Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 7
30.10.2018

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]

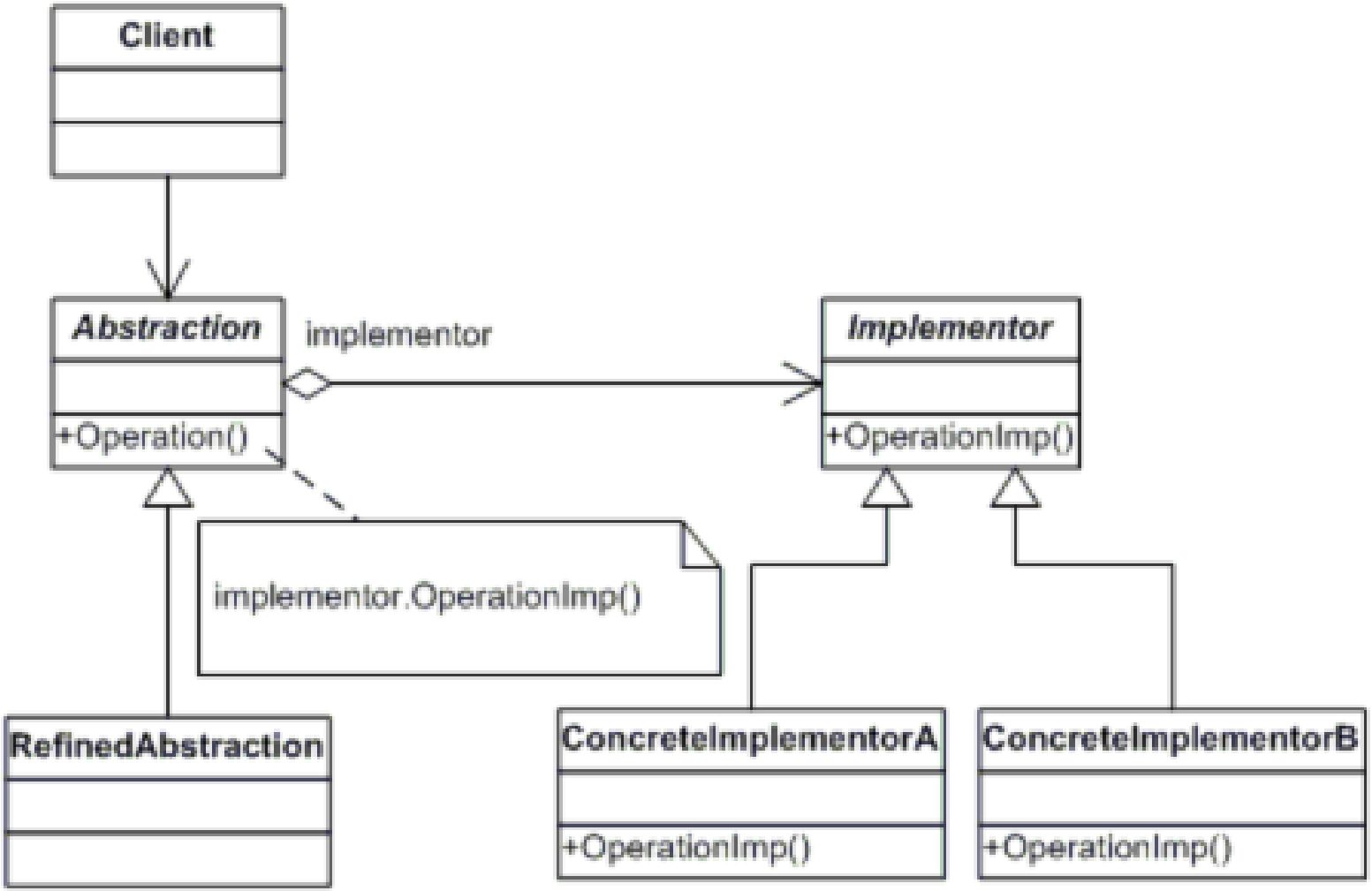
Ф. Пикус Идиомы и паттерны проектирования в современном C++.

