

Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 9
03.11.2017

Пример: паттерн Singleton

Глобальные переменные — это некоторое зло.

a.cpp:

```
std::vector<int> va;  
//...
```

b.cpp:

```
extern std::vector<int> va;  
struct TInit {  
    TInit() { va.push_back(1);}  
};  
TInit Init;
```

Порядок инициализации?

Глобальные объекты → local static объекты:

```
std::vector& GetVal() {  
    static std::vector<int> va;  
    return va;  
}
```

Пример: паттерн Singleton

Singleton — класс, у которого в любой момент времени существует не более одного объекта.

Пример: паттерн Singleton

Singleton — класс, у которого в любой момент времени существует не более одного объекта.

```
class Singleton {
private:
    Singleton(){}
    static Singleton* instance;
public:
    // data
    // ...
    Singleton(const Singleton&) = delete;
    static Singleton* Instance() {
        if (instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }
};

Singleton* Singleton::instance = nullptr;
```

Пример: паттерн Singleton

```
class Singleton {
protected:
    Singleton(){ /*...*/}
    ~Singleton(){ /*...*/}
public:
    // data
    // ...
    Singleton(const Singleton&) = delete;
    Singleton(Singleton&&) = delete;
    Singleton& operator=(Singleton const&) = delete;
    Singleton& operator=(Singleton &&) = delete;
    static Singleton& Instance() {
        static Singleton instance;
        return instance;
    }
};
```

Пример: паттерн Singleton

Почему это плохой паттерн?

- ▶ Это скрывание глобальной переменной — в обход всего к ней можно получить доступ.
- ▶ Сложно работать с наследованием.
- ▶ Невозможно простым способом развернуть код в несколько функций с разными объектами-синглтонами.

Пример: паттерн Strategy

Паттерн, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости.

- ▶ Инкапсуляция алгоритма,
- ▶ увеличение модульности и проверяемости кода,
- ▶ дешевое масштабирование кода,
- ▶ выбор алгоритма, основываясь на данных (в процессе исполнения кода можно это изменить).

Пример: паттерн Strategy

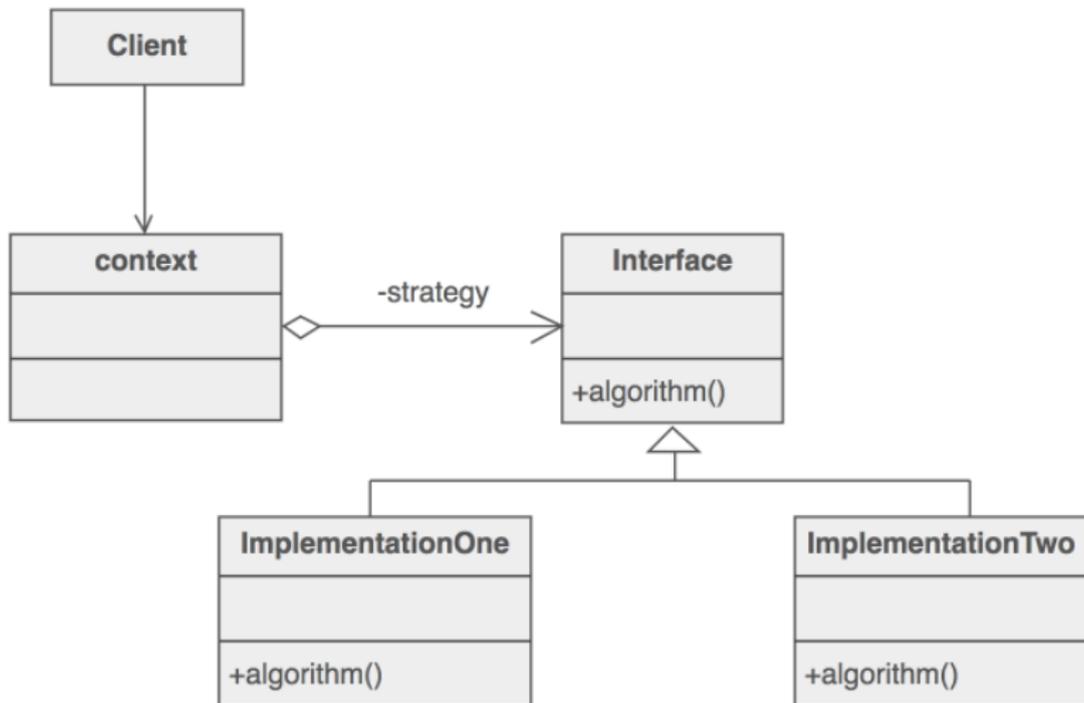
Паттерн, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости.

