steps.

```
class Builder {
public:
    virtual ~Builder() = default;
    virtual void buildBasicAttributes() = 0;
    virtual void buildSpecialAttributes() = 0;
```

```
    virtual Champion getResult() = 0;
};
```

2. Implementation of Specific Generator Classes

We have implemented two specific generator classes: `MeleeBuilder` and `RangedBuilder`. The `MeleeBuilder` is used to construct melee heroes, while the `RangedBuilder` is used to construct ranged heroes. They possess different base attributes and special attributes.

```
class MeleeBuilder : public Builder {
private:
    ChampionAttributes attributes;

public:
    void buildBasicAttributes() override {}
    void buildSpecialAttributes() override {}
    Champion getResult() override {}
};
```

```
class RangedBuilder : public Builder {
private:
    ChampionAttributes attributes;

public:
    void buildBasicAttributes() override {}
    void buildSpecialAttributes() override {}
    Champion getResult() override {}
};
```

3. Refactoring the `Champion` class

In the `Champion` class, we added a `Builder` pointer and provided `changeBuilder` and `build` methods. The `changeBuilder` method is used to dynamically switch generators, allowing us to choose to use `MeleeBuilder` or `RangedBuilder` at runtime. The `build` method calls the `buildBasicAttributes` and `buildSpecialAttributes` methods of the generator to complete the build of the hero and return the build result.

```
void Champion::changeBuilder(Builder* newBuilder) {
    if (builder) {
        delete builder;
    }
```

```
    builder = newBuilder;
}

Champion Champion::build() {
    if (builder) {
        builder->buildBasicAttributes();
        builder->buildSpecialAttributes();
        return builder->getResult();
    }
    throw std::runtime_error("No builder set!");
}
```

4. Use the generator mode to create melee heroes and ranged heroes

Next, we will create melee heroes and ranged heroes through the generator mode.

```
MeleeBuilder meleeBuilder;
Champion champion1;
champion1.changeBuilder(&meleeBuilder);
Champion meleeChampion = champion1.build();

RangedBuilder rangedBuilder;
Champion champion2;
champion2.changeBuilder(&rangedBuilder);
Champion rangedChampion = champion2.build();
```

Improvements Achieved

1. **Clear distinction between melee and ranged heroes**
   - By introducing `MeleeBuilder` and `RangedBuilder` , we separated the building logic of melee and ranged heroes into different generator classes. Each generator class focuses on building specific types of heroes, making the code structure clearer.
2. **Improve code maintainability**

   The generator pattern breaks down the hero building process into multiple steps (such as `buildBasicAttributes` and `buildSpecialAttributes`), and centralizes the setting of properties in the generator class. When adding or modifying properties, only adjustments need to be made in the corresponding generator class, without modifying the global `CHAMPION_ATTR_MAP` .
3. **Scalability Enhancement**

   The generator pattern abstracts the construction process of properties into interface methods (such as `buildBasicAttributes` and `buildSpecialAttributes`). If new

properties need to be added in the future, only the corresponding construction steps need to be added in the generator class, without modifying the constructor function or other related code of the `Champion` class.

4. **Reduce initialization complexity**

   The generator pattern breaks down the complex initialization process into several simple steps (such as `buildBasicAttributes` and `buildSpecialAttributes`) and returns the built `Champion` object through the `getResult` method. This makes the initialization logic more modular and easy to understand.

5. **Increased flexibility**

   - Generator mode allows us to dynamically switch generators at runtime (via the `changeBuilder` method), thus flexibly building different types of heroes. For example, the attributes of heroes can be dynamically adjusted based on game progress or player choices.

6. **Reduce attribute redundancy**

   - Generator mode allows us to dynamically set attributes based on hero type (melee or ranged), avoiding unnecessary attribute redundancy. For example, `MeleeBuilder` only sets the attributes required by melee heroes, while `RangedBuilder` only sets the attributes required by ranged heroes.

Refactoring with the Singleton Pattern

Original Issues

Before refactoring, the `LocationMap` class had the following issues:

1. Multiple Instances Problem

The `LocationMap` class can be instantiated multiple times, leading to multiple `LocationMap` objects in the system. Each object contains the same key-value pairs for location attributes and screen coordinates. This not only wastes memory resources but also potentially causes data inconsistency issues.

```
1 LocationMap::LocationMap() {
2     initializeLocationMap();
3 }
```

2. Global Access Problem

Since the `LocationMap` class can be instantiated multiple times, other modules need to explicitly know which instance of `LocationMap` to use. This increases the complexity and

coupling of the code.

```
const std::map<Location, cocos2d::Vec2>& LocationMap::getLocationMap() const {
    return locationMap;
}
```

3. Initialization Problem

Each time a `LocationMap` object is created, the `initializeLocationMap` method is called for initialization. This leads to redundant initialization operations, adding unnecessary performance overhead.

```
void LocationMap::initializeLocationMap() {
    locationMap.clear(); // 清空现有的Map

    // 初始化等待区域的位置
    for (int i = 0; i < WAITING_MAP_COUNT; i++) {
        const Location location = { WaitingArea, i };
        locationMap[location] = cocos2d::Vec2(WAITING_AREA_START_X + i * CHAMPION_HORIZONTAL_INTERVAL, WAITING_AREA_START_Y);
    }

    // 初始化战斗区域的位置
    for (int i = 0; i < BATTLE_MAP_ROWS; i++) {
        for (int j = 0; j < BATTLE_MAP_COLUMNS; j++) {
            const Location location = { BattleArea, i * BATTLE_MAP_COLUMNS + j };
            locationMap[location] = cocos2d::Vec2(BATTLE_AREA_START_X + j * CHAMPION_HORIZONTAL_INTERVAL - ((i % 2 == 0) ? 0 : CHAMPION_HORIZONTAL_INTERVAL / 2), BATTLE_AREA_START_Y + i * CHAMPION_VERTICAL_INTERVAL);
        }
    }
}
```

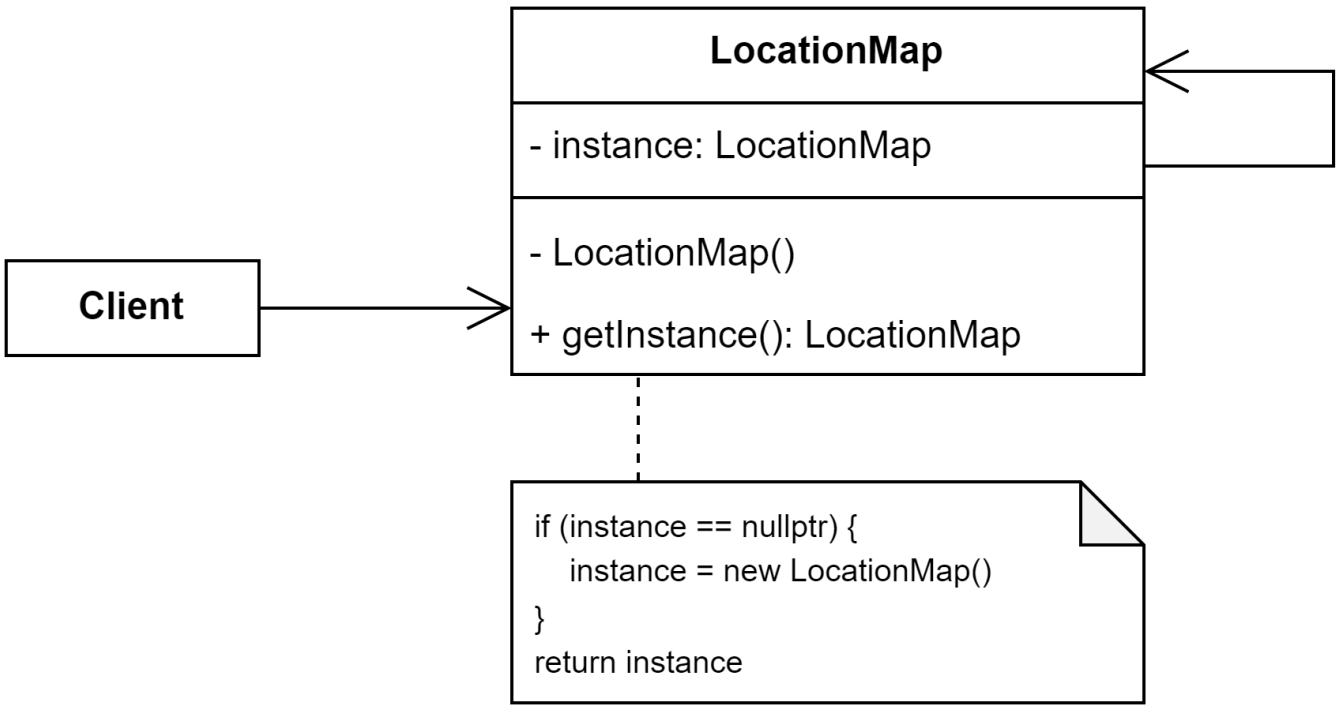
Necessity of Refactoring

Before refactoring, the `LocationMap` class had issues such as multiple instances, global access problems, and initialization issues. Specifically, the `LocationMap` class could be instantiated multiple times, resulting in multiple objects in the system, wasting memory resources and potentially causing data inconsistency. Other modules had to explicitly know which instance of `LocationMap` to use, increasing the complexity and coupling of the code.

Additionally, the `initializeLocationMap` method was called repeatedly every time a `LocationMap` object was created, adding unnecessary performance overhead.

To address these issues, refactoring with the Singleton Pattern is essential. The Singleton Pattern ensures that a class has only one instance and provides a global access point, thus avoiding the multiple instances problem, simplifying global access, and optimizing the initialization logic.

UML Class Diagram



Singleton Pattern UML Class Diagram

Refactoring Steps

To address the issues mentioned above, we will refactor the `LocationMap` class using the Singleton Pattern. The Singleton Pattern ensures that a class has only one instance and provides a global access point. Here are the specific steps for the refactoring:

1. Private Constructor

Make the `LocationMap` constructor private to prevent external code from directly creating instances of the `LocationMap` class.

```
class LocationMap {
public:
    ...

private:
    std::map<Location, cocos2d::Vec2> locationMap; // 位置属性与屏幕坐标键值对
```

```
    // 构造函数
    LocationMap();

    ...
};
```

2. Delete Copy Constructor and Assignment Operator

To prevent the creation of new instances through the copy constructor or assignment operator, we will delete both of these operators.

```
class LocationMap {
public:
    ...

private:
    ...

    // 禁止拷贝构造函数和赋值操作符
    LocationMap(const LocationMap&) = delete;
    LocationMap& operator=(const LocationMap&) = delete;
};
```

3. Provide a Static Method to Get the Instance

Add a static method `getInstance` to retrieve the unique instance of `LocationMap`. This method will use a static local variable to ensure that only one instance is created.

```
class LocationMap {
public:
    // 获取单例
    static LocationMap& getInstance();

    // 获取位置属性与屏幕坐标键值对
    const std::map<Location, cocos2d::Vec2>& getLocationMap() const;

private:
    ...
};
```

4. Simplify Initialization Logic

Move the logic of the `initializeLocationMap` method directly into the constructor, as the Singleton Pattern ensures that the constructor will only be called once.

```
LocationMap::LocationMap()
{
    for (int i = 0; i < WAITING_MAP_COUNT; i++) {
        const Location location = { WaitingArea, i };
        locationMap[location] = cocos2d::Vec2(WAITING_AREA_START_X + i * CHAMPION_HORIZONTAL_INTERVAL, WAITING_AREA_START_Y);
    }
    for (int i = 0; i < BATTLE_MAP_ROWS; i++) {
        for (int j = 0; j < BATTLE_MAP_COLUMNS; j++) {
            const Location location = { BattleArea, i * BATTLE_MAP_COLUMNS + j };
            locationMap[location] = cocos2d::Vec2(BATTLE_AREA_START_X + j * CHAMPION_HORIZONTAL_INTERVAL - ((i % 2 == 0) ? 0 : CHAMPION_HORIZONTAL_INTERVAL / 2), BATTLE_AREA_START_Y + i * CHAMPION_VERTICAL_INTERVAL);
        }
    }
}
```

5. Remove Unnecessary Member Functions

Since the Singleton Pattern ensures there is only one instance, the `initializeLocationMap` method is no longer needed. Initialization can be directly handled within the constructor.

