Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 8
06.11.2018

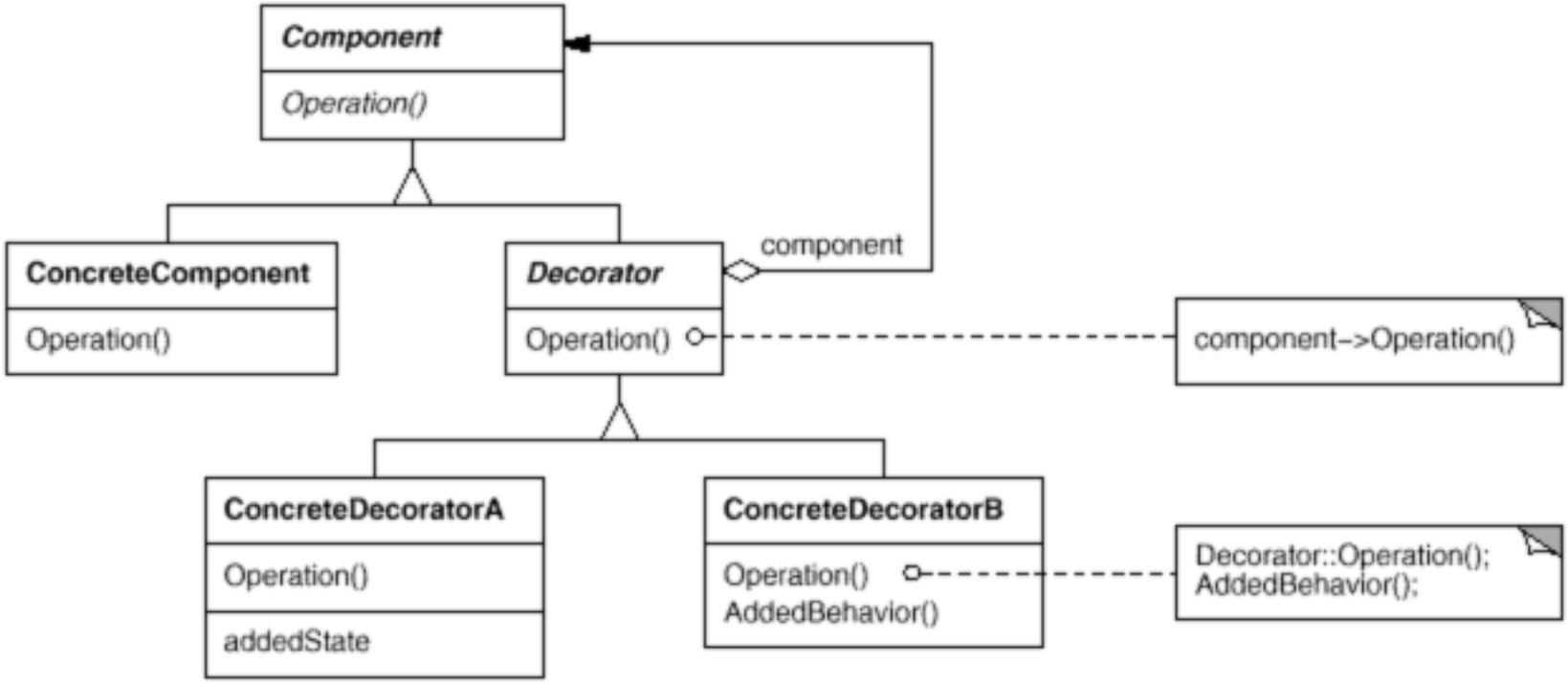
Пример: паттерн Decorator

Динамически добавляет дополнительное поведение объекту.