- ▶ Инкапсуляция алгоритма,
- ▶ увеличение модульности и проверяемости кода,
- ▶ дешевое масштабирование кода,
- ▶ выбор алгоритма, основываясь на данных (в процессе исполнения кода можно это изменить).

Когда?

- ▶ Нужны разные варианты алгоритма или поведения,
- ▶ нужно изменять поведение объектов в runtime,
- ▶ нужны разные алгоритмы в зависимости от состояния.

Пример: паттерн Strategy



Пример: паттерн Strategy и кофе-машина

```
class Recipe {
public:
    virtual double GetAmountOfWater() const = 0;
    virtual void Make() = 0;
};

class HotBeverage {
    void BoilWater(double amount) {
        std::cout << "boiling " << amount << " ml of water..."
        << std::endl;
    }
    void Pour() {
        std::cout << "pouring in cup" << std::endl;
    }
    std::shared_ptr<Recipe> recipe;
public:
    HotBeverage(std::shared_ptr<Recipe> r) : recipe(r) {}
    void prepare() {
        BoilWater(recipe->GetAmountOfWater());
        recipe->Make();
        Pour();
    }
};
```

Пример: паттерн Strategy и кофе-машина

```
class Coffee: public Recipe {
    double AmountOfWater;
    int StrongLevel;
public:
    Coffee(double amountOfWater, int level)
        : AmountOfWater(amountOfWater)
        , StrongLevel(level)
    { }
    virtual double GetAmountOfWater() const { return AmountOfWater; }
    virtual void Make() { std::cout << "brewing coffee..."; }
};
```

```
class HotChocolate : public Recipe {
    double AmountOfWater;
public:
    HotChocolate(double amountOfWater)
        : AmountOfWater(amountOfWater)
    { }
    virtual double GetAmountOfWater() const { return AmountOfWater; }
    virtual void Make() { std::cout << "making hot chocolate..."; }
};
```

Пример: паттерн Strategy и кофе-машина

```
int main() {  
    auto coffee = std::make_shared<Coffee>(200, 3);  
    auto hotChocolate = std::make_shared<HotChocolate>(100);  
    std::vector<HotBeverage> beverages = {  
        HotBeverage(coffee),  
        HotBeverage(hotChocolate)  
    };  
    for (auto&x : beverages) x.prepare();  
}
```

Пример: паттерн Strategy и кофе-машина через лямбды

```
class HotBeverage {
    void BoilWater(double amount) {
        std::cout << "boiling " << amount << " ml of water...";
    }
    void Pour() {
        std::cout << "pouring in cup" << std::endl;
    }
    std::function<double()> GetAmountOfWater;
    std::function<void()> Make;
public:
    HotBeverage(std::function<double()> getAmountOfWater,
               std::function<void()> make)
        : GetAmountOfWater(getAmountOfWater)
        , Make(make) {}
    void prepare() {
        BoilWater(GetAmountOfWater());
        Make();
        Pour();
    }
};
```

Пример: паттерн Strategy и кофе-машина через лямбды

```
static void MakeCofee() { std::cout << "brewing coffee..."; }
static void MakeHotChocolate() { std::cout << "making chocolate..."; }
static double GetAmountOfWater(double amount) { return amount; }

int main() {
    auto coffee = HotBeverage(
        [] { return GetAmountOfWater(200); },
        MakeCofee
    );
    auto hotChocolate = HotBeverage(
        [] { return GetAmountOfWater(100); },
        MakeHotChocolate
    );
    std::vector<HotBeverage> beverages = {
        coffee, hotChocolate
    };
    for (auto&x : beverages) x.prepare();
}
```

