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) 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;
}
```

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 поддержать 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 Aplly(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 Accept наследниках можно устранить:

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

Проблема перегрузки и универсальных ссылок

Параметр-универсальная ссылка обычно обеспечивает точное соответствие для всего, что бы ни было передано:

```
using TStringSet = std::set<std::string>;
template <typename T>
void Do(TStringSet& strings, T&& str) {
    std::cout << str << std::endl;
    strings.emplace(std::forward<T>(str));
}
void Do(TStringSet& strings, int x) {
    std::cout << x << std::endl;
    strings.emplace(std::to_string(x));
}
```

Ломается код:

```
    short x = 2;
    Do(strings, x);
```

Проблема перегрузки и универсальных ссылок

Добавим новую функцию, которая вызывает две другие:

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
  DoImpl(
    strings,
    std::forward<T>(x),
    std::is_integral<T>()
  )
}
```

Какая проблема?

Проблема перегрузки и универсальных ссылок

Добавим новую функцию, которая вызывает две другие:

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
  DoImpl(
    strings,
    std::forward<T>(x),
    std::is_integral<T>()
  )
}
```

Какая проблема? `std::is_integral<int&>` имеет ложное значение.

Проблема перегрузки и универсальных ссылок

Добавим новую функцию, которая вызывает две другие:

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
  DoImpl(
    strings,
    std::forward<T>(x),
    std::is_integral<T>()
  )
}
```

Какая проблема? `std::is_integral<int&>` имеет ложное значение.

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
  DoImpl(
    strings,
    std::forward<T>(x),
    std::is_integral<typename std::remove_reference<T>::type>()
  )
}
```

Проблема перегрузки и универсальных ссылок

Меньше символов с C++14:

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
  DoImpl(
    strings,
    std::forward<T>(x),
    std::is_integral<std::remove_reference_t<T>>()
  )
}
```

Проблема перегрузки и универсальных ссылок

Меньше символов с C++14:

```
template <typename T>
void Do(TStringSet& strings, T&& x) {
  DoImpl(
    strings,
    std::forward<T>(x),
    std::is_integral<std::remove_reference_t<T>>()
  )
}
```

Диспетчеризация дескрипторов: вызов перегруженных функций «диспетчеризует» передачу работы правильной функции путем создания нужного объекта дескриптора.

Проблема перегрузки и универсальных ссылок

Две перегрузки для DoImpl:

```
template <typename T>
void DoImpl(TStringSet& strings, T&& str, std::false_type /*f*/) {
    std::cout << str << std::endl;
    strings.emplace(std::forward<T>(str));
}
void DoImpl(TStringSet& strings, int x, std::true_type /*t*/) {
    std::cout << x << std::endl;
    strings.emplace(std::to_string(x));
}
```

Диспетчеризация дескрипторов

- Решаем проблемы перегрузку и оставляем универсальные ссылки,
- неперегружаемая функция диспетчеризации Do принимает параметр, являющийся универсальной ссылкой,
- перегружаемая impl-функция имеет параметр дескриптора, который спроектирован так, что не существует более одной перегрузки,
- вызов нужной функции определяется дескриптором.

Вопрос

Что напечатает программа?

```
void f(unsigned i) {
    std::cout << "f(int)" << std::endl;
}

template <typename T>
void f(const T& i) {
    std::cout << "template f" << std::endl;
}

int main() {
    f(3);
}
```

Вопрос

Что напечатает программа?

```
void f(unsigned i) {
    std::cout << "f(int)" << std::endl;
}

template <typename T>
void f(const T& i) {
    std::cout << "template f" << std::endl;
}

int main() {
    f(3);
}
```

```
template f
```

SFINAE: Substitution Failure Is Not An Error

А если так?

```
unsigned f(unsigned i) {
    std::cout << "f(int)" << std::endl;
    return i + 1;
}

template <typename T>
typename T::value_type f(const T& i) {
    std::cout << "template f" << std::endl;
    return i + 1;
}

int main() {
    f(3);
}
```

SFINAE: Substitution Failure Is Not An Error

А если так?

```
unsigned f(unsigned i) {
    std::cout << "f(int)" << std::endl;
    return i + 1;
}

template <typename T>
typename T::value_type f(const T& i) {
    std::cout << "template f" << std::endl;
    return i + 1;
}

int main() {
    f(3);
}
```

`f(int)`

Подстановка `int::value_type f(const int i)` невалидна.

std::enable_if

std::enable_if: если передано true, то в структуре присутствует тип type (второй параметр), если передано false, то никакого type нет.

```
template<bool B, class T = void >
struct enable_if;
```