Improvements Achieved

1. **Single Instance Management**
   - The Singleton Pattern ensures that the `LocationMap` class has only one instance, eliminating the multi-instance issue, saving memory resources, and preventing data inconsistencies.
2. **Simplified Global Access**
   - The static method `getInstance` allows other modules to easily access the unique instance of `LocationMap`, reducing code complexity and coupling.
3. **Initialization Optimization**
   - The initialization logic has been moved directly into the constructor, leveraging the Singleton Pattern's guarantee that the constructor will only be called once. This avoids redundant initialization operations and improves performance.
4. **Prevention of Illegal Instantiation**

- The constructor has been made private, and the copy constructor and assignment operator have been deleted to prevent external code from creating new instances via copying or assignment, enhancing the security of the code.

5. **Code Simplification**
   - The unnecessary `initializeLocationMap` method has been removed, simplifying the class structure and making the code clearer and easier to maintain.
6. **Enhanced Testability**
   - The Singleton Pattern makes the behavior of `LocationMap` more predictable, making it easier to test and debug.

Refactoring with Structural Patterns

Refactoring with the Composite Pattern

Composite Pattern

In the original code, the `ChampionAttributesLayer` class directly creates and manages multiple `Label` objects to display hero attributes. The problems with this approach are as follows:

1. Code Duplication

Each time a label is created, the `createLabel` method must be called, which leads to repetitive code that is difficult to maintain.

```
1 void ChampionAttributesLayer::createLabel(const std::string& text, const float
  x, const float y, const int fontSize, const cocos2d::Color4B color)
2 {
3     auto label = Label::createWithTTF(text,
  "../Resources/Fonts/DingDingJinBuTi.ttf", fontSize);
4     label->setAnchorPoint(Vec2(0, 0.5));
5     label->setPosition(x, y);
6     label->setTextColor(color);
7     this->addChild(label);
8 }
```

2. Lack of Flexibility

If new types of nodes (e.g., `Sprite`, `Button`) need to be added in the future, a significant amount of code modification will be required.

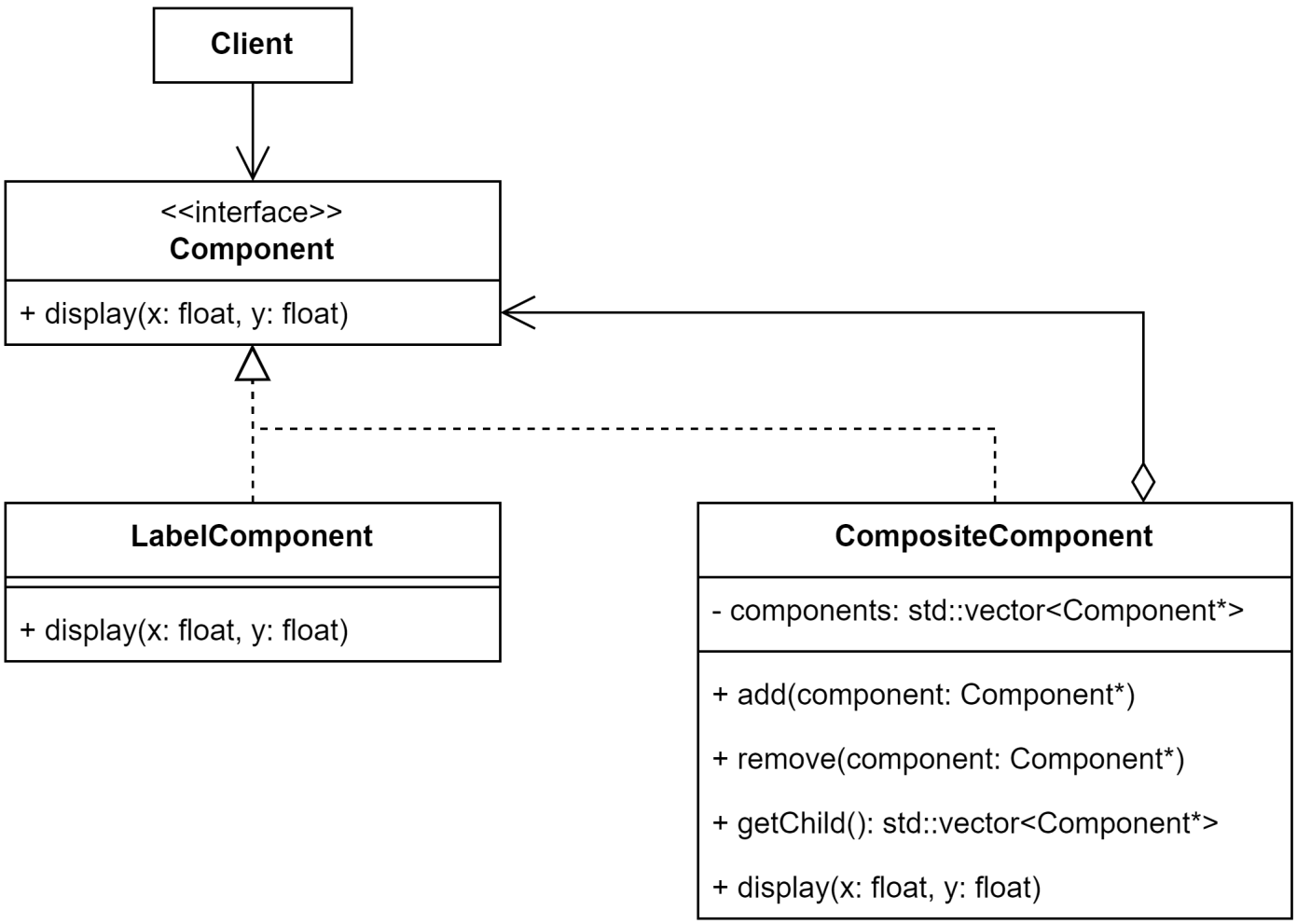
3. Tight Coupling

The `ChampionAttributesLayer` class is tightly coupled with the `Label` class, making it difficult to extend and reuse.

Necessity of Refactoring

In the original code, the `ChampionAttributesLayer` class directly manages multiple `Label` objects to display hero attributes, leading to code duplication, lack of flexibility, and tight coupling with the `Label` class. Specifically, every time a label is created, the `createLabel` method is called, resulting in repetitive and difficult-to-maintain code. Additionally, if new types of nodes (such as `Sprite`, `Button`, etc.) need to be added in the future, the code would require significant changes. The tight coupling between `ChampionAttributesLayer` and `Label` makes the system hard to extend and reuse. To address these issues, using the **Composite Pattern** is essential. The Composite Pattern organizes objects into tree structures, allowing the client to treat both individual objects and compositions uniformly. This increases flexibility, scalability, and maintainability in the code.

UML Class Diagram



Composite Pattern UML Class Diagram

Refactoring Steps

By using the Composite Pattern, we can decouple the `ChampionAttributesLayer` class from specific node types (like `Label`) and make the code more flexible and extensible. In the future, if we need to add new types of nodes (e.g., `Sprite`, `Button`), we can simply create new leaf node classes that implement the `Component` interface, without modifying the `ChampionAttributesLayer` class. Below are the steps to refactor the `ChampionAttributesLayer` class using the Composite Pattern:

1. Define a Component Interface

All node types (like `Label`, `Sprite`, etc.) will implement this interface.

```
class Component {
public:
    virtual ~Component() {}
    virtual void add(Component* component) {}
    virtual void remove(Component* component) {}
    virtual void display(float x, float y) = 0;
};
```

2. Create Leaf Nodes

Leaf nodes are nodes that do not have any child nodes, such as `Label`.

```
class LabelComponent : public Component {
public:
    static LabelComponent* create(const std::string& text, int fontSize,
cocos2d::Color4B color);
    virtual void display(float x, float y) override;

private:
    LabelComponent(const std::string& text, int fontSize, cocos2d::Color4B
color);
    cocos2d::Label* label;
};
```

```
LabelComponent::LabelComponent(const std::string& text, int fontSize,
cocos2d::Color4B color) {
    label = cocos2d::Label::createWithTTF(text,
"../Resources/Fonts/DingDingJinBuTi.ttf", fontSize);
    label->setTextColor(color);
}

```

```
LabelComponent* LabelComponent::create(const std::string& text, int fontSize, cocos2d::Color4B color) {
    return new LabelComponent(text, fontSize, color);
}

void LabelComponent::display(float x, float y) {
    label->setPosition(x, y);
    label->setAnchorPoint(cocos2d::Vec2(0, 0.5));
    cocos2d::Director::getInstance()->getRunningScene()->addChild(label);
}
```

3. Create Composite Nodes

Composite nodes can contain other nodes (leaf nodes or other composite nodes).

```
class CompositeComponent : public Component {
public:
    virtual void add(Component* component) override;
    virtual void remove(Component* component) override;
    virtual void display(float x, float y) override;
    std::vector<Component*> getChild() const;

private:
    std::vector<Component*> components;
};
```

```
void CompositeComponent::add(Component* component) {
    components.push_back(component);
}

void CompositeComponent::remove(Component* component) {
    components.erase(std::remove(components.begin(), components.end(), component), components.end());
}

void CompositeComponent::display(float x, float y) {
    for (auto& component : components) {
        component->display(x, y);
    }
}

std::vector<Component*> CompositeComponent::getChild() const {
    return components;
}
```

4. Refactor the ChampionAttributesLayer Class

Use the composite pattern to manage the nodes.

```
class ChampionAttributesLayer : public cocos2d::Layer {
public:
    virtual bool init();
    void showAttributes(const ChampionCategory championCategory);
    CREATE_FUNC(ChampionAttributesLayer);

private:
    void createLabel(const std::string& text, float x, float y, int fontSize = CHAMPION_ATTRIBUTES_FONT_SIZE, cocos2d::Color4B color = cocos2d::Color4B::WHITE);
    std::string formatFloat(float value, int precision = 2);
    CompositeComponent* rootComponent;
};
```

```
// 初始化战斗英雄属性层
bool ChampionAttributesLayer::init() {
    if (!Layer::init()) {
        return false;
    }

    rootComponent = new CompositeComponent();

    auto backgroundImage = cocos2d::Sprite::create("../Resources/ImageElements/ChampionAttributesLayerBackground.png");
    backgroundImage->setPosition(BACKGROUND_IMAGE_START_X, BACKGROUND_IMAGE_START_Y);
    backgroundImage->setOpacity(BACKGROUND_IMAGE_TRANSPARENCY);
    this->addChild(backgroundImage);

    return true;
}

// 显示战斗英雄属性
void ChampionAttributesLayer::showAttributes(const ChampionCategory championCategory) {
    auto championAttributes = CHAMPION_ATTR_MAP.at(championCategory);

```

```
21      int championNumber = championAttributes.championCategory % 2 == 1 ? championAttributes.championCategory : championAttributes.championCategory - 1;
22      std::string championImagePath = "../Resources/Champions/Champion" + std::to_string(championNumber) + "&" + std::to_string(championNumber + 1) + ".png";
23      auto championImage = cocos2d::Sprite::create(championImagePath);
24      championImage->setPosition(CHAMPION_IMAGE_START_X, CHAMPION_IMAGE_START_Y);
25      this->addChild(championImage);
26
27      createLabel(championAttributes.championName, CHAMPION_NAME_LABEL_START_X, CHAMPION_NAME_LABEL_START_Y, CHAMPION_NAME_LABEL_FONT_SIZE, championAttributes.championCategory % 2 == 1 ? cocos2d::Color4B::WHITE : cocos2d::Color4B({ GOLDEN_R, GOLDEN_G, GOLDEN_B }));
28      createLabel(std::to_string(championAttributes.level), LEVEL_LABEL_START_X, LEVEL_LABEL_START_Y);
29      createLabel(std::to_string(championAttributes.healthPoints), FIRST_COLUMN_START_X, HEALTH_POINTS_LEBEL_START_Y);
30      createLabel(std::to_string(championAttributes.attackDamage), FIRST_COLUMN_START_X, ATTACK_DAMAGE_LABEL_START_Y);
31      createLabel(std::to_string(championAttributes.attackRange), FIRST_COLUMN_START_X, ATTACK_RANGE_LABEL_START_Y);
32      createLabel(formatFloat(championAttributes.attackSpeed), FIRST_COLUMN_START_X, ATTACK_SPEED_LABEL_START_Y);
33      createLabel(std::to_string(championAttributes.price), SECOND_COLUMN_START_X, PRICE_LABEL_START_Y);
34      createLabel(formatFloat(championAttributes.movementSpeed), SECOND_COLUMN_START_X, MOVEMENT_SPEED_START_Y);
35      createLabel(formatFloat(championAttributes.defenseCoefficient), SECOND_COLUMN_START_X, DEFENSE_COEFFICIENT_START_Y);
36      createLabel(std::to_string(championAttributes.skillTriggerThreshold), SECOND_COLUMN_START_X, SKILL_TRIGGER_THRESHOLD_START_Y);
37
38      rootComponent->display(0, 0);
39  }
40
41  // 创建属性标签
42  void ChampionAttributesLayer::createLabel(const std::string& text, float x, float y, int fontSize, cocos2d::Color4B color) {
43      auto labelComponent = LabelComponent::create(text, fontSize, color);
44      rootComponent->add(labelComponent);
45      labelComponent->display(x, y);
46  }
47
48  // 浮点数格式化输出
49  std::string ChampionAttributesLayer::formatFloat(const float value, const int precision)
50  {
```

```
    std::ostringstream oss;
    oss << std::fixed << std::setprecision(precision) << value;
    return oss.str();
}
```