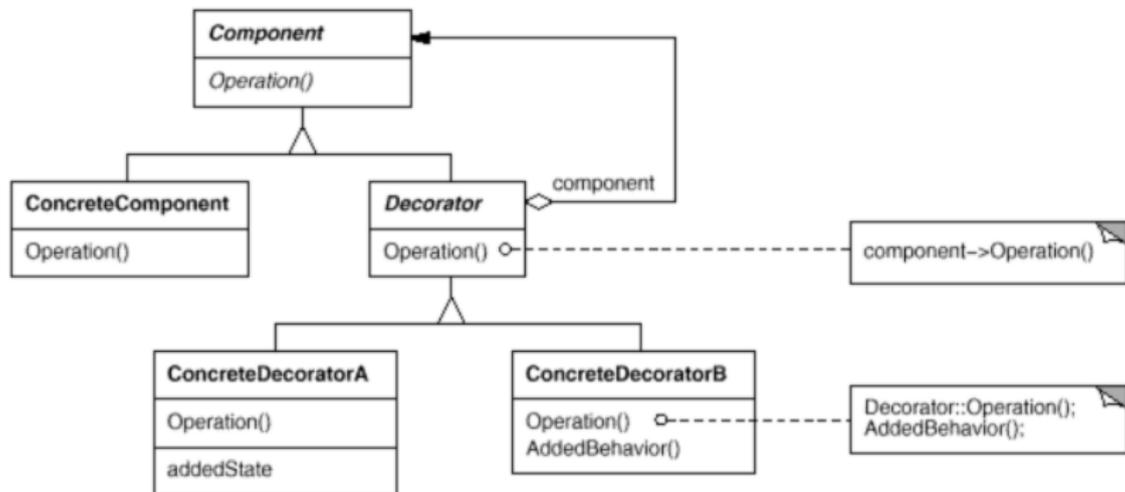
Пример: паттерн Decorator

Динамически добавляет дополнительное поведение объекту.

Декоратор создает список объектов-обертки над другими объектами. Они наследуются от того же самого интерфейса.

Перегрузкой методов можно либо использовать исходные варианты, либо добавлять свою функциональность.

Пример: паттерн Decorator



- ▶ Декоратор имеет тот же интерфейс, что и **Component** (использование декоратора).
- ▶ Декоратор содержит указатель на конкретный **Component** (реализация декоратора).

Пример: паттерн Decorator

```
class TWriterInterface {
public:
    virtual ~TWriterInterface() = default;
    virtual void Write(const std::string& s) = 0;
};

class TStandardWriter : public TWriterInterface {
public:
    virtual ~TStandardWriter() = default;
    virtual void Write(const std::string& s) { std::cout << s
                                                << std::endl; }
};

using TWriterInterfacePtr = std::unique_ptr<TWriterInterface>;

class Decorator : public TWriterInterface {
    TWriterInterfacePtr Interface;
public:
    Decorator(TWriterInterfacePtr ptr) { Interface = std::move(ptr);}
    virtual void Write(const std::string& s) override {
        Interface->Write(s);
    }
};
```

Пример: паттерн Decorator

```
class DecoratorWithBorder : public Decorator {
    std::string Name;
public:
    DecoratorWithBorder(TWriterInterfacePtr ptr, const std::string& n)
        : Decorator(std::move(ptr))
        , Name(n)    {}
    virtual void Write(const std::string& s) override {
        std::cout << "=== " << Name << " ===" << std::endl;
        Decorator::Write(s);
        std::cout << "====" << std::string(Name.size(), '=')
            << "====" << std::endl;
    }
};

class DecoratorWithExclamation : public Decorator {
public:
    DecoratorWithExclamation(TWriterInterfacePtr ptr)
        : Decorator(std::move(ptr)) {}
    virtual void Write(const std::string& s) override {
        std::cout << "ATTENTION!!!" << std::endl;
        Decorator::Write(s);
    }
};
```

Пример: паттерн Decorator

```
int main() {  
    TWriterInterfacePtr writer = std::make_unique<TStandardWriter>();  
    writer->Write("some information");  
}
```

some information

Пример: паттерн Decorator

```
int main() {
    TWriterInterfacePtr writer = std::make_unique<TStandardWriter>();
    TWriterInterfacePtr writer2 =
        std::make_unique<DecoratorWithBorder>(
            std::move(writer), "Magic");
    writer2->Write("some information again");
}
```

```
=== Magic ===
some information again
=====
```

