Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 8
27.10.2017

Идиома CRTP

The curiously recurring template pattern. Класс отнаследован от шаблонного класса, в котором наследник – аргумент шаблона:

```
template <typename Derived>
class CuriousBase {
    // ...
};
class Curious : public CuriousBase<Curious> {
    // ..
};
```

Coplien, James O. (February 1995).

Идиома CRTP

The curiously recurring template pattern. Класс отнаследован от шаблонного класса, в котором наследник – аргумент шаблона:

```
template <typename Derived>
class CuriousBase {
    // ...
};
class Curious : public CuriousBase<Curious> {
    // ..
};
```

Coplien, James O. (February 1995).

Пример: `enable_shared_from_this`.

Идиома CRTP

Ограничиваем число объектов класса.

```
#include <stdexcept>
template <typename T, size_t maxN>
class LimitedInstances {
    static size_t counter;
  protected:
    LimitedInstances() {
        if (counter >= maxN) {
            throw std::logic_error("too many instances");
        }
        ++counter;
    }
    ~LimitedInstances() {
        --counter;
    }
};

template <typename T, size_t maxN>
size_t LimitedInstances<T, maxN>::counter(0);
```

Идиома CRTP

```
class oneInst: public LimitedInstances<oneInst, 1> {};
class twoInst: public LimitedInstances<twoInst, 2> {};

int main() {
    oneInst obj;
    try {
        oneInst();
    } catch (std::logic_error &e) {
        std::cerr << "Caught: " << e.what() << std::endl;
    }

    twoInst obj1;
    twoInst obj2;
    try {
        twoInst();
    } catch (std::logic_error &e) {
        std::cerr << "Caught: " << e.what() << std::endl;
    }
};
```

Идиома CRTP

- Часто заменяют динамический полиморфизм через статический: в базовом классе вызываем методы класса, которым он параметризован при инстанциировании

Идиома PImpl

Pointer to implementation.
Метод, при котором члены-данные класса заменяются указателем на класс реализации с этими данными.
a.h:

```
#include "myitems.h"
class A {
    TMyItem item1, item2;
  public:
    A();
    // ...
};
```

Break compilation dependencies!

a.h:

```
class A {
    struct Impl;
    Impl *pImpl;
  public:
    A();
    // ...
};
```

---

a.cpp:

```
#include "a.h"
#include "myitems.h"
struct A::Impl {
    TMyItem item1, item2;
};

A::A() : pImpl(new Impl) {}
A::~A() {delete pImpl;}
```

Идиома PImpl: C++11

a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
  public:
    A();
    // ...
};
```

---

a.cpp:

```
#include "a.h"
#include "myitems.h"
struct A::Impl {
    TMyItem item1, item2;
};

A::A() : pImpl(std::make_unique<A::Impl>()) {}
```

Идиома PImpl: C++11

a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
  public:
    A();
    // ...
};
```

---

a.cpp:

```
#include "a.h"
#include "myitems.h"
struct A::Impl {
    TMyItem item1, item2;
};

A::A() : pImpl(std::make_unique<A::Impl>()) {}
```

---

main.cpp:

```
#include "a.h"
A a;  // !error
```

Идиома PImpl

Нужно обеспечить полноту в точке уничтожения `std::unique_ptr<A::Impl>`.

a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
  public:
    A();
    ~A();
    // ...
};
```

---

a.cpp:

```
~A::A() = default;
```

Идиома PImpl

Нужны перемещающие функции:
a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
  public:
    A();
    ~A();
    A(A&& other) = default;
    A& operator=(const A& other) = default;
    // ...
};
```

И снова та же проблема!