Improvements Achieved

1. **Decoupling and Abstraction**
   - By defining the `Component` interface, the `ChampionAttributesLayer` class is decoupled from specific node types (such as `Label`, `Sprite`, etc.), making the code more flexible and extensible.
2. **Reduced Code Duplication**
   - After applying the composite pattern, the logic for creating and managing nodes is encapsulated in the `Component` interface and its implementation classes, eliminating the need to repeatedly call the `createLabel` method and reducing code redundancy.
3. **Support for Multiple Node Types**
   - The composite pattern allows for the easy addition of new node types (such as `SpriteComponent`, `ButtonComponent`, etc.). You only need to implement the `Component` interface without modifying the `ChampionAttributesLayer` class.
4. **Unified Node Management**
   - The `CompositeComponent` class now manages all child nodes (including both leaf nodes and other composite nodes), simplifying the addition, deletion, and display of nodes.
5. **Enhanced Extensibility**
   - If new node types need to be added or existing node behaviors need to be modified in the future, you can extend the `Component` interface and its implementations without changing the code of the `ChampionAttributesLayer` class.
6. **Improved Code Readability**
   - The composite pattern breaks down complex node management logic into multiple simple classes (such as `LabelComponent`, `CompositeComponent`, etc.), making the code structure clearer and easier to understand and maintain.
7. **Support for Dynamic Structures**
   - The composite pattern allows dynamic addition or removal of nodes at runtime, making the `ChampionAttributesLayer` class adaptable to varying requirements.
8. **Simplified Client Code**

- The `ChampionAttributesLayer` class only interacts with the `Component` interface and does not need to be concerned with specific node types or implementation details, simplifying client code.

Refactoring with the Facade Pattern

Original Issues

In the original code, the audio playback logic is directly exposed through global functions and global variables, which presents the following problems:

1. High Coupling

External code directly depends on `AudioEngine` and global variables, leading to high coupling, making the code difficult to maintain and extend.

2. Code Duplication

If audio playback is needed in multiple places, similar code (such as stopping current music, adjusting volume, etc.) might be repeated.

3. Difficult to Extend

If new features (like pause, resume, volume control, etc.) need to be added, the code must be modified in multiple places.

```
1  void audioPlayer(const std::string& audioPath, bool isLoop)
2  {
3      if (isLoop) {
4          if (g_backgroundMusicSign != DEFAULT_MUSIC_SIGN) {
5              cocos2d::experimental::AudioEngine::stop(g_backgroundMusicSign);
6          }
7          g_backgroundMusicSign =
   cocos2d::experimental::AudioEngine::play2d(audioPath, isLoop);
8          cocos2d::experimental::AudioEngine::setVolume(g_backgroundMusicSign,
   g_backgroundMusicVolumn);
9      }
10     else {
11         g_soundEffectSign =
   cocos2d::experimental::AudioEngine::play2d(audioPath, isLoop);
12         cocos2d::experimental::AudioEngine::setVolume(g_soundEffectSign,
   g_soundEffectVolumn);
13     }
14 }
```

4. Global Variable Risks

Using global variables to store audio state (e.g., `g_backgroundMusicSign`) can lead to unintended modifications or conflicts.

```
int g_backgroundMusicSign = DEFAULT_MUSIC_SIGN;
int g_soundEffectSign = DEFAULT_MUSIC_SIGN;
float g_backgroundMusicVolumn = DEFAULT_MUSIC_VOLUMN;
float g_soundEffectVolumn = DEFAULT_MUSIC_VOLUMN;
```

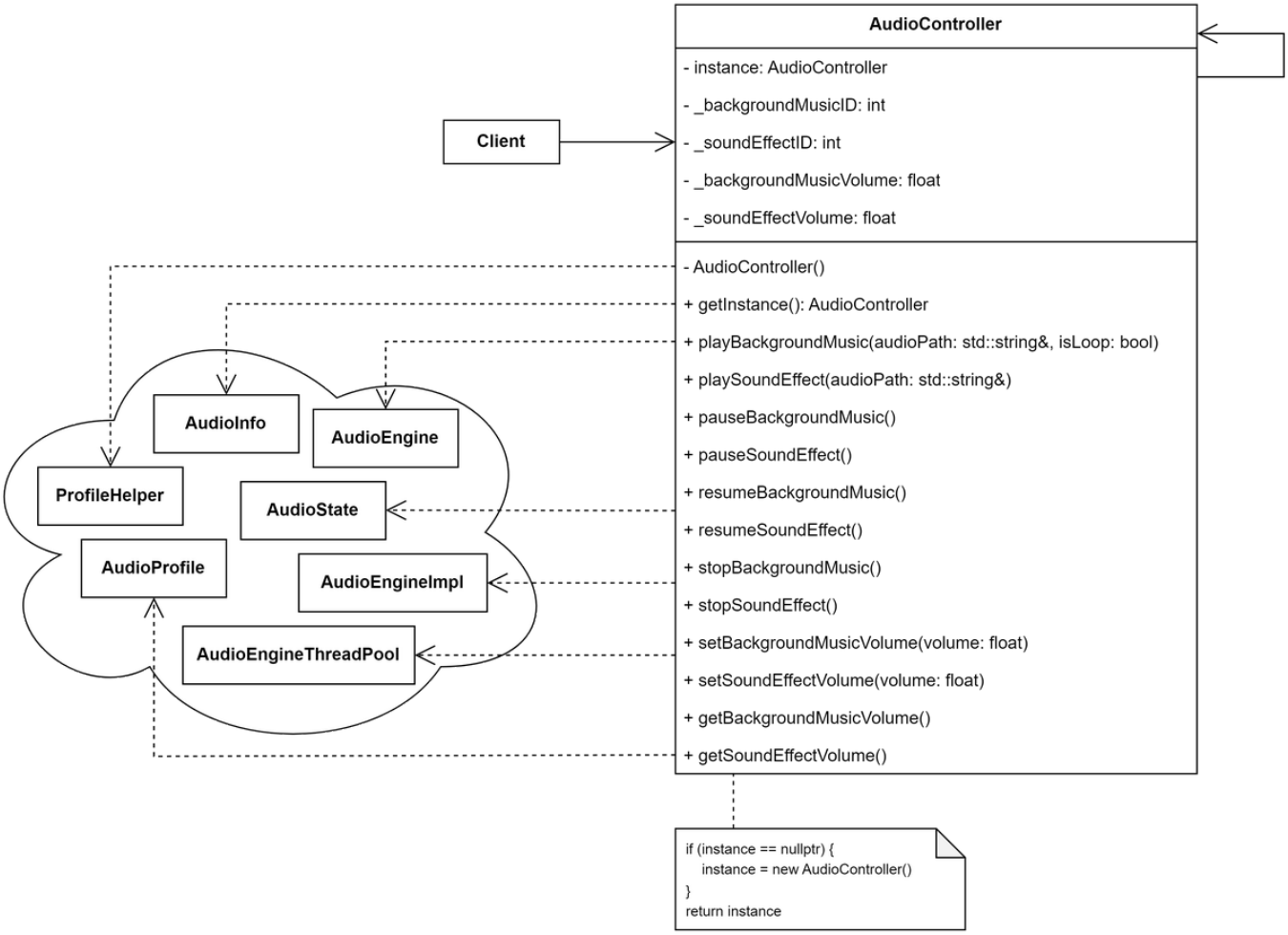
5. Lack of Encapsulation

The audio playback logic is scattered across global functions, lacking a unified interface, which makes the code harder to organize and reuse.

Refactoring Necessity

In the original code, the audio playback logic is exposed through global functions and variables, leading to issues such as high coupling, code repetition, difficulty in extending functionality, global variable risks, and lack of encapsulation. Specifically, external code directly depends on the `AudioEngine` and global variables, making the code tightly coupled and hard to maintain and extend; when audio needs to be played in multiple places, similar code is often repeated; adding new features (such as pause, resume, volume control, etc.) requires modifications in several places; the use of global variables can lead to unintended modifications or conflicts; and the audio logic is scattered across global functions, making it harder to organize and reuse. To address these issues, refactoring with the **Facade Pattern** is necessary. The facade pattern encapsulates the complexities of a subsystem behind a unified interface, which reduces coupling and improves maintainability and scalability.

UML Class Diagram



Facade Pattern UML Class Diagram

Refactoring Steps

To solve the above problems, we can encapsulate the audio playback logic using the facade pattern, providing a unified interface for external calls. The specific steps are as follows:

1. Analyze the functionality of the original code

- The original code includes:
  - Playing background music (looping)
  - Playing sound effects (non-looping)
  - Controlling volume
  - Stopping the current music
- Features to extend:
  - Pausing and resuming audio
  - Retrieving current volume
  - More flexible volume control

2. Design the Facade Class (AudioController) and define its interface

- Encapsulate the audio playback logic within a single class.
- Provide a unified interface that hides the complexity of `AudioEngine`.
- Use the singleton pattern to ensure there is only one instance of `AudioController` globally.
- Define the facade class interface with the following methods:
  - Play background music
  - Play sound effects
  - Pause, resume, and stop audio
  - Set and get volume

```
class AudioController {
public:
    // 单例模式
    static AudioController* getInstance();

    // 播放音频
    void playBackgroundMusic(const std::string& audioPath, bool isLoop = true);
    void playSoundEffect(const std::string& audioPath);

    // 暂停音频
    void pauseBackgroundMusic();
    void pauseSoundEffect();

    // 恢复音频
    void resumeBackgroundMusic();
    void resumeSoundEffect();

    // 停止音频
    void stopBackgroundMusic();
    void stopSoundEffect();

    // 设置音量
    void setBackgroundMusicVolume(float volume);
    void setSoundEffectVolume(float volume);

    // 获取音量
    float getBackgroundMusicVolume() const;
    float getSoundEffectVolume() const;

private:
    // 私有构造函数（单例模式）
    AudioController();
```

```
    ~AudioController();

    // 禁用拷贝构造函数和赋值运算符
    AudioController(const AudioController&) = delete;
    AudioController& operator=(const AudioController&) = delete;

    // 背景音乐和音效的ID
    int _backgroundMusicID;
    int _soundEffectID;

    // 音量
    float _backgroundMusicVolume;
    float _soundEffectVolume;
};
```

