Поведенческие шаблоны

Command, Iterator, Observer, Chain of Responsibility

Иван Поляков
Разработчик Go/C++ в инфраструктуре поиска в Авито

Проверка связи

Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включён звук
- обновите страницу вебинара (или закройте страницу и заново присоединитесь к вебинару)
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти

Поставьте в чат:

\+ если меня видно и слышно

— если нет

Иван Поляков

О спикере:

- Разработчик Go/C++ в инфраструктуре поиска в Авито
- 5 лет работал в Nexign, писал real time сервисы для телекома Мегафона на C++

Вспоминаем прошлое занятие

Вопрос: зачем нужен паттерн адаптер?

Вспоминаем прошлое занятие

Вопрос: зачем нужен паттерн адаптер?

Ответ: чтобы использовать вместе объекты с несовместимым интерфейсом

Вспоминаем прошлое занятие

Вопрос: зачем нужен паттерн декоратор?

Вспоминаем прошлое занятие

Вопрос: зачем нужен паттерн декоратор?

Ответ: чтобы добавлять поведение объектам, не меняя интерфейс

Вспоминаем прошлое занятие

Вопрос: зачем нужен паттерн заместитель?

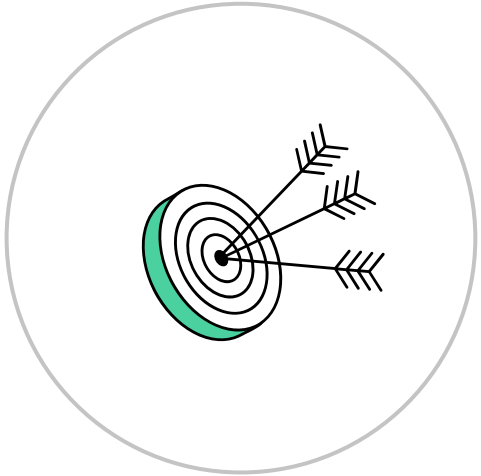
Вспоминаем прошлое занятие

Вопрос: зачем нужен паттерн заместитель?

Ответ: чтобы перехватывать момент вызова оригинального объекта и добавлять свою логику

Цели занятия

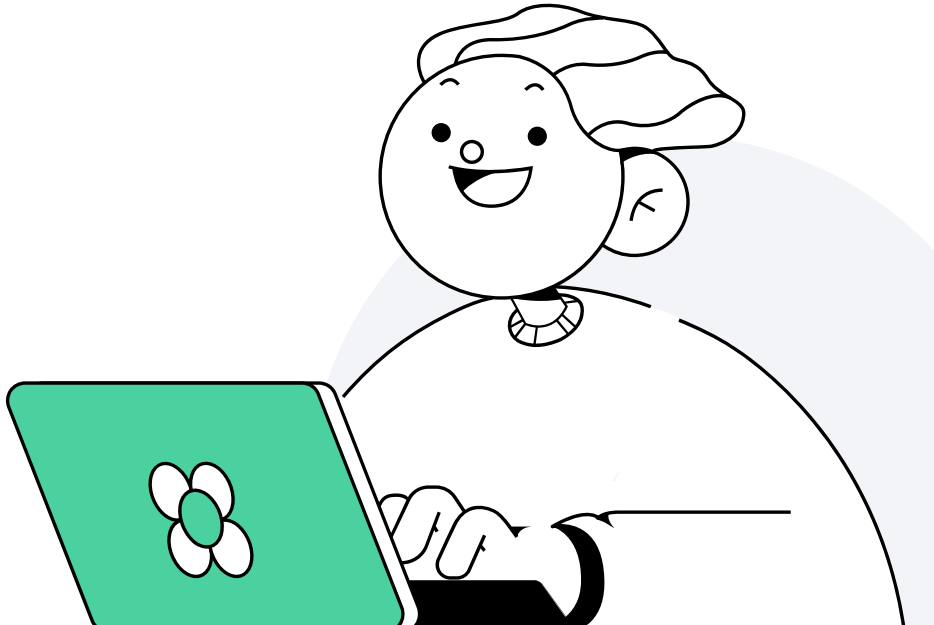
- Узнаем, что такое поведенческие шаблоны проектирования
- Перечислим поведенческие шаблоны проектирования
- Разберём основные поведенческие шаблоны проектирования