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

Инкапсулирует 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;
    }
};
```

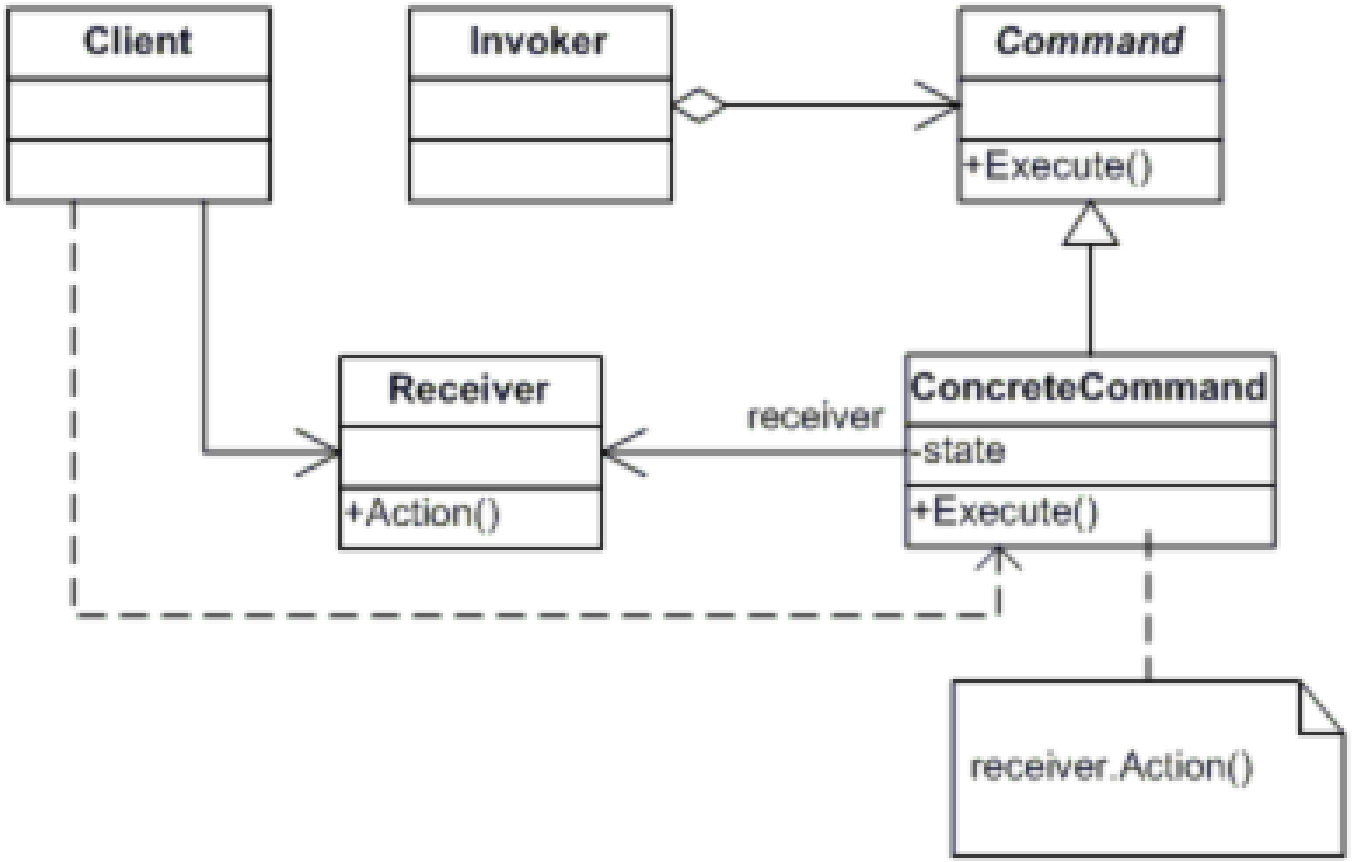
Пример: паттерн Command

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

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

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 → 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.

Когда можно использовать:

- Физическая причина для единственного объекта (физическая величина, автомобиль для программы им управляющей, ...)
- Глобальные объекты из проектных соображений (диспетчеры ресурсов, ...)

Реализация через local static объекты потокобезопасная.

Пример: паттерн Singleton

```
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;
```

Утечки памяти!

Пример: паттерн Singleton

Синглтон С. Мейерса.