3. Implement the Facade Class

- Implement the above interface in the `AudioController` class.
- Use member variables to store audio states (such as the current playing audio ID, volume, etc.).
- Encapsulate the calls to the underlying `AudioEngine`.

```
// 单例实例
AudioController* AudioController::_instance = nullptr;

AudioController* AudioController::getInstance() {
    if (!_instance) {
        _instance = new AudioController();
    }
    return _instance;
}

AudioController::AudioController()
    : _backgroundMusicID(-1), _soundEffectID(-1),
      _backgroundMusicVolume(1.0f), _soundEffectVolume(1.0f) {}

AudioController::~AudioController() {
    // 释放资源
    stopBackgroundMusic();
    stopSoundEffect();
    delete _instance;
}

// 播放背景音乐
```

```cpp
void AudioController::playBackgroundMusic(const std::string& audioPath, bool isLoop) {
    if (_backgroundMusicID != -1) {
        cocos2d::experimental::AudioEngine::stop(_backgroundMusicID);
    }
    _backgroundMusicID = cocos2d::experimental::AudioEngine::play2d(audioPath, isLoop);
    cocos2d::experimental::AudioEngine::setVolume(_backgroundMusicID, _backgroundMusicVolume);
}

// 播放音效
void AudioController::playSoundEffect(const std::string& audioPath) {
    _soundEffectID = cocos2d::experimental::AudioEngine::play2d(audioPath, false);
    cocos2d::experimental::AudioEngine::setVolume(_soundEffectID, _soundEffectVolume);
}

// 暂停背景音乐
void AudioController::pauseBackgroundMusic() {
    if (_backgroundMusicID != -1) {
        cocos2d::experimental::AudioEngine::pause(_backgroundMusicID);
    }
}

// 暂停音效
void AudioController::pauseSoundEffect() {
    if (_soundEffectID != -1) {
        cocos2d::experimental::AudioEngine::pause(_soundEffectID);
    }
}

// 恢复背景音乐
void AudioController::resumeBackgroundMusic() {
    if (_backgroundMusicID != -1) {
        cocos2d::experimental::AudioEngine::resume(_backgroundMusicID);
    }
}

// 恢复音效
void AudioController::resumeSoundEffect() {
    if (_soundEffectID != -1) {
        cocos2d::experimental::AudioEngine::resume(_soundEffectID);
    }
}

```

```cpp
// 停止背景音乐
void AudioController::stopBackgroundMusic() {
    if (_backgroundMusicID != -1) {
        cocos2d::experimental::AudioEngine::stop(_backgroundMusicID);
        _backgroundMusicID = -1;
    }
}

// 停止音效
void AudioController::stopSoundEffect() {
    if (_soundEffectID != -1) {
        cocos2d::experimental::AudioEngine::stop(_soundEffectID);
        _soundEffectID = -1;
    }
}

// 设置背景音乐音量
void AudioController::setBackgroundMusicVolume(float volume) {
    _backgroundMusicVolume = volume;
    if (_backgroundMusicID != -1) {
        cocos2d::experimental::AudioEngine::setVolume(_backgroundMusicID, volume);
    }
}

// 设置音效音量
void AudioController::setSoundEffectVolume(float volume) {
    _soundEffectVolume = volume;
    if (_soundEffectID != -1) {
        cocos2d::experimental::AudioEngine::setVolume(_soundEffectID, volume);
    }
}

// 获取背景音乐音量
float AudioController::getBackgroundMusicVolume() const {
    return _backgroundMusicVolume;
}

// 获取音效音量
float AudioController::getSoundEffectVolume() const {
    return _soundEffectVolume;
}
```

4. Replace the Original Code

- Replace the parts of the original code that directly call `AudioEngine` with calls to the `AudioController` interface.
- Remove global variables and global functions.

```
#include "AudioController.h"

// 播放背景音乐
AudioController::getInstance()->playBackgroundMusic("../Resources/Music/PreparationScene_RagsToRings.mp3");

// 播放音效
AudioController::getInstance()->playSoundEffect("../Resources/Music/SkillSoundEffect.mp3");

// 设置背景音乐音量
AudioController::getInstance()->setBackgroundMusicVolume(0.5f);

// 暂停背景音乐
AudioController::getInstance()->pauseBackgroundMusic();

// 停止音效
AudioController::getInstance()->stopSoundEffect();
```

Improvements Achieved

1. **Reduced Coupling**
   - By encapsulating the audio playback logic in the `AudioController` class, external code only interacts with `AudioController` and no longer directly depends on `AudioEngine`, reducing the coupling of the code.
2. **Reduced Code Duplication**
   - By centralizing the audio playback logic in the `AudioController` class, it avoids the repetition of similar code (such as stopping the current music, adjusting the volume, etc.) in multiple places.
3. **Enhanced Extensibility**
   - New functionalities (such as pause, resume, volume control) can be added by simply adding corresponding methods to the `AudioController` class without modifying external code, increasing the extensibility of the code.
4. **Eliminated Global Variable Risks**

- The `AudioController` class stores audio states (such as the currently playing audio ID, volume, etc.) in its member variables, avoiding the use of global variables and reducing the risk of unintended modifications or conflicts.

5. **Unified Interface Provided**
   - The `AudioController` class provides a unified interface, hiding the complexity of the underlying `AudioEngine`, making the code more modular and easier to maintain.
6. **Supports Singleton Pattern**
   - The singleton pattern ensures that only one instance of `AudioController` exists globally, preventing issues related to inconsistent states across multiple instances.
7. **Simplified Client Code**
   - External code only needs to call the `AudioController` interface to perform audio playback, pause, resume, and stop actions, which simplifies the client code.
8. **Improved Code Readability**
   - By encapsulating the complex audio playback logic within the `AudioController` class, the code structure becomes clearer, easier to understand, and maintain.

Refactoring with Behavioral Patterns

Using Strategy Pattern for Refactoring

Original Issue

The strategy pattern is a behavioral design pattern that defines a series of algorithms (strategies) and encapsulates each algorithm in a separate class, making them interchangeable. Each skill is an independent algorithm, and the strategy pattern can encapsulate the algorithm in separate classes to avoid conditional branches. The `skill` method in the `Champion` class has the following issues:

1. Too Many Conditional Branches

The `skill` function contains a large number of if-else conditional branches to determine the effects of different heroes' skills. Each conditional branch handles a different skill logic, resulting in lengthy and hard-to-maintain code.

```
1 void Champion::skill()
2 {
3     if (attributes.price == CHAMPION_ATTR_MAP.at(FIFTH_LEVEL[1]).price) {
4         triggerSkill(ATTACK_DAMAGE_MAGNIFICATION_HIGH);
5     }
6     else if (attributes.price == CHAMPION_ATTR_MAP.at(FOURTH_LEVEL[1]).price) {
```

```
        if (attributes.championCategory == Champion25 ||
attributes.championCategory == Champion26) {
            attributes.attackDamage += SKILL_ATTACK_DAMAGE_UP;
            attributes.defenseCoefficient += SKILL_DEFENSE_COEFFICIENT_UP;
        }
        else {
            triggerSkill(ATTACK_DAMAGE_MAGNIFICATION_MIDDLE, false);
        }
    }
    else if (attributes.price == CHAMPION_ATTR_MAP.at(SECOND_LEVEL[1]).price
|| attributes.price == CHAMPION_ATTR_MAP.at(THIRD_LEVEL[1]).price) {
        if (attributes.attackRange > ATTACK_RANGE_THRESHOLD) {
            attributes.attackDamage += SKILL_ATTACK_DAMAGE_UP;
            attributes.attackSpeed += SKILL_ATTACK_SPEED_UP;
        }
        else if (attributes.defenseCoefficient >=
DEFENSE_COEFFICIENT_THRESHOLD_HIGH) {
            attributes.defenseCoefficient += SKILL_DEFENSE_COEFFICIENT_UP;
        }
        else {
            triggerSkill(ATTACK_DAMAGE_MAGNIFICATION_LOW);
        }
    }
    else if (attributes.price == CHAMPION_ATTR_MAP.at(FIRST_LEVEL[1]).price) {
        if (attributes.attackRange > ATTACK_RANGE_THRESHOLD) {
            attributes.attackDamage += SKILL_ATTACK_DAMAGE_UP;
            attributes.attackSpeed += SKILL_ATTACK_SPEED_UP;
        }
        else if (attributes.defenseCoefficient >=
DEFENSE_COEFFICIENT_THRESHOLD_LOW) {
            attributes.defenseCoefficient += SKILL_DEFENSE_COEFFICIENT_UP;
        }
        else {
            triggerSkill(ATTACK_DAMAGE_MAGNIFICATION_LOW);
        }
    }
    attributes.magicPoints = 0;
}
```

2. Violates the Open-Closed Principle

When a new skill needs to be added, the internal logic of the `skill` function must be modified, which violates the Open-Closed Principle (open for extension, closed for modification).

3. Violates the Single Responsibility Principle

The `skill` function is responsible not only for executing the skill but also for selecting the skill, which violates the Single Responsibility Principle.。

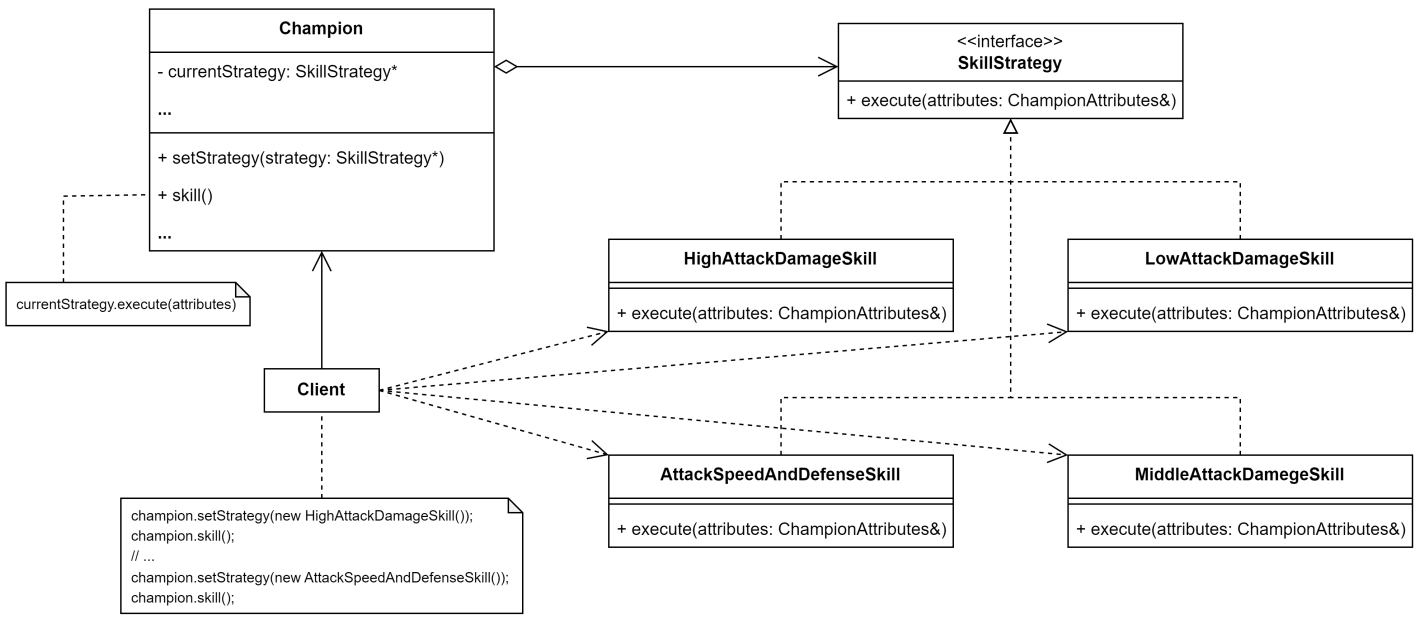
4. Poor Extensibility

Since all the skill logic is concentrated in the `skill` function, adding or modifying a skill requires a deep understanding of the entire function's logic, increasing the risk of errors.

Refactoring Necessity

In the original code, the `skill` method in the `Champion` class has multiple issues, including too many conditional branches, violating the Open-Closed Principle, violating the Single Responsibility Principle, and poor extensibility. Specifically, the `skill` function contains a large number of if-else conditional branches to determine the effects of different heroes' skills, resulting in long and hard-to-maintain code. When adding a new skill, the internal logic of the `skill` function needs to be modified, violating the Open-Closed Principle. The `skill` function is responsible not only for executing the skill but also for selecting the skill, which violates the Single Responsibility Principle. Since all the skill logic is concentrated in the `skill` function, adding or modifying a skill requires a deep understanding of the entire function's logic, increasing the risk of errors. To solve these problems, refactoring using the Strategy Pattern is essential. The Strategy Pattern encapsulates the logic for each skill into independent classes, allowing them to be interchangeable, which improves the maintainability, extensibility, and readability of the code.

UML Class Diagram



Strategy Pattern UML Class Diagram

Refactoring Steps

The Strategy Pattern allows us to define a series of algorithms and encapsulate each algorithm in a separate class, making them interchangeable. We can encapsulate the logic for each skill in an independent strategy class and then choose the appropriate strategy in the `skill` function based on different conditions.