План занятия

1. Поведенческие шаблоны проектирования
2. Command
3. Iterator
4. Observer
5. Chain of Responsibility
6. Итоги
7. Домашнее задание

*Нажми на нужный раздел для перехода

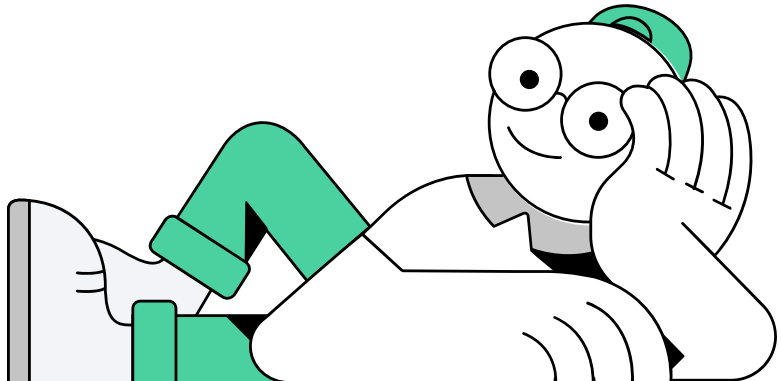


Поведенческие шаблоны проектирования

1

Поведенческие шаблоны проектирования

Определяют алгоритмы и способы реализации взаимодействия различных объектов и классов.



Список поведенческих шаблонов

1. **Chain of responsibility**
2. **Command**
3. Interpreter
4. **Iterator**
5. Mediator
6. Memento
7. **Observer**
8. State
9. Strategy
10. Template Method
11. Visitor

Источник

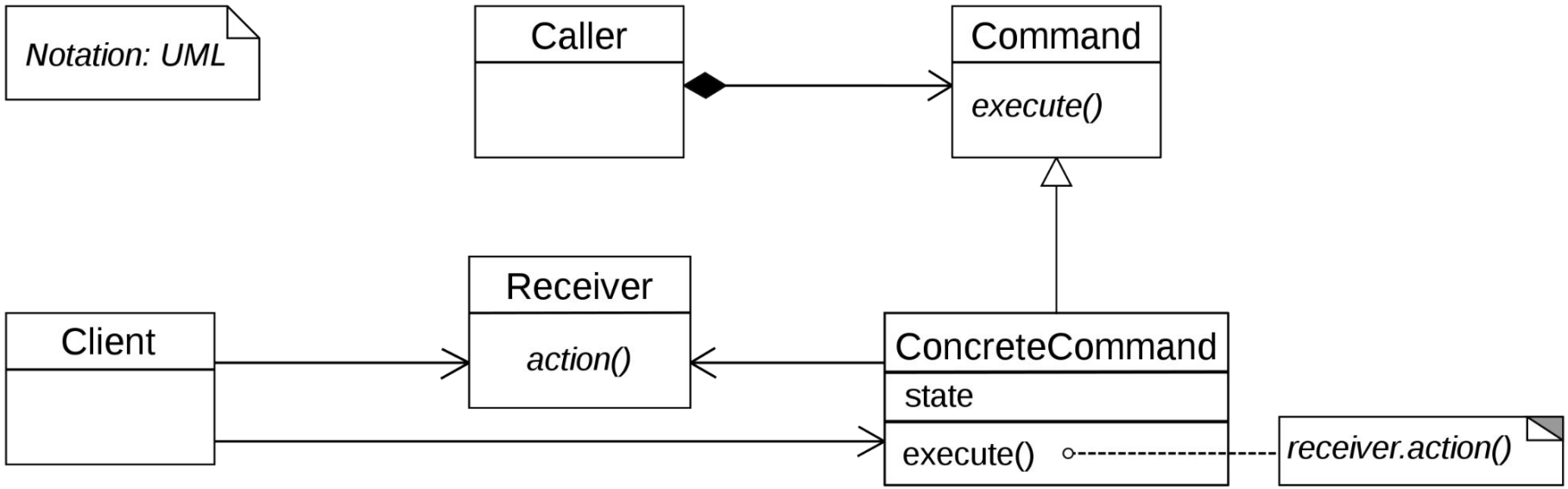
Command

2

Паттерн команда представляет собой действие**.**
Объект команды заключает в себе само действие и его параметры.