```
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;
  }
};
```

Единственный способ получить доступ — обратиться к `Singleton::Instance()`.

Singleton + PImpl

a.h:

```
struct SingletonImpl;
class Singleton {
 public:
  // api
 private:
  static SingletonImpl& impl();
};
```

a.cpp:

```
struct SingletonImpl{
    SingletonImpl() {...}
};
SingletonImpl& Singleton::impl() {
    static SingletonImpl 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.

Пример: паттерн Strategy

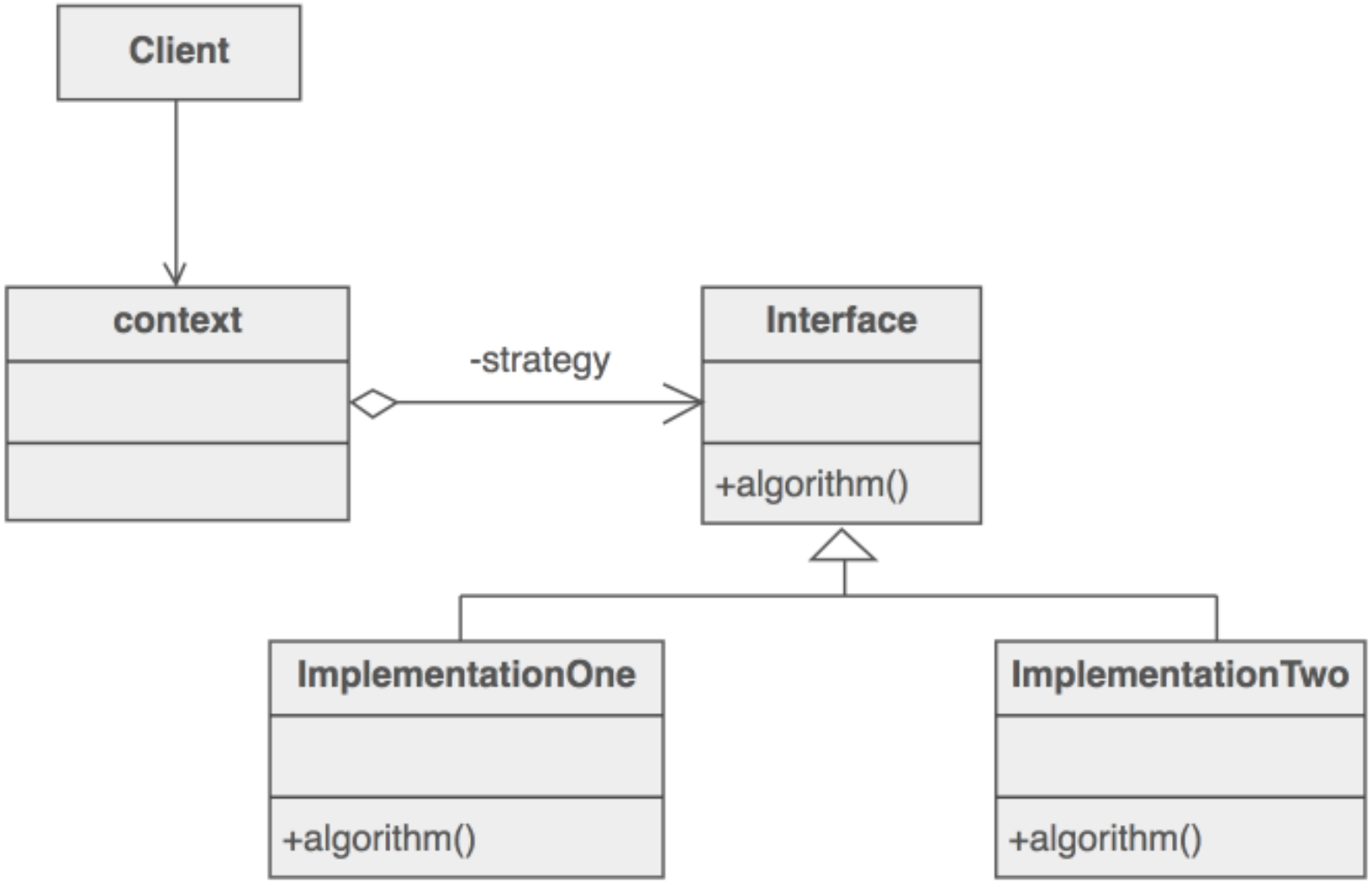
Паттерн, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости.

- 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();
}
```