1. Define the Strategy Interface

Define a `SkillStrategy` interface, which includes an `execute` method.

```
class SkillStrategy {
public:
    virtual void execute(ChampionAttributes& attributes) = 0;
    virtual ~SkillStrategy() = default;
};
```

2. Implement Concrete Strategy Classes

Implement a concrete strategy class for each skill, providing the implementation for the `execute` method.

```
class AttackSpeedAndDefenseSkill : public SkillStrategy {
public:
    void execute(ChampionAttributes& attributes) override {
        attributes.attackSpeed += SKILL_ATTACK_SPEED_UP;
        attributes.defenseCoefficient += SKILL_DEFENSE_COEFFICIENT_UP;
    }
};

class HighAttackDamageSkill : public SkillStrategy {
public:
    void execute(ChampionAttributes& attributes) override {
        attributes.attackDamage *= ATTACK_DAMAGE_MAGNIFICATION_HIGH;
    }
};


class LowAttackDamageSkill : public SkillStrategy {
public:
    void execute(ChampionAttributes& attributes) override {
        attributes.attackDamage *= ATTACK_DAMAGE_MAGNIFICATION_LOW;
    }
};

class MiddleAttackDamageSkill : public SkillStrategy {
public:
```

```
    void execute(ChampionAttributes& attributes) override {
        attributes.attackDamage *= ATTACK_DAMAGE_MAGNIFICATION_MIDDLE;
    }
};
```

3. Add the `setStrategy` function and refactor the `skill` function.

Add the private variable `currentStrategy` to the `Champion` class, and then select the appropriate strategy based on the conditions and execute it in the `skill` function.

```
void Champion::setStrategy(SkillStrategy* strategy) {
    if (currentStrategy) {
        delete currentStrategy;
    }
    currentStrategy = strategy;
}

void Champion::skill() {
    if (currentStrategy) {
        currentStrategy->execute(attributes);
    }

    attributes.magicPoints = 0;
}
```

Improvements Achieved

1. **Elimination of Conditional Branches**
   - By encapsulating the logic of each skill into independent strategy classes, the large number of `if-else` conditional branches in the `skill` function is eliminated, making the code more concise and easier to maintain.
2. **Adherence to the Open/Closed Principle**
   - When adding a new skill, only a new strategy class needs to be created without modifying the internal logic of the `skill` function, adhering to the Open/Closed Principle (open for extension, closed for modification).
3. **Single Responsibility Principle**
   - The `skill` function is only responsible for selecting and executing the appropriate strategy, while the specific skill logic is handled by individual strategy classes, adhering to the Single Responsibility Principle.
4. **Improved Extensibility**

- When adding or modifying a skill, only the corresponding strategy class needs to be added or modified without needing to delve into the logic of the `skill` function, reducing the risk of errors.

5. **Code Reusability**
   - Common skill logic is encapsulated in base classes or independent strategy classes, improving code reusability.
6. **Enhanced Readability**
   - Each strategy class has a clear responsibility, making the code structure more straightforward, easier to understand, and maintain.
7. **Dynamic Strategy Switching**
   - The skill strategy can be dynamically switched at runtime, making the behavior of the `Champion` class more flexible.

Refactor Using State Pattern

Original Issue

State pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. Each bond effect is a distinct state, and using the state pattern, we can encapsulate the state logic into separate classes, avoiding conditional branching. The `bond` method in the `Champion` class has the following issues:

1. Excessive Conditional Branches

The `bond` function uses a `switch-case` statement to execute different logic based on different bond types. Each `case` handles a distinct bond effect, leading to long, hard-to-maintain code.

```
void Champion::bond()
{
    switch (attributes.bond) {
        case Brotherhood:
            attributes.movementSpeed *= BROTHERHOOD_MOVEMENT_SPEED_MULTIPLIER;
            attributes.attackSpeed *= BROTHERHOOD_ATTACK_SPEED_MULTIPLIER;
            break;
        case Lout:
            attributes.healthPoints = static_cast<int>(attributes.healthPoints * LOUT_HEALTH_POINTS_MULTIPLIER);
            attributes.movementSpeed *= LOUT_MOVEMENT_SPEED_MULTIPLIER;
            attributes.attackDamage *= LOUT_ATTACK_DAMAGE_MULTIPLIER;
            break;
        case DarkSide:
```

```
            attributes.skillTriggerThreshold = static_cast<int>
(attributes.skillTriggerThreshold * DARKSIDE_SKILL_TRIGGER_MULTIPLIER);
            attributes.attackDamage *= DARKSIDE_ATTACK_DAMAGE_MULTIPLIER;
            break;
        case GoodShooter:
            attributes.attackSpeed *= GOODSHOOTER_ATTACK_SPEED_MULTIPLIER;
            break;
        case PopStar:
            attributes.attackSpeed *= POPSTAR_ATTACK_SPEED_MULTIPLIER;
            attributes.movementSpeed *= POPSTAR_MOVEMENT_SPEED_MULTIPLIER;
            break;
        default:
            break;
    }
}
```

2. Violation of the Open/Closed Principle

When a new bond effect needs to be added, the internal logic of the `bond` function must be modified, violating the Open/Closed Principle (open for extension, closed for modification).

3. Violation of the Single Responsibility Principle

The `bond` function is responsible not only for applying the bond effects but also for selecting the appropriate bond type, violating the Single Responsibility Principle.

4. Poor Extensibility

Since all the bond logic is concentrated in the `bond` function, adding or modifying a bond requires a deep understanding of the entire function's logic, increasing the risk of errors.

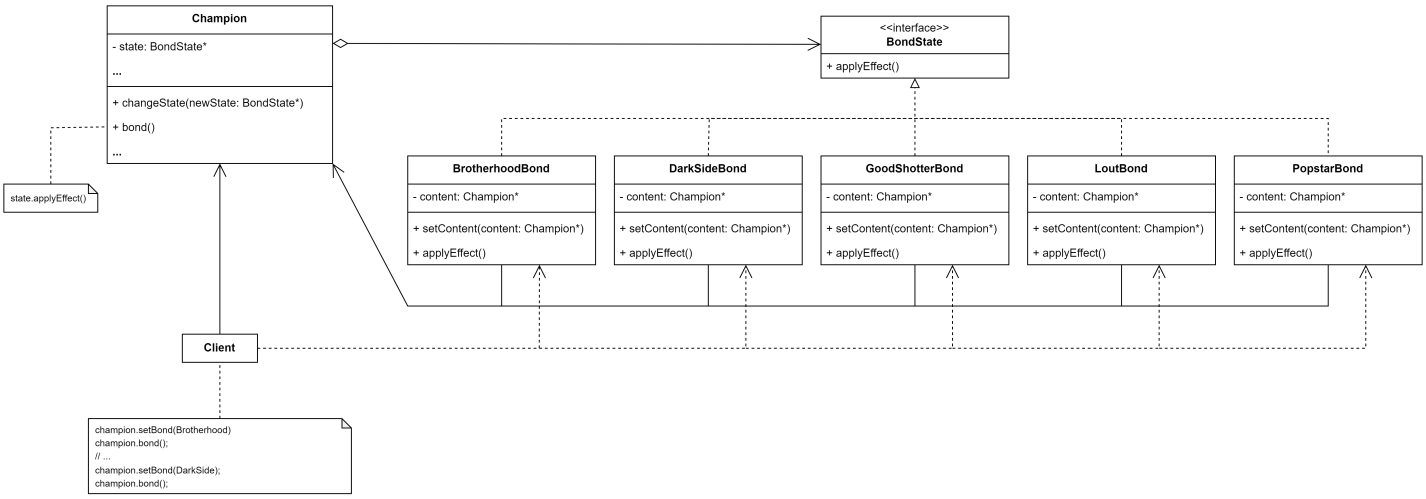
Necessity for Refactoring

In the original code, the `bond` method of the `Champion` class has several issues, such as excessive conditional branching, violation of the Open/Closed Principle, violation of the Single Responsibility Principle, and poor extensibility. Specifically, the `bond` function uses a `switch-case` statement to execute different logic based on bond types, leading to long and difficult-to-maintain code. When a new bond effect needs to be added, the internal logic of the `bond` function must be modified, which violates the Open/Closed Principle. The `bond` function is also responsible for both applying the bond effect and selecting the bond type, which violates the Single Responsibility Principle. Since all the bond logic is concentrated in the `bond` function, adding or modifying bonds requires an in-depth understanding of the entire function, increasing the risk of errors.

To solve these problems, refactoring using the State Pattern is necessary. The State Pattern encapsulates each bond effect into an independent state class, allowing the object to change its

behavior when its internal state changes, thereby improving code maintainability, extensibility, and readability.

UML Class Diagram



State Pattern UML Class Diagram

Refactoring Steps

The State Pattern allows objects to alter their behavior when their internal state changes. We can encapsulate each bond effect into an independent state class, and then in the `bond` function, execute the appropriate logic based on the current bond state.

1. Define a State Interface

Define a `BondState` interface with an `applyEffect` method.

```
class BondState {
public:
    virtual void applyEffect() = 0;
    virtual void setContent(Champion* content) = 0;
    virtual ~BondState() = default;
};
```

2. Implementing concrete state classes

Implement a specific state class for each custody effect that implements the `applyEffect` method.

```
class BrotherhoodBond : public BondState {
public:
    void applyEffect() override {
```

```
        this->getAttributes().movementSpeed *= BROTHERHOOD_MOVEMENT_SPEED_MULTIPLIER;
        this->getAttributes().attackSpeed *= BROTHERHOOD_ATTACK_SPEED_MULTIPLIER;
    }

    void setContent(Champion* content) override {
        this->content = content;
    }

private:
    Champion* content;
};

class DarkSideBond : public BondState {
public:
    void applyEffect() override {
        this->getAttributes().skillTriggerThreshold = static_cast<int>(this->getAttributes().skillTriggerThreshold * DARKSIDE_SKILL_TRIGGER_MULTIPLIER);
        this->getAttributes().attackDamage *= DARKSIDE_ATTACK_DAMAGE_MULTIPLIER;
    }

    void setContent(Champion* content) override {
        this->content = content;
    }

private:
    Champion* content;
};

class GoodShooterBond : public BondState {
public:
    void applyEffect() override {
        this->getAttributes().attackSpeed *= GOODSHOOTER_ATTACK_SPEED_MULTIPLIER;
    }

    void setContent(Champion* content) override {
        this->content = content;
    }

private:
    Champion* content;
};

class LoutBond : public BondState {
```

```
public:
    void applyEffect() override {
        this->getAttributes().healthPoints = static_cast<int>(this->getAttributes().healthPoints * LOUT_HEALTH_POINTS_MULTIPLIER);
        this->getAttributes().movementSpeed *= LOUT_MOVEMENT_SPEED_MULTIPLIER;
        this->getAttributes().attackDamage *= LOUT_ATTACK_DAMAGE_MULTIPLIER;
    }

    void setContent(Champion* content) override {
        this->content = content;
    }

private:
    Champion* content;
};

class PopStarBond : public BondState {
public:
    void applyEffect() override {
        this->getAttributes().attackSpeed *= POPSTAR_ATTACK_SPEED_MULTIPLIER;
        this->getAttributes().movementSpeed *= POPSTAR_MOVEMENT_SPEED_MULTIPLIER;
    }

    void setContent(Champion* content) override {
        this->content = content;
    }

private:
    Champion* content;
};
```

3. Add `changeState` function and refactor `bond` function

Add the private variable `state` to the `Champion` class, add the `changeState` function, and perform the appropriate logic in the `bond` function based on the current custody state.

```
void Champion::changeState(BondState* newState) {
    if (state) {
        delete state;
    }
    state = newState;
    if (state) {
        state->setContent(this);
    }
```

```
 9 }
10
11 void Champion::bond() {
12     if (state) {
13         state->applyEffect(attributes);
14     }
15 }
```

Improvements Achieved

1. **Elimination of Conditional Branches**
   - By encapsulating each bond effect into independent state classes, the `bond` function no longer has a switch-case conditional branch, making the code simpler and easier to maintain.
2. **Adherence to Open/Closed Principle**
   - When adding a new bond effect, only a new state class needs to be added without modifying the internal logic of the `bond` function, adhering to the Open/Closed Principle (open for extension, closed for modification).
3. **Single Responsibility Principle**
   - The `bond` function only handles the selection and application of states, while the specific bond effects are handled by individual state classes, adhering to the Single Responsibility Principle.
4. **Improved Extensibility**
   - When adding or modifying a bond effect, only the relevant state class needs to be modified, without needing to understand the entire `bond` function's logic, reducing the risk of errors.
5. **Code Reusability**
   - Common bond logic is encapsulated into base or independent state classes, enhancing the reusability of the code.
6. **Improved Readability**
   - Each state class has a clear responsibility, making the code structure clearer and easier to understand and maintain.
7. **Dynamic State Switching**
   - Bond states can be dynamically switched at runtime, making the behavior of the `Champion` class more flexible.