C++14 добавляет алиас:

```
template <bool B, typename T = void>
using enable_if_t = typename enable_if<B, T>::type;
```

std::enable_if: пример

Использование в возвращаемом значении:

```
template <typename T>
std::enable_if_t<std::is_floating_point<T>::value, T> Do(const T& x) {
    std::cout << "float type" << std::endl;
    return x;
}

template <typename T>
std::enable_if_t<std::is_integral<T>::value, T> Do(const T& x) {
    std::cout << "integral type" << std::endl;
    return x;
}
```

std::enable_if: пример

В классах:

```
template<typename T, typename = void>
class A;

template<class T>
class A<T, std::enable_if_t<std::is_integral<T>::value>> {
};
```

- `std::enable_if_t<std::is_integral<int>::value>` → `void`
- `std::enable_if_t<std::is_integral<float>::value>` → ошибка компиляции

Перегрузка и универсальные ссылки – 2

```
#include <string>

class A {
  private:
    std::string text;
  public:
    template <typename T>
    explicit A(T&& str) : text(std::forward<T>(str)) {}
    explicit A(int x) : text(std::to_string(x)) {}
};

int main() {
    A x("123");
    auto copyX(x); // error!
}
```

Перегрузка и универсальные ссылки – 2

Хочется написать так в шаблонный конструктор:

```
template <typename T, typename = std::enable_if_t<CONDITION>>
explicit A(T&& str) : text(std::forward<T>(str)) {}
```

`CONDITION` — тип T не является классом A.

Перегрузка и универсальные ссылки – 2

Хочется написать так в шаблонный конструктор:

```
template <typename T, typename = std::enable_if_t<CONDITION>>
explicit A(T&& str) : text(std::forward<T>(str)) {}
```

`CONDITION` — тип `T` не является классом `A`.

$\downarrow\downarrow\downarrow$

```
!std::is_same<A, T>::value
```

Перегрузка и универсальные ссылки – 2

Хочется написать так в шаблонный конструктор:

```
template <typename T, typename = std::enable_if_t<CONDITION>>
explicit A(T&& str) : text(std::forward<T>(str)) {}
```

`CONDITION` — тип `T` не является классом `A`.

`!std::is_same<A, T>::value`

Тип `T` может быть выведен как lvalue-ссылка `A&`, а это не `A`.

Перегрузка и универсальные ссылки – 2

Хочется написать так в шаблонный конструктор:

```
template <typename T, typename = std::enable_if_t<CONDITION>>
explicit A(T&& str) : text(std::forward<T>(str)) {}
```

`CONDITION` — тип `T` не является классом `A`.

$\downarrow\downarrow\downarrow$

`!std::is_same<A, T>::value`

Тип `T` может быть выведен как lvalue-ссылка `A&`, а это не `A`.

$\downarrow\downarrow\downarrow$

`!std::is_same<A, std::decay_t<T>>::value`

Перегрузка и универсальные ссылки – 2

Всё ок, но есть проблема:

```
class B: public A {
public:
    B(const B& other) : A(other) {...}
    // ...
};
```

Перегрузка и универсальные ссылки – 2

Всё ок, но есть проблема:

```
class B: public A {
public:
    B(const B& other) : A(other) {...}
    // ...
};
```

Воспольуемся этим:

```
std::is_base_of<T1,T2>::value; // истинно, если
                               // T2 - производный от T1

std::is_base_of<int,int>::value; // false
std::is_base_of<A,A>::value;     // true
```

Перегрузка и универсальные ссылки – 2

```
template <
    typename T,
    typename = std::enable_if_t<
        !std::is_base_of<A, std::decay_t<T>>::value
    >
>
explicit A(T&& str) : text(std::forward<T>(str)) {}
```

Но есть еще вторая перегрузка, которую нам нужно вызывать для целочисленных типов:

```
explicit A(int x) : text(std::to_string(x)) {}
```

Перегрузка и универсальные ссылки – 2

Отключаем шаблонный конструктор для обработки целочисленных аргументов:

```
class A {
  private:
    std::string text;
  public:
    template <
        typename T,
        typename = std::enable_if_t<
            !std::is_base_of<A, std::decay_t<T>>::value
            &&
            !std::is_integral<std::remove_reference_t<T>>::value
        >
    >
    explicit A(T&& str) : text(std::forward<T>(str)) {}
    explicit A(int x) : text(std::to_string(x)) {}
};
```

Замечание

`is_arithmetic<T>`: если T является арифметическим типом (целочисленным или в формате с плавающей запятой), то имеется константа-член `value`, которая будет равна `true`. Для всех остальных типов `value` будет равна `false`.

```
std::is_arithmetic<int*>::value;        // false
std::is_arithmetic<int const>::value;   // true
std::is_arithmetic<int&>::value;        // false
```