Идиома PImpl

Объявляем в заголовочном файле, реализуем в файле реализации:

a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
  public:
    A();
    ~A();
    A(A&& other);
    A& operator=(const A& other);
    // ...
};
```

---

a.cpp:

```
A::A(A&& other) = default;
A& A::operator=(const A& other) = default;
```

Идиома PImpl

Потребуются копирующие операции:
a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
  public:
    A();
    ~A();
    A(A&& other);
    A& operator=(const A& other);
    A(const A& other);
    A& operator=(const A& other);
    // ...
};
```

Идиома PImpl

a.cpp:

```
A::A(const A& other) : pImpl(nullptr) {
    if (other.pImpl) {
        pImpl = std::make_unique<Impl>(*other.Impl);
    }
}

A& A::operator=(const A& other) {
    if (!other.pImpl) {
        pImpl.reset();
    } else if (!pImpl) {
        pImpl = std::make_unique<Impl>(*other.Impl);
    } else {
        *pImpl = *other.pImpl;
    }
    return *this;
}
```

Идиома PImpl

a.cpp:

```
A::A(const A& other) : pImpl(nullptr) {
    if (other.pImpl) {
        pImpl = std::make_unique<Impl>(*other.Impl);
    }
}

A& A::operator=(const A& other) {
    if (!other.pImpl) {
        pImpl.reset();
    } else if (!pImpl) {
        pImpl = std::make_unique<Impl>(*other.Impl);
    } else {
        *pImpl = *other.pImpl;
    }
    return *this;
}
```

В случае `std::shared_ptr` всё проще!

Паттерны проектирования

- Способ построения кода для решения часто встречающихся проблем проектирования
- successful stories
- Готовые абстракции для решения классов проблем + унификация деталей и названий
- Не нужно их употреблять везде, не нужно им строго следовать

Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования.

Паттерны проектирования

- Способ построения кода для решения часто встречающихся проблем проектирования
- successful stories
- Готовые абстракции для решения классов проблем + унификация деталей и названий
- Не нужно их употреблять везде, не нужно им строго следовать

Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. [contains a lot of «ancient» C++ code]

Пример: паттерн Bridge

Цель: разделить абстракцию и реализацию на две отдельные иерархии классов так, что их можно изменять независимо друг от друга.

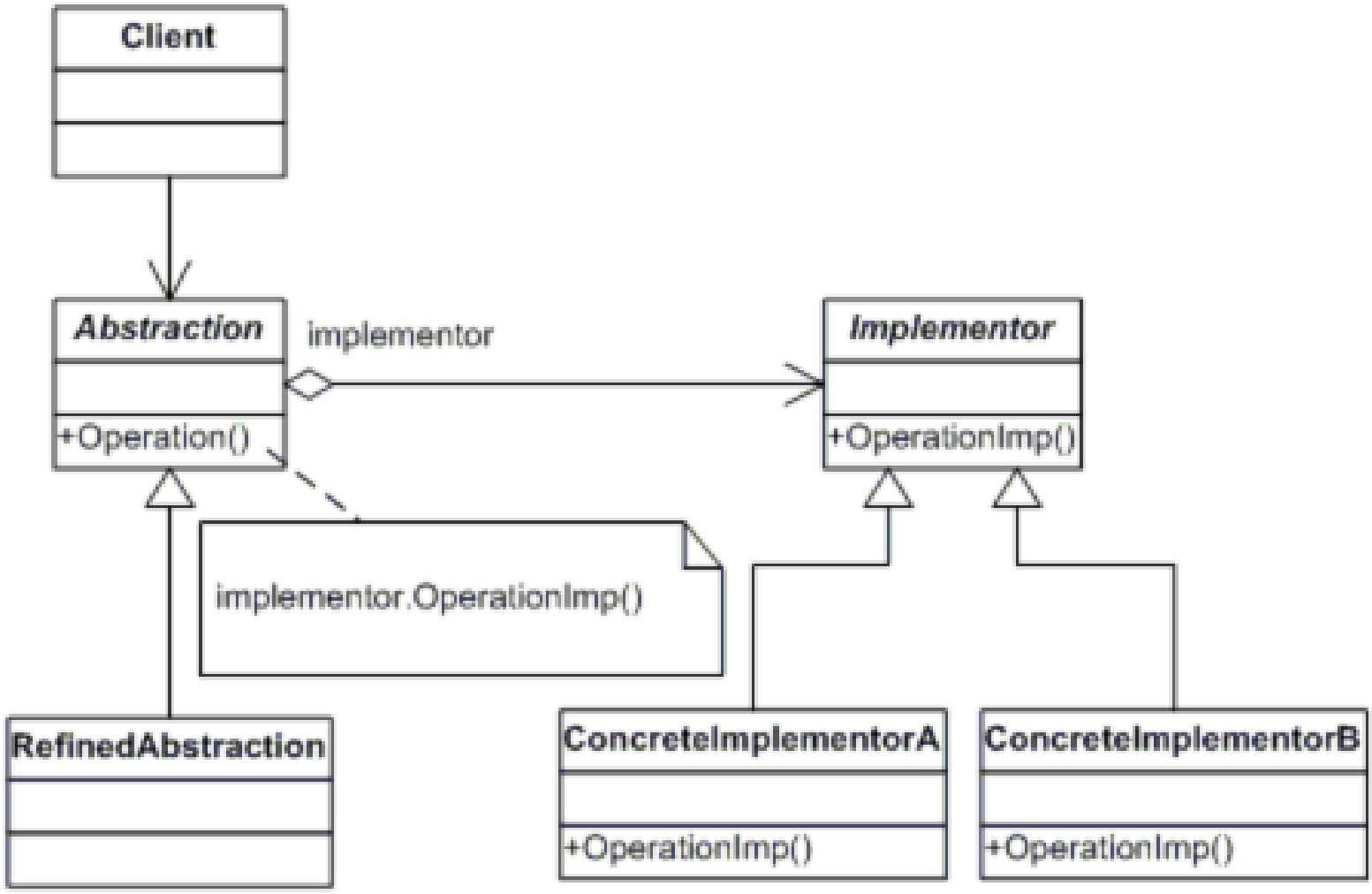
Почему не наследование: наследование жестко привязывает реализацию к абстракции. Это затрудняет расширение и повторное использование абстракции и ее реализации.

