Ответ на вопрос по выбору по теме «паттерн Decorator»

Выполнил:
Студент 2 курса группы БПМ-22-4 Исупов Г.С. 29.12.23

Паттерн Декоратор (или Wrapper)

Декоратор - структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Паттерн «Декоратор» позволяет динамически добавлять объекту новые обязанности, не прибегая при этом к порождению классов.

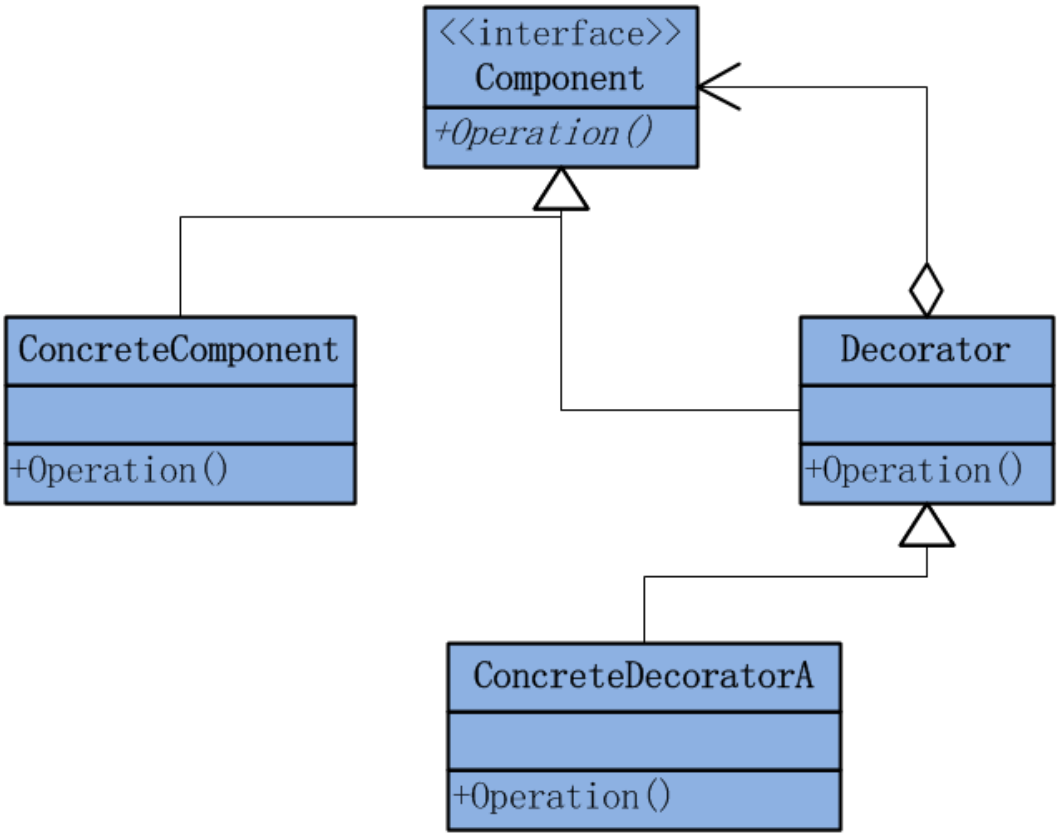
Назначение паттерна

Шаблон декоратора предназначен для решения этой проблемы: он избегает множественного наследования и реализует функцию изменения определенного метода.

Намерение: динамически добавлять некоторые дополнительные обязанности к объекту. С точки зрения расширенной функциональности шаблон Декоратор является более гибким, чем генерация подклассов.

То есть паттерн «Декоратор», по сути, использует схему "обертываем подарок, кладем его в коробку, обертываем коробку".

Логическая схема



1. Компонент (Component): определяет интерфейс для объектов, которые могут быть декорированы.
2. Конкретный компонент (ConcreteComponent): реализует интерфейс компонента и предоставляет базовую функциональность.
3. Декоратор (Decorator): содержит ссылку на объект типа Component и определяет интерфейс, совместимый с интерфейсом Component.
4. Конкретный декоратор (ConcreteDecoratorA): расширяет функциональность компонента, добавляя дополнительное поведение.

Пример кода

```cpp
#include <iostream>
#include <string>

// 1. Компонент
class Coffee {
public:
    virtual ~Coffee() = default;

    virtual std::string getDescription() const = 0;

    virtual double cost() const = 0;
};

// 2. Конкретный компонент
class Espresso : public Coffee {
public:
    std::string getDescription() const override {
        return "Espresso";
    }

    double cost() const override {
        return 1.99;
    }
};

// 3. Декоратор
class CoffeeDecorator : public Coffee {
public:
    explicit CoffeeDecorator(Coffee* c) : coffee(c) {}

    std::string getDescription() const override {
        return coffee->getDescription();
    }

    double cost() const override {
        return coffee->cost();
    }

protected:
    Coffee* coffee;
};

// 4. Конкретный декоратор
class MilkDecorator : public CoffeeDecorator {
public:
    explicit MilkDecorator(Coffee* c) : CoffeeDecorator(c) {}
```

```
    std::string getDescription() const override {
        return coffee->getDescription() + ", Milk";
    }

    double cost() const override {
        return coffee->cost() + 0.50;
    }
};

int main() {
    // Создаем базовый компонент
    Coffee* espresso = new Espresso();
    std::cout << "Coffee: " << espresso->getDescription() << ", Cost: $" << espresso-
>cost() << std::endl;

    // Декорируем его молоком
    Coffee* milkCoffee = new MilkDecorator(espresso);
    std::cout << "Decorated coffee: " << milkCoffee->getDescription() << ", Cost: $" <<
milkCoffee->cost() << std::endl;

    delete espresso;
    delete milkCoffee;

    return 0;
}
```

В этом примере Espresso — это конкретный компонент, а MilkDecorator - конкретный декоратор, добавляющий функциональность в виде молока к базовому кофе.

Вывод в консоль:

```
Coffee: Espresso, Cost: $1.99

Decorated coffee: Espresso, Milk, Cost: $2.49
```

То есть в этом коде показано то, как мы добавили и использовали функциональность к уже существующему объекту.

Другие примеры использования паттерна

1. Логирование
2. Выделение выполнения кода в отдельный поток
3. Декорированный/форматированный вывод данных
4. Шифрование - можно использовать для добавления слоя шифрования к существующему объекту
5. Кэширование - можно применить для добавления кеширования к некоторому вычислительному или запросному объекту
6. Валидация данных, переданных в объект – можно проверять корректность формата данных перед их обработкой
7. Контроль доступа - можно обеспечить контроль над выполнением определенных действий в зависимости от прав пользователя

Заключение

Преимущества паттерна «Декоратор»:

1. Декоратор упрощает расширение функциональности существующего объекта во время выполнения и компиляции.
2. Декоратор также обеспечивает гибкость для добавления любого количества декораторов в любом порядке и смешивания.
3. Декораторы - хорошее решение проблем с перестановкой, потому что вы можете обернуть компонент любым количеством декораторов.

Недостатки:

1. Декораторы могут усложнить процесс создания экземпляра компонента, потому что вам нужно не только создать экземпляр компонента, но и обернуть его рядом декораторов.
2. Чрезмерное использование шаблона проектирования декоратора может усложнить систему с точки зрения как обслуживания

Источник

1. https://ru.wikipedia.org/wiki/Декоратор_(шаблон_проектирования)