Описание паттерна команда

UML-диаграмма



Пример

Рассмотрим простой пример:

```
class Command {
public:
    virtual ~Command() = default;

    virtual void execute() = 0;
};
```

Основная часть объекта команды — метод для выполнения действия.

Применение

Паттерн команда имеет широкое распространение.

Например:

- Для выполнения задач в очереди пула потоков
- Для передачи действий в программе по сети
- В качестве реализации транзакций, например, в базах данных
- Для реализации стека пользовательских действий с возможностью отмены
- Для реализации элементов пользовательских интерфейсов, например, кнопок

Объект команды с возможностью отмены

Допустим, в программе есть очередь задач для распределения их на разные потоки.

В какой-то момент может возникнуть необходимость отменить выполнение задачи.

В этом случае, если задача не начала выполняться, метод cancel может выбросить задачу из очереди.

```
class Task {
public:
    virtual ~Task() = default;

    virtual void execute() = 0;
    virtual void cancel() = 0;
};
```

Объект команды с отменой действия

Зачастую в редакторах текста пользовательский интерфейс позволяет отменить предыдущую операцию.

Чтобы это было возможно, нужен стек команд и в каждой команде возможность отменить её действие.

```
class EditorCommand {
public:
    virtual ~EditorCommand() = default;

    virtual void do() = 0;
    virtual void undo() = 0;
};
```

Пример реализации

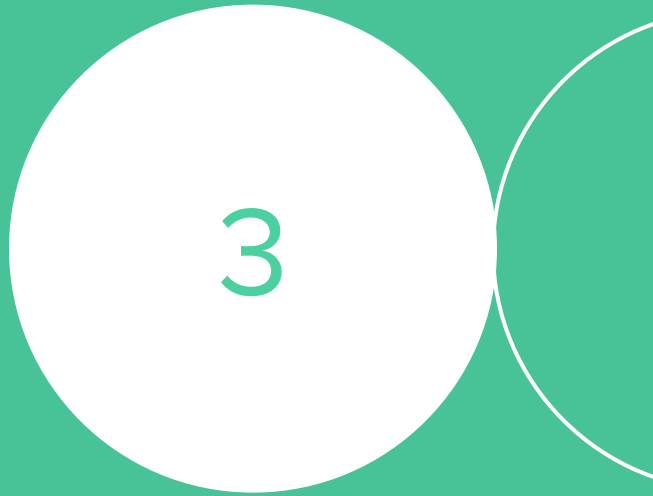
```
class Command {
public:
    virtual ~Command() = default;

    void apply(GameField* game_field) = 0;
};

class PauseCommand : public Command {  // Останавливаем или возобновляем игру в зависимости от параметра pause
public:
    explicit PauseCommand(bool pause) : pause_(pause) {}

    void apply(GameField* game_field) override {
        game_field->setPause(pause_);
    }

private:
    bool pause_ = false;
};

class AddUnitCommand : public Command {  // Добавляем нового юнита на игровое поле по координатам x, y
public:
    AddUnitCommand(int x, int y) : x_(x), y_(y) {}

    void apply(GameField* game_field) override {
        game_field->addUnit(x_, y_);
    }

private:
    int x_ = 0;
    int y_ = 0;
};
```

Комбинация

Можно комбинировать несколько команд в одну:

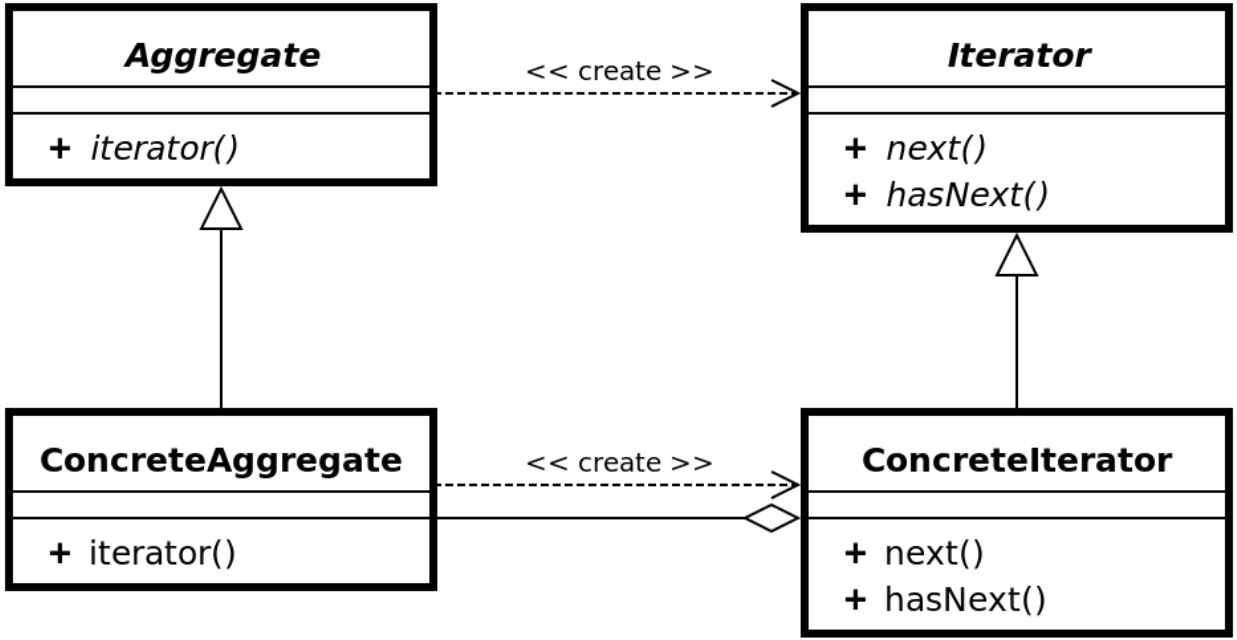
```
class ComplexCommand : public Command {
public:
    explicit ComplexCommand(std::vector<std::unique_ptr<Command>> commands)
        : commands_(std::move(commands)) {}

    void apply(GameField* game_field) override {
        for (const auto& command : commands_) {
            command->apply(game_field);
        }
    }

private:
    std::vector<std::unique_ptr<Command>> commands_;
};
```

Iterator



Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из агрегированных объектов.

UML-диаграмма



Назначение

- Предоставляет способ последовательного доступа ко всем элементам составного объекта
- Абстракция, позволяющая разделить классы коллекций и алгоритмов
- Придает обходу коллекции "объектно-ориентированный статус"
- Полиморфный обход

Канонический пример итератора

```
template<class T>
class Iterator {
public:
    virtual ~Iterator() = default;
    virtual bool Done() const = 0; // Элементов в составном объекте больше нет
    virtual T* Current() const = 0; // Метод для получения текущего элемента
    virtual T* Next() = 0; // Метод для получения следующего элемента
};

template<class T>
class Collection {
public:
    virtual ~Collection() = default;
    virtual std::unique_ptr<Iterator<T>> MakeIterator() = 0;
};
```

Обратите внимание на методы итератора — Done и Current в отличие от Next отмечены const. Это значит, что они не меняют состояние итератора.

Пример

Реализуем коллекцию, которая будет представлять собой последовательность целых чисел.

Добавим в эту последовательность возможность прохода как по возрастанию, так и по убыванию с произвольной длиной шага.