Using the Observer Pattern for Refactoring

Original Problem

In the current code, the `Server` class directly handles client connections, message reception, and broadcasting logic. This design has the following issues:

1. Tight Coupling

The `Server` class is tightly coupled with client communication logic, making the code hard to maintain and extend. If new features (such as logging, message filtering, etc.) need to be added in the future, the `Server` class's code will need to be modified.

```
class Server {
public:
    // 构造函数
    Server();

    // 析构函数
    ~Server();

    // 运行服务器
    void run();

    // 友元函数声明
    friend void clientHandler(const SOCKET clientSocket, Server& server);

private:
    WSADATA wsaData;                                    // Windows Sockets API
    SOCKET serverSocket, clientSocket;                  // 服务器和客户端的 socket
    struct sockaddr_in server, client;                  // 服务器和客户端的地址信息
    char hostname[HOSTNAME_MAX_LENGHT];                 // 主机名
    struct hostent* host;                               // 主机地址
    int port;                                           // 端口
    int currentConnections;                             // 服务器当前连接数量
    std::vector<SOCKET> clients;                        // 所有连接到服务器的客户端套接字
    std::vector<std::map<SOCKET, std::string>> playerNames; // 所有连接到服务器的客户端玩家昵称
    std::vector<std::map<SOCKET, std::string>> battleMaps;  // 所有连接到服务器的客户端玩家战斗区地图

    // 创建和尝试绑定套接字
    void createAndBindSocket();

    // 监听和接受客户端的连接请求并进行处理
```

```
    void handleConnections();

    // 检查所有连接到服务器的客户端信息均已发送
    bool areAllReady(const std::vector<std::map<SOCKET, std::string>>& data);

    // 关闭客户端套接字
    void closeClientSocket(const SOCKET clientSocket);

    // 序列化所有连接到服务器的客户端玩家昵称
    std::string serializePlayerNames();

    // 获取键值对数据
    std::string getPairedData(const std::vector<std::map<SOCKET,
std::string>>& data, size_t index);

    // 清空所有连接到服务器的客户端玩家字符串数据
    void clearStrings(std::vector<std::map<SOCKET, std::string>>& data);
};
```

2. Single Responsibility Principle

The `Server` class takes on too many responsibilities, including connection management, message processing, and broadcasting, violating the Single Responsibility Principle.

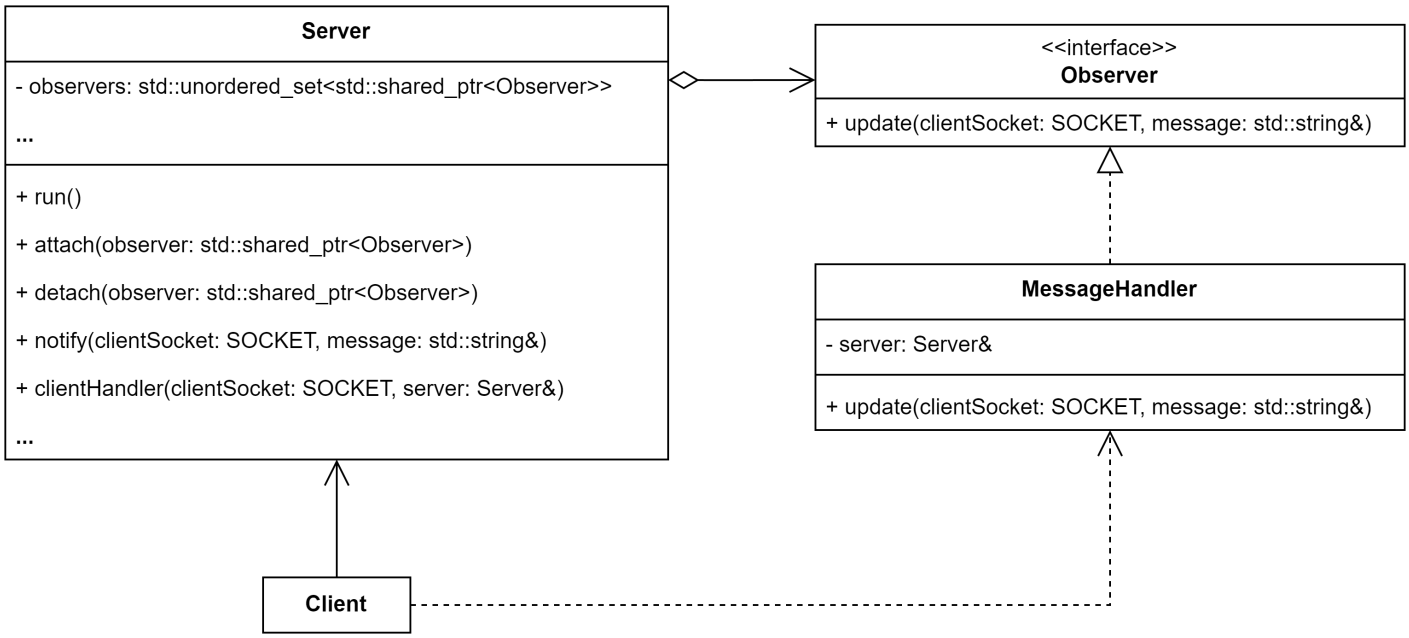
3. Poor Extensibility

If, in the future, support for different types of clients or message handling logic is required, the current design is difficult to extend.

Necessity of Refactoring

In the current code, the `Server` class directly handles client connections, message reception, and broadcasting logic, which leads to tight coupling, violations of the Single Responsibility Principle, and poor extensibility. Specifically, the `Server` class is tightly coupled with client communication logic, making the code difficult to maintain and extend; the `Server` class takes on too many responsibilities, including connection management, message processing, and broadcasting, violating the Single Responsibility Principle; and if new types of clients or message handling logic need to be supported in the future, the current design will be hard to extend. To solve these issues, it is crucial to refactor using the Observer Pattern. The Observer Pattern decouples the message handling logic from the `Server` class, allowing the `Server` to focus solely on connection management and notifying observers, thereby improving the maintainability, extensibility, and readability of the code.

UML Class Diagram



Observer Pattern UML Class Diagram

Refactoring Steps

To refactor the code using the Observer Pattern, we can decouple the message handling logic from the `Server` class, allowing the `Server` to focus only on connection management and notifying observers. The specific steps are as follows:

1. Define the Observer Interface

Create an `Observer` interface that defines the methods the observer needs to implement (e.g., `update`).

```
class Observer {
public:
    virtual ~Observer() = default;
    virtual void update(const SOCKET clientSocket, const std::string& message) = 0;
};
```

2. Create Concrete Observers

Implement concrete observer classes (e.g., `MessageHandler`) responsible for handling client messages.

```
class MessageHandler : public Observer {
public:
    MessageHandler(Server& server) : server(server) {}
    void update(const SOCKET clientSocket, const std::string& message) override;
```

```
5
6 private:
7     Server& server;
8 };
```

3. Modify the Server Class

Modify the `Server` class to become the *Subject* and maintain a list of observers. When a new message arrives, the `Server` class notifies all observers.

```
1  class Server {
2  public:
3      ...
4
5  private:
6      ...
7
8      std::unordered_set<std::shared_ptr<Observer>> observers;
9
10     ...
11 };
```

4. Client Handling Logic

Move the original client handling logic into the specific observer classes.

```
1  void MessageHandler::update(const SOCKET clientSocket, const std::string&
   message) {
2      char buffer[BUFFER_SIZE];
3      strncpy(buffer, message.c_str(), BUFFER_SIZE);
4
5      std::time_t now =
   std::chrono::system_clock::to_time_t(std::chrono::system_clock::now());
6      std::cout << std::put_time(std::localtime(&now), "[%H:%M:%S]");
7
8      std::cout << " [Client: " << clientSocket << "] Message: " << buffer <<
   std::endl;
9
10     if (!strncmp(buffer, BATTLE_MAP_IDENTIFIER, MESSAGE_IDENTIFIER_LENGTH)) {
11         char battleMap[BUFFER_SIZE];
12         sscanf(buffer, BATTLE_MAP_FORMAT, battleMap);
13         for (auto& map : server.battleMaps) {
14             if (map.find(clientSocket) != map.end()) {
15                 map[clientSocket] = static_cast<std::string>(battleMap);
```

```
                break;
            }
        }
    }

    if (!strncmp(buffer, PLAYER_NAME_IDENTIFIER, MESSAGE_IDENTIFIER_LENGTH)) {
        char playerName[BUFFER_SIZE];
        sscanf(buffer, PLAYER_NAME_FORMAT, playerName);
        for (auto& map : server.playerNames) {
            if (map.find(clientSocket) != map.end()) {
                map[clientSocket] = static_cast<std::string>(playerName);
                break;
            }
        }
    }

    if (!strncmp(buffer, HEALTH_POINTS_IDENTIFIER, MESSAGE_IDENTIFIER_LENGTH))
{
        for (const SOCKET& sock : server.clients) {
            send(sock, buffer, strlen(buffer), 0);
        }
    }

    if (server.currentConnections % 2 == 0) {
        if (server.areAllReady(server.playerNames)) {
            strcpy(buffer, server.serializePlayerNames().c_str());
            std::cout << "Broadcast: " << buffer << std::endl;
            for (const SOCKET& sock : server.clients) {
                send(sock, buffer, strlen(buffer), 0);
            }
        }
    }

    if (server.areAllReady(server.battleMaps)) {
        for (size_t i = 0; i < server.clients.size(); i++) {
            sprintf(buffer, BATTLE_MAP_FORMAT,
server.getPairedData(server.battleMaps, i).c_str());
            std::cout << "Send Client " << server.clients[i ^ 1] << "[" << (i
^ 1) << "]'s BattleMap to Client " << server.clients[i] << "[" << i << "]: "
<< buffer << std::endl;
            send(server.clients[i], buffer, strlen(buffer), 0);
        }
        server.clearStrings(server.battleMaps);
    }
}
```

Improvements Achieved

1. **Decoupling and Abstraction**
   - By defining the `Observer` interface, the message processing logic was decoupled from the `Server` class, allowing the `Server` class to focus solely on connection management and notifying observers.
2. **Single Responsibility Principle**
   - The `Server` class no longer handles message processing, but rather focuses on managing connections, in accordance with the Single Responsibility Principle.
3. **Enhanced Extensibility**
   - Adding new message processing logic only requires adding new observer classes without modifying the `Server` class code, improving code extensibility.
4. **Dynamic Observer Addition**
   - Observers can be dynamically added or removed at runtime, making the system more flexible.
5. **Code Reusability**
   - Common message processing logic is encapsulated in base classes or independent observer classes, enhancing code reusability.
6. **Improved Readability**
   - Each observer class has a clear responsibility, resulting in a more organized, easier-to-understand code structure.
7. **Support for Multiple Message Types**
   - Different observer classes can be created for different types of messages, making message handling logic more modular.

Refactoring Using a Design Pattern Not Covered in Class: **Reactor Pattern**

Refactoring Using **Reactor Pattern**

Reactor Pattern

The Reactor Pattern is an event-driven design pattern used for efficiently handling multiple service requests concurrently. The key component of this pattern is an event loop, typically running in a single thread or process, which handles multiplexing incoming requests and dispatches them to the appropriate request handler.

Core Concepts:

- **Event-driven:** The Reactor Pattern handles I/O operations through event notifications instead of blocking I/O or multithreading. This allows efficient handling of numerous concurrent I/O requests in a single thread.
- **Non-blocking I/O:** The Reactor Pattern relies on non-blocking I/O, where events are triggered only when I/O operations are completed, avoiding the issue of blocking threads waiting for I/O.
- **Callback Mechanism:** Request handlers are registered as callback functions within the event loop. When an event occurs, the event loop calls the corresponding handler function to process the event. This design separates event handling logic from event dispatching, enhancing flexibility and maintainability.