Пример: паттерн Decorator

```
int main() {
    TWriterInterfacePtr writer = std::make_unique<TStandardWriter>();
    writer->Write("some information");
    TWriterInterfacePtr writer2 =
        std::make_unique<DecoratorWithBorder>(
            std::move(writer), "Magic");
    TWriterInterfacePtr writer3 =
        std::make_unique<DecoratorWithExclamation>(std::move(writer2));
    writer3->Write("some information again and again");
}
```

ATTENTION!!!

=== Magic ===

some information again and again

=====

Пример: паттерн Decorator

Feature: возможность кастомизации и конфигурации ожидаемого поведения. Работа начинается с пустым объектом, который имеет базовую функциональность. Затем происходит выбор декораторов, оборачивающих и обогащающих базовый объект.

Пример: паттерн Decorator

Feature: возможность кастомизации и конфигурации ожидаемого поведения. Работа начинается с пустым объектом, который имеет базовую функциональность. Затем происходит выбор декораторов, оборачивающих и обогащающих базовый объект.

Наследование или Декоратор?

- ▶ В случае декоратора проще изменять объекты в run-time.
- ▶ Проще создавать множественные изменения поведений.
- ▶ Если динамически менять поведение объекта не нужно — не нужен и декоратор, наследование может быть проще.

Пример: паттерн Decorator

Feature: возможность кастомизации и конфигурации ожидаемого поведения. Работа начинается с пустым объектом, который имеет базовую функциональность. Затем происходит выбор декораторов, оборачивающих и обогащающих базовый объект.

Наследование или Декоратор?

- ▶ В случае декоратора проще изменять объекты в run-time.
- ▶ Проще создавать множественные изменения поведений.
- ▶ Если динамически менять поведение объекта не нужно — не нужен и декоратор, наследование может быть проще.

Стратегия? Декоратор?

- ▶ Декораторы оборачивают объект снаружи, стратегии же вставляются в него внутрь по неким интерфейсам.
- ▶ Недостаток стратегии: класс должен быть спроектирован с возможностью вставки стратегий.
- ▶ Недостаток декоратора: не всегда желательное смешение публичного интерфейса и интерфейса кастомизации.

Пример: паттерн Observer

Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

- ▶ субъекты (объекты, которые могут изменяться)
- ▶ наблюдатели (объекты, уведомляемые при изменении состояния)

Пример: паттерн Observer

Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

- ▶ субъекты (объекты, которые могут изменяться)
- ▶ наблюдатели (объекты, уведомляемые при изменении состоянии)

Субъекты не заинтересованы в управлении временем жизни своих наблюдателей, но заинтересованы в том, чтобы если наблюдатель был уничтожен, субъекты не пытались к нему обратиться. Тогда так: каждый субъект хранит контейнер указателей `????_ptr` на своих наблюдателей.

Пример: паттерн Observer

```
class Observer {
    std::string name;
public:
    Observer(const std::string& s) : name(s) {}
    void Notify(const std::string& source) { /*...*/ }
};

class Observable {
    std::string name;
public:
    void Subscribe(std::shared_ptr<Observer> observer);
    void Unsubscribe(std::shared_ptr<Observer> observer);
    void Notify();
    Observable(const std::string& s) : name(s) {}
private:
    std::vector<std::weak_ptr<Observer>> observers;
};
```

Пример: паттерн Observer

```
void Observable::Subscribe (std::shared_ptr<Observer> observer) {
    observers.push_back(observer);
}
void Observable::Notify() {
    for (auto wptr: observers) {
        if (!wptr.expired()) {
            auto observer = wptr.lock();
            observer->Notify(this->name);
        }
    }
}
void Observable::Unsubscribe(std::shared_ptr<Observer> observer) {
    observers.erase(
        std::remove_if(
            observers.begin(),
            observers.end(),
            [&](const std::weak_ptr<Observer>& wptr) {
                return wptr.expired() || wptr.lock() == observer;
            }
        ),
        observers.end());
}
```

Пример: паттерн Observer