Последовательность сделаем полуоткрытой, то есть включая начальное значение, но исключая конечное, например: [0, 5)

Пример использования IntRange

```
int main() {
    // Последовательность от 0 до 5 с шагом в 1
    auto range = std::make_unique<IntRange>(0, 5);
    auto it = range->MakeIterator();
    const int* i = it->Current();  // Получаем текущее значение — 0
    while (!it->Done()) {  // Продолжаем, пока не выйдем за пределы последовательности
        std::cout << *i << std::endl;
        i = it->Next();  // Переключаем итератор так, чтобы он сделал шаг вперёд
    }
}
```

Результат: 0 1 2 3 4

Пример использования IntRange

```
int main() {
    // Последовательность от 5 до -5 с шагом в 2
    auto range = std::make_unique<IntRange>(5, -5, 2);
    auto it = range->MakeIterator();
    for (const int* i = it->Current(); !it->Done(); i = it->Next()) {
        std::cout << *i << std::endl;
    }
}
```

Результат: ???

Пример использования IntRange

```
int main() {
    // Последовательность от 5 до -5 с шагом в 2
    auto range = std::make_unique<IntRange>(5, -5, 2);
    auto it = range->MakeIterator();
    for (const int* i = it->Current(); !it->Done(); i = it->Next()) {
        std::cout << *i << std::endl;
    }
}
```

Результат: 5 3 1 -1 -3

Реализация итератора

```
// const int, потому что нет возможности менять промежуток
class RangeIterator : public Iterator<const int> {
public:
    RangeIterator(int current, int end, int step)
    : current_(current), end_(end), step_(step) {}

    bool Done() const override {
        return step_ < 0 ? current_ <= end_ : current_ >= end_;
    }

    const int* Current() const override {
        return Done() ? nullptr : &current_;
    }

    const int* Next() override {
        current_ += step_;
        return Current();
    }

private:
    int current_ = 0;
    int end_ = 0;
    int step_ = 0; // Переменная для шага может иметь отрицательное значение
};
```

Реализация коллекции

```
class IntRange : public Collection<const int> {
public:
    IntRange(int min, int max, unsigned step = 1)
    : min_(min), max_(max), step_(min < max ? step : -step) {}

    std::unique_ptr<Iterator<const int>> MakeIterator() override {
        return std::make_unique<RangeIterator>(min_, max_, step_);
    }

private:
    int min_ = 0;
    int max_ = 0;
    int step_ = 0;
};
```

Итераторы в стандартной библиотеке

Коллекции стандартной библиотеки имеют два метода:

- **begin()** возвращает итератор, указывающий на начала контейнера
- **end()** возвращает итератор, указывающий на конец контейнера. А точнее на одну ячейку больше последней. При попытке использовать значение случится ошибка

Итераторы в стандартной библиотеке

Итератор, возвращаемый из коллекции, имеет методы:

- **operator++** для перехода к следующему элементу коллекции
- **operator==** для сравнения с другими итераторами на равенство
- **operator!=** для сравнения с другими итераторами на **не**равенство
- **operator*** для получения значения, на которое указывает итератор

Какой стандартный тип языка C++ может выступать в качестве итератора?

Тип указателя!

```
int main() {
    const int array[] = {1, 2, 3, 4, 5};
    const int* begin = array; // Итератор на начало массива
    const int* end = array + sizeof(array) / sizeof(array[0]);  // Итератор на конец
    for (const int* it = begin; it != end; ++it) {
        std::cout << *it << std::endl; // Итератор на текущий элемент массива
    }
}
```

Результат: 0 1 2 3 4

Range-based for loop

В современном C++ циклы, которые работают с итераторами можно сократить.

```
int main() {
    const int array[] = {0, 1, 2, 3, 4};
    // Под капотом эта конструкция превращается в обычный цикл for.
    for (int value : array) {
        std::cout << value << std::endl;
    }
}
```

Перепишем IntRange в стиле C++ итераторов

```
int main() {
    auto range = IntRange(5, -5, 2);
    for (auto i : range) {
        std::cout << i << std::endl;
    }
}
```