Пример: паттерн Bridge

Цель: разделить абстракцию и реализацию на две отдельные иерархии классов так, что их можно изменять независимо друг от друга.
Почему не наследование: наследование жестко привязывает реализацию к абстракции. Это затрудняет расширение и повторное использование абстракции и ее реализации.

Первая иерархия определяет интерфейс абстракции, доступный пользователю. Основной класс содержит указатель на реализацию `pimpl`, который используется для перенаправления пользовательских запросов в неё.
Все детали реализации, связанные с какими-либо особенностями находятся во второй иерархии.

Пример: паттерн Bridge



Abstraction перенаправляет объекту Implementation запросы клиента.

Пример: паттерн Bridge

Когда: когда нужно часто изменять реализацию какого-нибудь метода с сохранением API

Когда: когда используется постоянно изменяющаяся внешняя библиотека

Когда: когда нужно добиться разделения ответственности между классами

Пример: паттерн Bridge

В чем отличие от PIMPL?

- PIMPL — способ скрыть реализацию, в основном для того, чтобы убрать зависимости
- Bridge — поддержка множественных реализаций, а в PIMPL обычно не изменяется реализация, это отдельно компилируемый класс
- PIMPL — идиома проектирования на уровне файлов с кодом, Bridge — паттерн объектно-ориентированного проектирования.

Пример: паттерн Command

Инкапцулирует запрос в виде объекта, делая возможной параметризацию клиентских объектов с другими запросами, организацию очереди или регистрацию запросов, а также поддержку отмены операций.

Пример: паттерн Command

```
struct talk {
    void operator()(){
        std::cout << "croak!" << std::endl;
    }
};

struct walk {
    void operator()() {
        std::cout << "im walking" << std::endl;
    }
};

struct jump {
    void operator()() {
        std::cout << "jump" << std::endl;
    }
};
```

Пример: паттерн Command

```
void dofunc(std::function<void()> f) {
    f();
}

int main() {
    dofunc(talk{});
    dofunc(walk{});
    auto f = jump();
    dofunc(f);
}
```