Декоратор создает список объектов-оберток 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 обогощающих 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 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

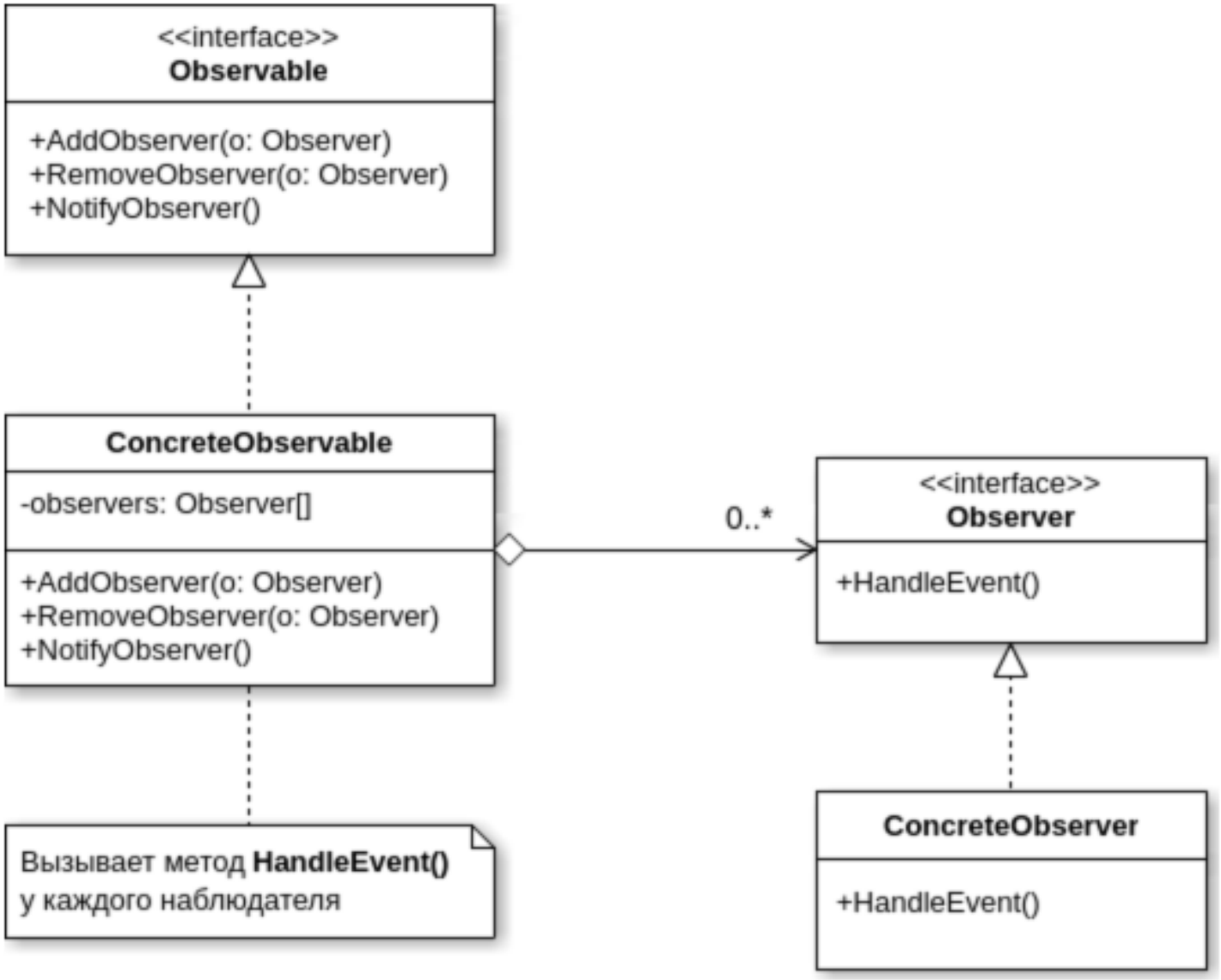
```
class Observer {
    std::string name;
  public:
    Observer(const std::string& s) : name(s) {}
    void Notify(const std::string& source) {/*...*/}
};

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



Пример: Factory method

Паттерн проектирования, основывающийся на наследовании, при котором создание объекта делегируется подклассам, которые реализуют фабричный метод для создания объекта. Создание объектов по внешним параметрам.
Абстрактная фабрика — порождающий паттерн проектирования, позволяющий создавать группы взаимосвязанных или взаимозависимых объектов без указаний их конкретных классов. Во многих реализациях использует шаблон Фабричный метод.

Фабрика

Иерархия объектов:

```
// objects.h

class IObject {
  public:
    virtual void Do() const = 0;
};

class TCustomObject : public IObject {
  public:
    virtual void Do() const override {
        std::cout << "TCustomObject DO " << std::endl;
    }
};

class TSuperObject : public IObject {
  public:
    virtual void Do() const override {
        std::cout << "TSuperObject DO " << std::endl;
    }
};
```

Фабрика

Как можно было бы сделать:

```
class TFactory {
  public:
    virtual IObject* Create(const std::string& name) {
        if (name == "custom object") {
            return new TCustomObject();
        } else if (name == "...")
        ....
        else return nullptr;
    }
};
```

Фабрика

```
//factory.h

class TFactory {
    class TImpl;
    std::unique_ptr<const TImpl> Impl;

  public:
    TFactory();
    ~TFactory();
    std::unique_ptr<IObject> CreateObject(
        const std::string& name
        /*, const TOptions opts*/) const;
    std::vector<std::string> GetAvailableObjects() const;
};
```

Фабрика

```
// factory.cpp

#include "factory.h"
class TFactory::TImpl {

    class ICreator {
      public:
        virtual ~ICreator(){}
        virtual std::unique_ptr<IObject> Create() const = 0;
    };
    using TCreatorPtr = std::shared_ptr<ICreator>;
    using TRegisteredCreators =
        std::map<std::string, TCreatorPtr>;
    TRegisteredCreators RegisteredCreators;
// ...
```

Фабрика

```
class TFactory::TImpl {
// ...
  public:
    template <class TCurrentObject>
    class TCreator : public ICreator{
        std::unique_ptr<IObject> Create() const override{
            return std::make_unique<TCurrentObject>();
        }
    };
// ...
```

Фабрика

```
class TFactory::TImpl {
// ...
  public:
    TImpl() { RegisterAll();}

    template <typename T>
    void RegisterCreator(const std::string& name) {
        RegisteredCreators[name] = std::make_shared<TCreator<T>>();
    }

    void RegisterAll() {
        RegisterCreator<TCustomObject>("custom object");
        RegisterCreator<TSuperObject>("super object");
    }

    std::unique_ptr<IObject> CreateObject(const std::string& n) const {
        auto creator = RegisteredCreators.find(n);
        if (creator == RegisteredCreators.end()) {
            return nullptr;
        }
        return creator->second->Create();
    }
```

Фабрика

```
class TFactory::TImpl {
// ...
  public:
    std::vector<std::string> GetAvailableObjects () const {
        std::vector<std::string> result;
        for (const auto& creatorPair : RegisteredCreators) {
            result.push_back(creatorPair.first);
        }
        return result;
    }
};

std::unique_ptr<IObject> TFactory::CreateObject(const std::string& n) c
    return Impl->CreateObject(n);
}

TFactory::TFactory() : Impl(std::make_unique<TFactory::TImpl>()) {}
TFactory::~TFactory(){}

std::vector<std::string> TFactory::GetAvailableObjects() const {
    return Impl->GetAvailableObjects();
}
```

Фабрика

```
#include "factory.h"

int main() {
    TFactory factory;
    auto objects = factory.GetAvailableObjects();
    for (const auto& obj : objects) {
        std::cout << obj << std::endl;
    }

    for (const auto& objName : {"super object", "custom object"}) {
        factory.CreateObject(objName)->Do();
    }
    return 0;
}
```