

# Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 10

10.11.2017

# 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) const {
    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;
}
```

# Задача

Спроектируйте и реализуйте иерархию классов для генерации случайных чисел, а также фабрику для создания экземпляра класса генератора с заданными параметрами.

```
class TRandomNumberGenerator {  
    public:  
        virtual ~TRandomNumberGenerator();  
        virtual double Generate() const = 0;  
};
```

Необходимо поддерживать три типа распределения: Пуассона, Бернулли и геометрическое. Тип распределения задаётся строковым параметром `type`, который принимает значения из множества `poisson`, `bernoulli`, `geometric`. При создании генератора передаются также параметры распределений.

# Visitor: предпосылки

Зададимся простой иерархией...

```
class TAnimal {  
    public:  
        virtual ~TAnimal() {}  
        virtual void Talk() const = 0;  
        virtual void Move() const = 0;  
};
```

# Visitor: предпосылки

```
using TAnimalPtr = std::shared_ptr<TAnimal>;

class TCat : public TAnimal {
public:
    virtual void Talk() const override{
        std::cout << "meow" << std::endl;
    }
    virtual void Move() const override{
        std::cout << "cat jumps" << std::endl;
    }
};

class TDog : public TAnimal {
public:
    virtual void Talk() const override{
        std::cout << "woof" << std::endl;
    }
    virtual void Move() const override{
        std::cout << "dog moves" << std::endl;
    }
};
```

# Visitor: предпосылки

Простая фабрика по созданию объектов:

```
TAnimalPtr CreateAnimal() {  
    std::string subj;  
    std::cin >> subj;  
    if (subj == "cat") {  
        return static_cast<TAnimalPtr>(new TCat);  
    } else if (subj == "dog") {  
        return static_cast<TAnimalPtr>(new TDog);  
    }  
    return nullptr;  
}
```

Обычный динамический полиморфизм:

```
int main() {  
    TAnimalPtr animal = CreateAnimal();  
    animal->Talk();  
    animal->Move();  
    return 0;  
}
```

# Visitor: операции

Вынесем операции:

```
class TOperation {
public:
    virtual ~TOperation() {}
    virtual void Apply(const TAnimal &animal) const = 0;
};

using TOperationPtr = std::shared_ptr<TOperation>;

class TTalkOperation : public TOperation {
public:
    virtual void Apply(const TAnimal &animal) const override {}
};

class TMoveOperation : public TOperation {
public:
    virtual void Apply(const TAnimal &animal) const override {}
};
```

# Visitor: операции

Фабрика операций:

```
TOperationPtr CreateOperation() {  
    std::string op;  
    std::cin >> op;  
    if (op == "talk") {  
        return static_cast<TOperationPtr>(new TTalkOperation);  
    } else if (op == "move") {  
        return static_cast<TOperationPtr>(new TMoveOperation);  
    }  
}
```

А теперь хочется делать так:

```
TAnimalPtr animal = CreateAnimal();  
TOperationPtr operation = CreateOperation();  
// (animal, operation) -> Apply();   ???
```



# Visitor: двойная диспетчеризация

Можно сделать так:

```
class TTalkOperation : public TOperation {
public:
    virtual void Aplly(const TAnimal &animal) const {
        if (const TCat* cat = dynamic_cast<const TCat*>(&animal) {
            std::cout << "meow" << std::endl;
        } else if (const TDog* dog = dynamic_cast<const TDog*>(&animal)
            std::cout << "woof" << std::endl;
        }
    }
};
```

Но:

- ▶ много дублирования,
- ▶ важен порядок условий.

# Visitor

- ▶ В основной иерархии только виртуальный метод `Accept`; Переопределяется в наследниках

```
TCat::Accept(op) { op.Visit(*this); }
```

- ▶ В `Operation(Visitor)` определяется набор методов `Visit`, они все виртуальные и их можно переопределять (`Visit(Animal)`, `Visit(Cat)`, `Visit(Dog)`).

# Visitor

```
class TOperation {
public:
    virtual ~TOperation() {}
    virtual void Visit(const TAnimal &animal) const = 0;
    virtual void Visit(const TCat &cat) const;
    virtual void Visit(const TDog &dog) const;
};

void TOperation::Visit(const TCat &cat) const {
    Visit(static_cast<const TAnimal&>(cat));
}

void TOperation::Visit(const TDog &dog) const {
    Visit(static_cast<const TAnimal&>(dog));
}
```

# Visitor

```
class TTalkOperation : public TOperation {
public:
    virtual void Visit(const TAnimal &animal) const override {
        std::cout << "unknown operation" << std::endl;
    }
    virtual void Visit(const TCat &cat) const override{
        std::cout << "meow" << std::endl;
    }
    virtual void Visit(const TDog &dog) const override{
        std::cout << "woof" << std::endl;
    }
};
```

```
class TMoveOperation : public TOperation {
public:
    virtual void Visit(const TAnimal &animal) const override {
        std::cout << "unknown operation" << std::endl;
    }
    // ...
};
```

# Visitor

```
class TCat : public TAnimal,
             public std::enable_shared_from_this<TCat> {
public:
    virtual void Accept(const TOperation& operation) const override{
        std::shared_ptr<TCat> p{shared_from_this()};
        operation.Visit(p);
    }
};

class TDog : public TAnimal,
             public std::enable_shared_from_this<TDog> {
public:
    virtual void Accept(const TOperation& operation) const override{
        std::shared_ptr<TDog> p{shared_from_this()};
        operation.Visit(p);
    }
};
```

# Visitor

Использование:

```
int main() {  
    TAnimalPtr animal = CreateAnimal();  
    TOperationPtr operation = CreateOperation();  
    animal->Accept(*operation);  
    return 0;  
}
```

**Двойная диспетчеризация:** при вызове `animal->Accept` находится правильный класс `TAnimal` (механизм виртуальных функций), а затем при вызове `operation->visit(*this)` управление передается правильному visitor'у.

# Visitor

Дублирование в Ассерт наследниках можно устранить:

```
template <typename T>
class TFinalAnimal : public T {
public:
    virtual void Accept(const TOperation& operation) const {
        operation.Visit(*this);
    }
};
```