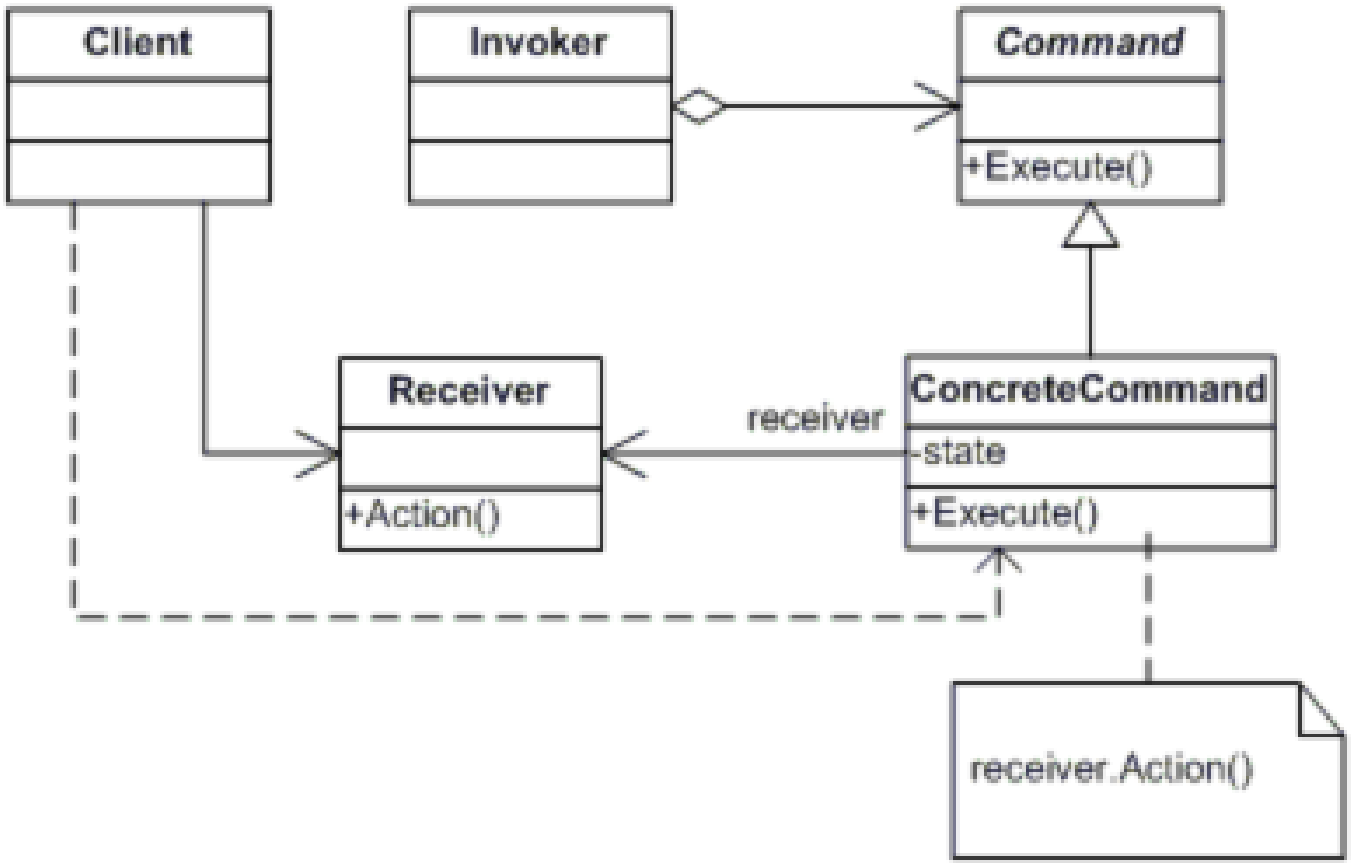
Пример: паттерн Command

```
void dofunc(std::function<void()> f) {
    f();
}

int main() {
    dofunc(talk{});
    dofunc(walk{});
    auto f = jump();
    dofunc(f);
}
```

Если все действия пользователя в программе реализованы в виде командных объектов, программа может сохранить стек последних выполненных команд.

Пример: паттерн Command



Client — среда генерации комманд; Receiver — знает, как провести операцию, связанную с командной; Command — инкапцуляция действия; Invoker — последующие действия с командой или пулом команд.

Пример: паттерн Command

- Undo-Redo
- Организация очереди при многопоточной обработке;
- Транзакционные алгоритмы - регистрация событий и восстановление после сбоя;

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 $\rightarrow$ 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 скрытие 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.