Workflow:

- **Register Event Handlers**: Before starting the event loop, the application registers event handlers along with their corresponding handles to the dispatcher.
- **Event Loop**: The event loop uses a multiplexer to monitor all registered handles, waiting for events to occur.
- **Event Triggering**: When an I/O operation on a handle is completed, the multiplexer notifies the dispatcher.
- **Calling Handlers**: The dispatcher calls the appropriate event handler based on the event type for processing.

Advantages:

- **High Concurrency**: By using non-blocking I/O and an event-driven mechanism, the Reactor Pattern can efficiently handle many concurrent requests within a single thread.
- **Low Resource Consumption**: Avoids the overhead of context switching and memory consumption typically associated with multithreading.
- **Flexibility**: Event handlers are registered as callbacks, making it easy to extend and modify the system.

Disadvantages:

- **Difficult Debugging**: Due to the callback mechanism, the program flow becomes more complex, making debugging and analysis more difficult.
- **Single-threaded Bottleneck**: For compute-heavy tasks, a single thread may become a performance bottleneck.
- **Increased Complexity**: The Reactor Pattern is more complex to implement compared to simpler "one thread per connection" models.

Applications:

The Reactor Pattern is widely used in scenarios requiring high concurrency, especially in I/O-bound applications, such as:

- **Web Servers**: e.g., Nginx, Node.js
- **Network Frameworks**: e.g., Netty, Twisted
- **Application Servers**: e.g., Spring Framework

Original Issue

The main issue with the current code is:

1. Blocking I/O

The `listenForServerMessages` function uses blocking I/O operations, causing threads to be blocked while waiting for data, thus reducing the responsiveness of the program.

```
void OnlineModeControl::listenForServerMessages()
{
    while (keepListening) {
        char buffer[BUFFER_SIZE];
        recv(s, buffer, BUFFER_SIZE, 0);
        if (!strncmp(buffer, HEALTH_POINTS_IDENTIFIER,
MESSAGE_IDENTIFIER_LENGTH)) {
            int healthPoints;
            SOCKET socket;
            sscanf(buffer, HEALTH_POINTS_FORMAT, &healthPoints, &socket);
            updatePlayerHealthPoints(healthPoints, socket);
        }

std::this_thread::sleep_for(std::chrono::milliseconds(THREAD_SLEEP_DURATION_MIL
LISECONDS));
    }
}
```

2. Complex Thread Management

The code uses explicit thread management (std::thread), which increases the complexity of the code and makes it prone to thread safety issues.

```
OnlineModeControl::OnlineModeControl(std::string ipv4, std::string portStr) :
    port(std::stoi(portStr)),
    recvSize(0),
    currentConnections(0),
```

```
    keepListening(true),
    Control(MAX_CONNECTIONS)
{
    strcpy(this->message, "");
    strcpy(this->ipv4, ipv4.c_str());
    try {
        humanPlayer = new HumanPlayer(g_PlayerName);
        enemyPlayer = new HumanPlayer("");
    }
    catch (const std::bad_alloc& e) {
        std::cerr << "Memory allocation failed: " << e.what() << std::endl;
        if (humanPlayer) {
            delete humanPlayer;
        }
        if (enemyPlayer) {
            delete enemyPlayer;
        }
        throw;
    }
    listeningThread = std::thread(&OnlineModeControl::listenForServerMessages,
this);
}
```

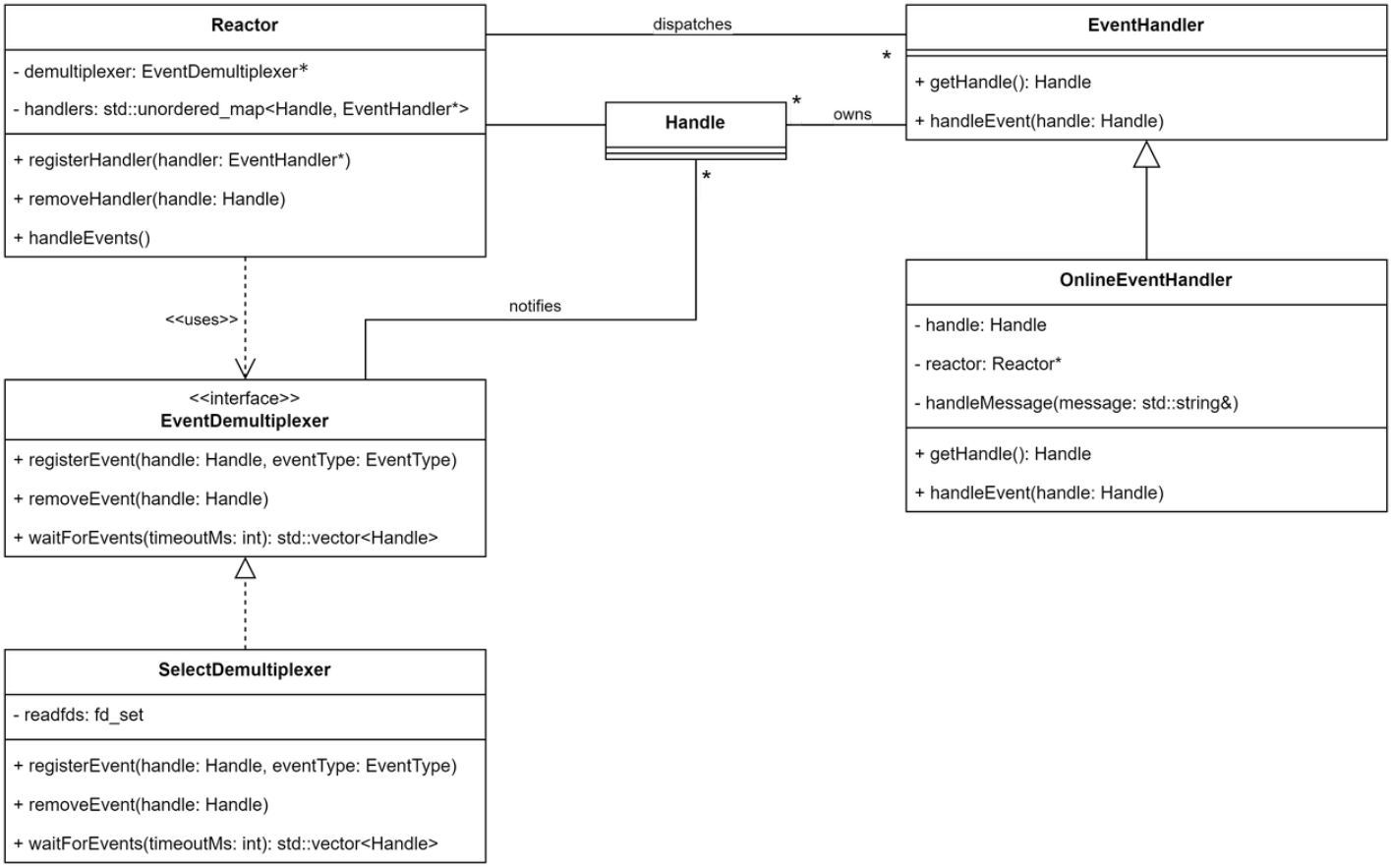
3. Lack of Event-Driven Mechanism

The current code does not fully leverage an event-driven mechanism, resulting in poor scalability and maintainability of the code.

Refactoring Necessity

In the current code, the `listenForServerMessages` function uses blocking I/O operations, which causes the thread to be unable to perform other tasks while waiting for data, reducing the responsiveness of the program. Additionally, the code uses explicit thread management (`std::thread`), which increases the complexity of the code and makes it prone to thread safety issues. Furthermore, the current code does not fully utilize an event-driven mechanism, resulting in poor scalability and maintainability. To address these issues, using the Reactor Pattern for refactoring is crucial. The Reactor pattern, with its event-driven and non-blocking I/O mechanisms, allows for the efficient handling of numerous concurrent requests in a single thread, improving the responsiveness, scalability, and maintainability of the code.

UML Class Diagram



Reactor Pattern UML Class Diagram

Refactoring Steps

In order to refactor the code using the Reactor Pattern, we need:

1. Define event types and handles

- In `EventDemultiplexer.h` , define the `EventType` enumeration and `Handle` type.
- `EventType` is used to distinguish between different types of events (such as read events, write events), and `Handle` is a type alias for file descriptors (such as sockets).

```
enum class EventType {
    READ_EVENT,
    WRITE_EVENT
};

using Handle = SOCKET;
```

2. Implement event multiplexer distributor

- In `EventDemultiplexer.h` implement the `EventDemultiplexer` interface and the `SelectDemultiplexer` class.

- `EventDemultiplexer` is responsible for listening to multiple file descriptor events, and `SelectDemultiplexer` is the specific implementation, using the `select` system call.

```cpp
class EventDemultiplexer {
public:
    virtual ~EventDemultiplexer() = default;
    virtual void registerEvent(Handle handle, EventType eventType) = 0;
    virtual void removeEvent(Handle handle) = 0;
    virtual std::vector<Handle> waitForEvents(int timeoutMs) = 0;
};

class SelectDemultiplexer : public EventDemultiplexer {
public:
    SelectDemultiplexer() {
        FD_ZERO(&readfds);
    }

    void registerEvent(Handle handle, EventType eventType) override {
        FD_SET(handle, &readfds);
    }

    void removeEvent(Handle handle) override {
        FD_CLR(handle, &readfds);
    }

    std::vector<Handle> waitForEvents(int timeoutMs) override {
        fd_set tmpfds = readfds;
        struct timeval timeout = {0, timeoutMs * 1000};

        int selectResult = select(0, &tmpfds, nullptr, nullptr, &timeout);
        if (selectResult > 0) {
            std::vector<Handle> activeHandles;
            for (int i = 0; i < FD_SETSIZE; ++i) {
                if (FD_ISSET(i, &tmpfds)) {
                    activeHandles.push_back(i);
                }
            }
            return activeHandles;
        }
        return {};
    }

private:
    fd_set readfds;
};
```

3. Define an event handler interface

- In `EventHandler.h` , define the `EventHandler` abstract class.
- `EventHandler` is the base class for all concrete event handlers and defines the interface for handling events.

```
class EventHandler {
public:
    virtual ~EventHandler() = default;
    virtual void handleEvent(Handle handle) = 0;
    virtual Handle getHandle() const = 0;
};
```

4. Implement specific event handlers

- Implement the `OnlineEventHandler` class in `OnlineEventHandler.h` .
- `The OnlineEventHandler` is the concrete event handler that handles network messages and updates player status.

```
class OnlineEventHandler : public EventHandler {
public:
    OnlineEventHandler(Handle handle, Reactor* reactor)
        : handle(handle), reactor(reactor) {}

    void handleEvent(Handle handle) override {
        char buffer[BUFFER_SIZE];
        int recvSize = recv(handle, buffer, BUFFER_SIZE, 0);
        if (recvSize > 0) {
            buffer[recvSize] = '\0';
            handleMessage(buffer);
        }
    }

    Handle getHandle() const override {
        return handle;
    }

private:
    void handleMessage(const std::string& message) {
        if (!strncmp(message.c_str(), HEALTH_POINTS_IDENTIFIER,
MESSAGE_IDENTIFIER_LENGTH)) {
            int healthPoints;
            SOCKET socket;
```

```
            sscanf(message.c_str(), HEALTH_POINTS_FORMAT, &healthPoints,
&socket);
            updatePlayerHealthPoints(healthPoints, socket);
        }
    }

    Handle handle;
    Reactor* reactor;
};
```

5. Implementing the `Reactor` class

- Implement the `Reactor` class in `Reactor.h`.
- `Reactor` is the core of the event loop, responsible for registering event handlers, listening to events, and calling the corresponding callbacks.

```
class Reactor {
public:
    Reactor() : demultiplexer(new SelectDemultiplexer()) {}

    ~Reactor() {
        delete demultiplexer;
    }

    void registerHandler(EventHandler* handler) {
        handlers[handler->getHandle()] = handler;
        demultiplexer->registerEvent(handler->getHandle(),
EventType::READ_EVENT);
    }

    void removeHandler(Handle handle) {
        demultiplexer->removeEvent(handle);
        handlers.erase(handle);
    }

    void handleEvents() {
        auto activeHandles = demultiplexer->waitForEvents(100);
        for (auto handle : activeHandles) {
            if (handlers.find(handle) != handlers.end()) {
                handlers[handle]->handleEvent(handle);
            }
        }
    }

private:
```

```
    EventDemultiplexer* demultiplexer;
    std::unordered_map<Handle, EventHandler*> handlers;
};
```