Результат: 5 3 1 -1 -3

Реализация итератора

```
class Iterator {
public:
    Iterator(int current, int step)
    : current_(current), step_(step) {}

    Iterator& operator++() {
        current_ += step_;
        return *this;
    }
    bool operator==(Iterator other) const {
        return current_ == other.current_;
    }
    bool operator!=(Iterator other) const {
        return !(*this == other);
    }
    int operator*() const { return current_; }

private:
    int current_ = 0;
    int step_ = 0;
};
```

Реализация коллекции

```
class IntRange {
public:
    IntRange(int min, int max, unsigned step = 1)
    : min_(min), max_(max), step_(min < max ? step : -step) {}

    Iterator begin() {
        return Iterator(min_, step_);
    }

    Iterator end() {
        return Iterator(max_, step_);
    }

private:
    int min_ = 0;
    int max_ = 0;
    int step_ = 0;
};
```

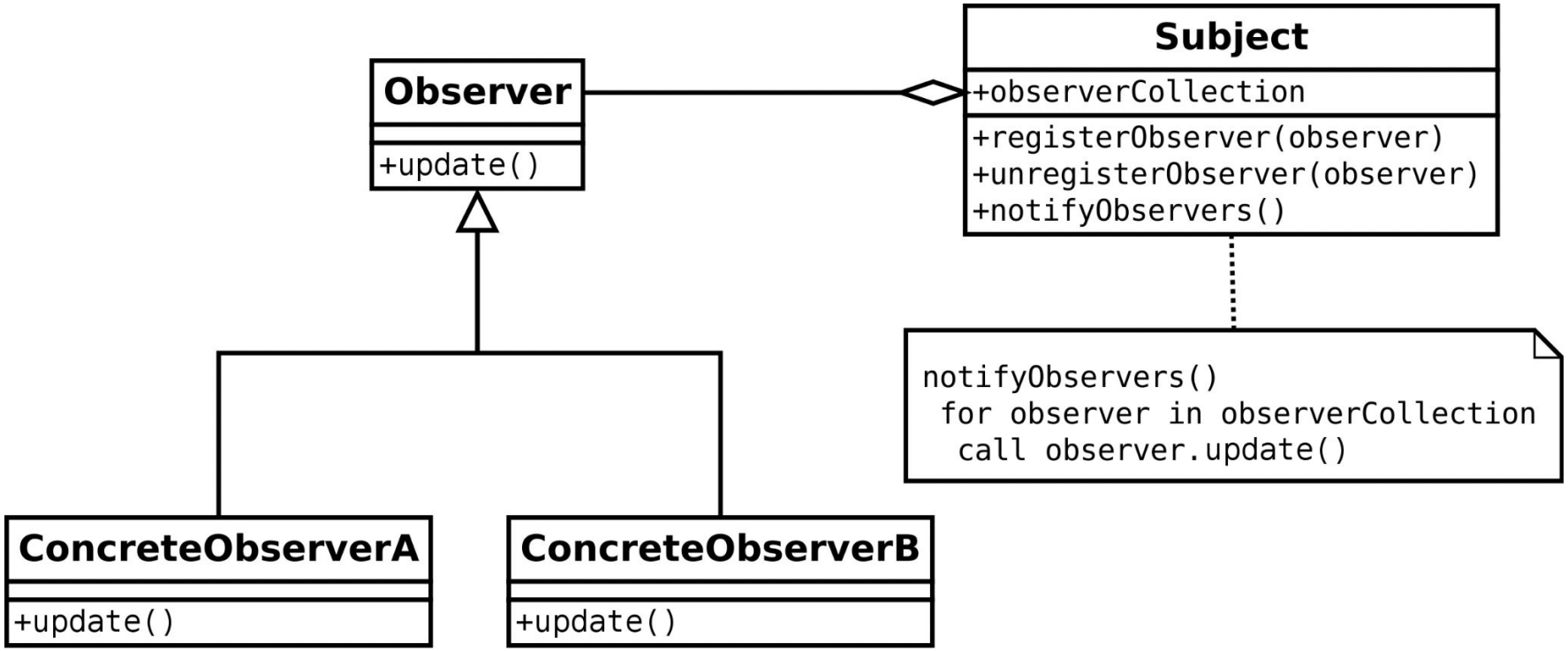
Observer

4

Реализует у класса механизм, который позволяет объекту этого класса получать оповещения об изменении состояния других объектов и тем самым наблюдать за ними.