6. Refactor main logic

- Initialize the `Reactor` and `OnlineEventHandler` in the main logic and start the event loop.
- With `Reactor` 's event loop, the main logic can efficiently handle multiple concurrent events without explicitly managing threads.

```
class OnlineModeControl {
public:
    OnlineModeControl(std::string ipv4, std::string portStr)
        : port(std::stoi(portStr)), keepListening(true) {
        ...
        reactor = new Reactor();
        eventHandler = new ConcreteEventHandler(s, reactor);
        reactor->registerHandler(eventHandler);
    }

    ~OnlineModeControl() {
        ...
        delete reactor;
        delete eventHandler;
    }

    void start() {
        while (keepListening) {
            reactor->handleEvents();

std::this_thread::sleep_for(std::chrono::milliseconds(THREAD_SLEEP_DURATION_MIL
LISECONDS));
        }
    }

    ...

private:
    ...
    Reactor* reactor;
    EventHandler* eventHandler;
    SOCKET s;
    int port;
    bool keepListening;
```

```
33 };
```

Improvements Achieved

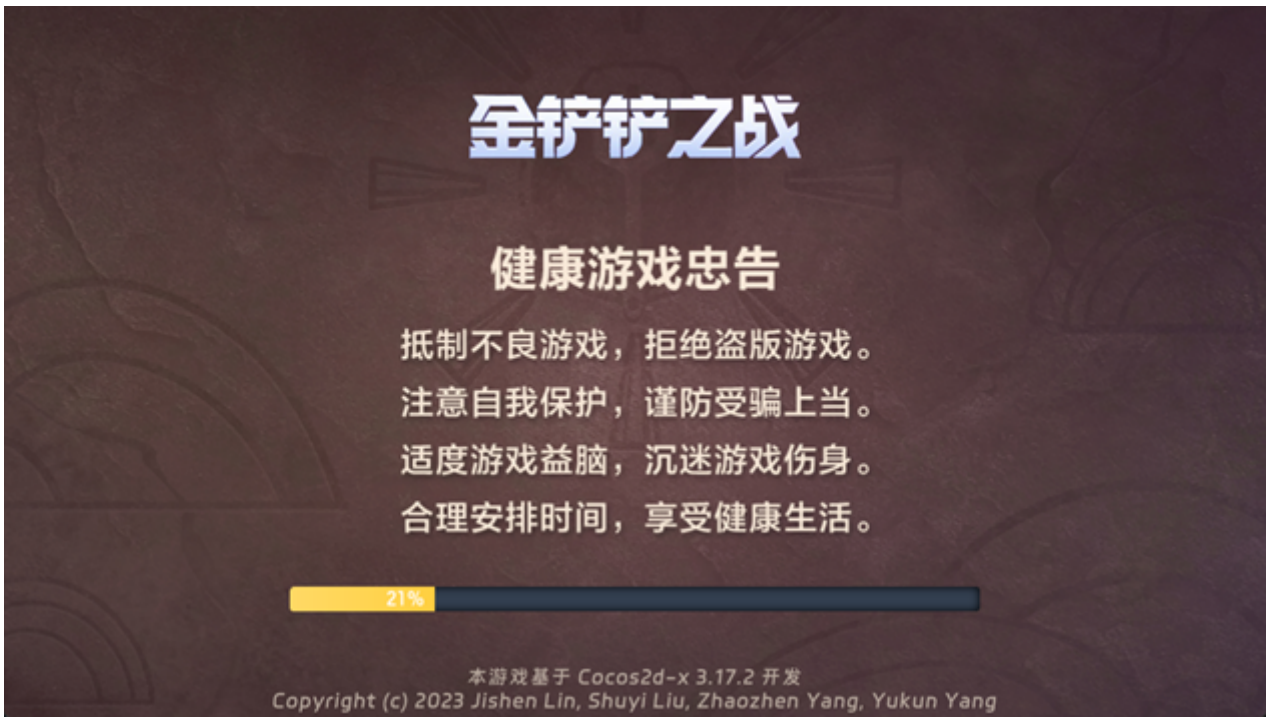
1. **Non-blocking I/O improves responsiveness**
   - **Improvement point** : The original `listenForServerMessages` function used blocking I/O operations (`recv`), which caused threads to be unable to perform other tasks while waiting for data, reducing the responsiveness of the program.
   - **After refactoring** : Through the Reactor pattern, I/O operations become non-blocking, and the Event Loop can handle other events while waiting for I/O operations to complete, significantly improving the responsiveness of the program.
2. **Simplify thread management**
   - **Improvement point** : The original code used explicit thread management (`std :: thread`), which increased the complexity of the code and easily caused thread safety issues.
   - **After refactoring** : The Reactor pattern handles all concurrent requests through a single-threaded event loop, avoiding the complexity of explicit thread management, reducing the overhead of thread context switching, and reducing the risk of thread safety issues.
3. **Event-driven mechanism improves scalability**
   - **Improvement point** : The original code lacks event-driven mechanism, resulting in poor scalability and maintainability of the code.
   - **After refactoring** : The Reactor pattern separates event handling logic from event distribution logic through an event-driven mechanism, making the code easier to extend and maintain. When adding an event type or handler, only the corresponding event handler needs to be registered without modifying the core logic.
4. **Resource consumption is reduced**
   - **Improvement point** : The original multi-threaded model will bring higher context switching and memory overhead.
   - **After refactoring** : Reactor mode handles all concurrent requests in a single thread, avoiding context switching and memory overhead caused by multi-threading, and reducing resource consumption.
5. **Code structure is clearer**
   - **Improvement point** : The original code structure is rather confusing, and the event handling logic is coupled with thread management logic.

    - **After refactoring** : The Reactor pattern separates event handling logic from event distribution logic, making the code structure clearer and easier to understand and maintain.

6. **Support high concurrency processing**

- **Improvement point** : The original blocking I/O and multi-threaded model performs poorly when handling a large number of concurrent requests.
- **After refactoring** : The Reactor pattern can efficiently handle a large number of concurrent requests in a single thread through non-blocking I/O and event-driven mechanisms, especially suitable for I/O-intensive applications.

Project Presentation




金铲铲之战
欢迎你!
请输入您的游戏昵称
游戏昵称长度不超过6个中英文与数字字符
点击此处开始游戏


金铲铲之战
人类玩家，魔法世界等你探索，奇迹即将发生.
英雄联盟云顶之弈正版授权手游
金铲铲之战
练习模式
联机模式
设置


金铲铲之战
多数玩家开始游戏，游戏中请与服务器保持连接。
服务器连接过程可能出现卡顿现象，请耐心等待。
联机模式
Online
请输入服务器 IPv4 地址
加入房间
返回菜单
请输入服务器端口


设置
Settings
背景音乐
音 效
练习模式
游戏难度
简单
正常
困难
返回菜单


游戏部分玩法介绍
1. 羁绊系统：本游戏共设置五种羁绊，满足条件会对局面有较大增益。
死神（DarkSide）
伊芙琳、阿木木、卡尔萨斯
神射手（GoodShooter）
库奇、金克斯
莽夫（Lout）
奥拉夫、潘森
星之守护（PopStar）
卡莎
兄弟（Brotherhood）
永恩、亚索
2. 天赋系统：本游戏共设置三种天赋，对局面有较大增益，可以在游戏开始前自选。
海盗：增加每回合金币数量
将军：增加英雄位
斗士：增加初始生命值
返回菜单


倒计时
5s
人类玩家 100 1
AI玩家 100 2
潘森
Level 1
HP 700 1
攻击伤害 70
移动速度 1.00
攻击范围 1
防御系数 1.20
攻击速度 0.50
技能阈值 200
等级：1
5
升级 12
刷新 2
库奇 2
奥拉夫 1
返回菜单

Summary

The Meaning of Refactoring with Design Pattern

Design pattern is a validated best practice in software development that can help developers solve common code design problems. The significance of refactoring is:

- **Improve Code Quality** : Refactoring improves the overall code quality by optimizing the code structure to make it easier to understand and maintain. Clear code not only reduces the cognitive burden on developers, but also reduces the likelihood of errors.
- **Improve maintainability** : Refactoring makes code easier to modify and extend. As the project grows, a clear code structure can significantly reduce maintenance costs, avoid the accumulation of technical debt, and ensure the long-term stable operation of the system.
- **Optimize performance** : Refactoring removes performance bottlenecks in code and improves system performance. By optimizing algorithms and resource usage, refactoring reduces unnecessary resource consumption and improves overall performance.
- **Support new features** : Refactoring makes the code more extensible and easier to add new features. Clear code structure reduces the risk of conflicts between new features and existing code, making the development process smoother.
- **Promote team collaboration** : Refactoring helps unify code style and standards, making it easier for team members to understand and collaborate. Clear code also helps new members get started quickly, promotes knowledge sharing and teamwork.
- **Reduce risk** : Refactoring reduces defects by identifying and fixing potential problems. Optimized code is more stable, reducing the likelihood of system crashes and errors.

- **Adapt to change** : Refactoring makes the code more flexible and able to better respond to changes in business needs. At the same time, refactoring also facilitates the introduction of new technologies and tools, ensuring that the system keeps up with the times.
- **Improve development efficiency** : Refactoring reduces debugging time and makes the development process more efficient. The clear code structure also facilitates automated testing, further improving development efficiency and code reliability.

Design pattern refactoring not only solves the problems in the current code, but also lays a good foundation for future development.

This project 's refactoring method

We have refactored several modules in the project, mainly applying the following design patterns:

- **Builder Pattern** : Refactored the initialization logic of the `Champion` class, solved the problem of hardcoding and complex initialization, and improved the maintainability and scalability of the code.
- **Singleton Pattern** : Applied to the `LocationMap` class, ensuring that there is only one instance globally, avoiding memory waste and data inconsistency caused by multiple instances.
- **Composite Pattern** : Refactored `ChampionAttributesLayer` classes, decoupled specific node types, reduced code duplication, and increased flexibility.
- **Facade pattern (Facade Pattern)** : encapsulates the audio playback logic in the `AudioController` class, simplifying external calls and reducing coupling.
- **Strategy Pattern** : Refactored the skill logic of the `Champion` class, eliminated a large number of conditional branches, and improved the scalability and maintainability of the code.
- **State Pattern** : The binding logic applied to the `Champion` class encapsulates different binding effects into independent state classes, improving the scalability of the code.
- **Observer Pattern** : Decouples the message processing logic of the `Server` class, making the system more flexible and easy to extend.
- **Reactor Pattern (Reactor Pattern)** : Introduces event-driven and non-blocking I/O mechanism, replaces the original blocking I/O, and improves the concurrent processing capability of the system.

Through these refactorings, the code quality of the project has been significantly improved, and the maintainability, scalability, and performance of the system have been optimized.

Summary

Through the refactoring case in this article, the following knowledge can be learned:

- **Application scenarios of Design patterns** : Different Design patterns are suitable for different scenarios. For example, the generator pattern is suitable for creating complex objects, the singleton pattern is suitable for managing globally unique instances, and the policy pattern is suitable for dynamic switching of algorithms, etc.
- **The importance of code decoupling** : Through Design patterns (such as facade pattern, observer pattern), various modules in the system can be decoupled, reducing the complexity of the code, improving the flexibility and maintainability of the system.
- **The steps of refactoring** : Refactoring is not just about modifying code, but also analyzing existing problems, designing solutions, implementing step by step, and verifying the effect. This article shows how to implement refactoring step by step through UML class diagrams and code examples.
- **Performance optimization** : Through the reactor mode, the system's concurrent processing capacity can be improved and resource consumption can be reduced. This is particularly important for high-concurrency application scenarios (such as online games).
- **Readability and maintainability of the code** : Design patterns not only solve technical problems, but also make the code structure clearer, easier to understand and maintain. This is very important for team collaboration and long-term project maintenance.

References

1. Refactoring.Guru (https://refactoring.guru)
2. Design Patterns (https://en.wikipedia.org/wiki/Design_Patterns)
3. Reactor Pattern (https://en.wikipedia.org/wiki/Reactor_pattern)