Подписывающиеся классы называются наблюдателями.

UML-диаграмма



Пример наблюдателя

Рассмотрим простой пример наблюдателя:

```
class Observer {
public:
    virtual ~Observer() = default;
    virtual void DidStartLoading() {}
    virtual void DidStopLoading() {}
};
```

Основная часть объекта наблюдателя — методы, реагирующие на изменения в наблюдаемом объекте.

Пример наблюдаемого объекта

```
class Observed {
public:
    void AddObserver(Observer* observer) {
        observers_.push_back(observer);
    }

    void StartLoading() {
        for (auto observer : observers_) {
            observer->DidStartLoading();
        }
    }

    void StopLoading() {
        for (auto observer : observers_) {
            observer->DidStopLoading();
        }
    }

private:
   std::vector<Observer*> observers_;
};
```

Какая потенциальная проблема скрыта в этом коде?

Сырой указатель!

```
void AddObserver(Observer* observer) {
    observers_.push_back(observer);
}
```

В случае если объект, на который смотрит указатель **observer** будет разрушен, то при попытке оповестить его, случится ошибка.

Решение 1

Добавляем метод **RemoveObserver** в наблюдаемый объект. Этот метод обязан вызвать объект-наблюдатель перед разрушением. Такое решение отлично ложится в логику RAII.

```
void AddObserver(Observer* observer) {
    observers_->push_back(observer);
}

void RemoveObserver(Observer* observer) {
    auto it = std::remove(observers_.begin(), observers_.end(), observer);
    observers_.erase(it, observers_.end());
}
```

Решение 2

Меняем сырой указатель на слабый умный указатель, например **std::weak_ptr**

```
class Observed {
public:
    void StartLoading() {
        for (auto observer : observers_) {
            if (auto strong_ptr = observer.lock()) {
                strong_ptr->DidStartLoading();
            }
        }
    }

    void StopLoading() {
        for (auto observer : observers_) {
            if (auto strong_ptr = observer.lock()) {
                strong_ptr->DidStopLoading();
            }
        }
    }

    void AddObserver(std::weak_ptr<Observer> observer) {
        observers_.push_back(observer);
    }

private:
    std::vector<std::weak_ptr<Observer>> observers_;
};
```

Chain of Responsibility

5

Шаблон проектирования, предназначенный для организации в системе уровней ответственности.

Описание паттерна цепочка ответственности

Назначение

- Имеется группа объектов, которые могут обрабатывать сообщения определенного типа.
- Все сообщения должны быть обработаны хотя бы одним объектом.
- Сообщения в системе обрабатываются по схеме «обработай сам либо перешли другому».
- Одни сообщения обрабатываются на том уровне, где они получены, а другие пересылаются объектам иного уровня.

Примеры

- Обработка push-уведомлений компонентами мобильного приложения.
- Обработка UI-элементами действий пользователя, например, клика.

Пример с обработкой UI-элементами

```
class View {  // Базовый класс окошка, которое видит пользователь
public:
    View(int x, int y, int width, int height)  // Кординаты позиции на экране и размер окошка
    : x_(x), y_(y), width_(width), height_(height) {}
    virtual ~View() = default;
    bool containsClick(int x, int y) const {  // Определяем пришёлся ли клик в окошко
        return x >= x_ && x <= x_ + width_ && y >= y_ && y <= y_ + height_;
    }
    void addSubview(std::unique_ptr<View> view) {  // Встраиваем другое окошко поверх текущего
        view->parent_ = this;
        subviews_.push_back(std::move(view));
    }
    void handleClick(int x, int y) {
        if (containsClick(x, y)) {
            std::cout << "Clicked view " << this << " at " << x << ":" << y << std::endl;
            onClick(); // Обрабатываем клик если он пришёлся в это окошко
            return;
        }
        if (parent_) {
            parent_->handleClick(x, y);  // Иначе пробрасываем клик в родительское окошко
        }
    }
    virtual void onClick() {}  // Обработка клика переопределяется в классах-наследниках

private:
    std::vector<std::unique_ptr<View>> subviews_;
    View* parent_ = nullptr;
    int x_ = 0;
    int y_ = 0;
    int width_ = 0;
    int height_ = 0;
};
```

Пример с обработкой UI-элементами

В качестве примера класса, наследующего окошко, выступает кнопка. У неё есть позиция на экране, размер и она умеет выполнять действие, если на неё кликнуть.

```
class Button : public View {
public:
    Button(int x, int y, int width, int height) : View(x, y, width, height) {}

    void onClick() override {
        std::cout << "Clicked button " << this << std::endl;
    }
};
```

Пример с обработкой UI-элементами

```
int main() {
    auto root = std::make_unique<View>(0, 0, 256, 256);  // Создаём основное окошко
    auto button = std::make_unique<Button>(50, 50, 25, 25);  // Создаём кнопку
    auto* foregroundView = button.get();  // Окошко перед пользователем
    root->addSubview(std::move(button));  // Встраиваем кнопку в основное окошко
    foregroundView->handleClick(56, 70);  // Кликаем в кнопку
    foregroundView->handleClick(160, 130);  // Кликаем мимо кнопки
}
```

Clicked view **0x7fd4fbc05b80** at 56:70

Clicked button **0x7fd4fbc05b80**// Кнопка обрабатывает клик по координатам 56:70

Clicked view 0x7fd4fbc05b40 at 160:130 // Клик обрабатывает основное окошко

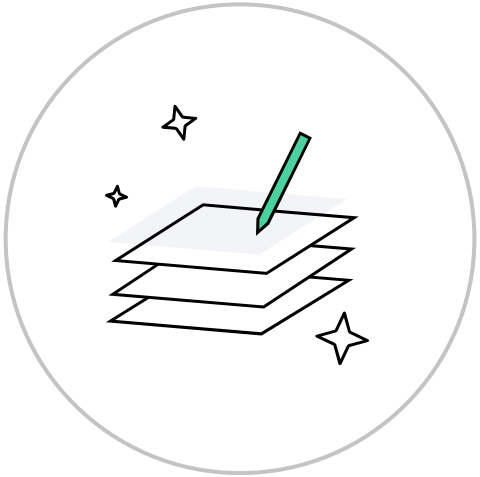
Итоги занятия

1. Узнали, какие бывают поведенческие паттерны
2. Познакомились с основными паттернами
3. Реализовали и разобрали основные подходы

Домашнее задание

Давайте посмотрим ваше домашнее задание.

1. Вопросы по домашней работе задавайте в чате группы
2. Задачи можно сдавать по частям
3. Зачёт по домашней работе ставят после того, как приняты все задачи



Дополнительные материалы

- Паттерны поведения в C++

Пример передачи команды по сети

```
class Command {
public:
    enum class Type : int32_t {
        Pause,
        AddUnit,
        Complex,
    };

    virtual ~Command() = default;

//  В зависимости от типа команды создаём нужный класс и передаём входящий поток для чтения параметров
    static std::unique_ptr<Command> parse(IStream& stream) {
        int32_t type = -1;
        stream >> type;
        switch (static_cast<Type>(type)) {
            case Type::Pause: return std::make_unique<PauseCommand>(stream);
            case Type::AddUnit: return std::make_unique<AddUnitCommand>(stream);
            case Type::Complex: return std::make_unique<ComplexCommand>(stream);
        }
    }
    virtual void write(OStream& stream) const = 0;  // Каждый объект команды записывает свой тип и параметры в поток
    virtual void apply(GameField* game_field) = 0;
};
```

Пример цепочки с обработкой уведомления

```
class PushMessage {  // Сообщение, которое получает мобильное приложение
public:
    enum class Type {
        kNews,
        kWeather,
        kChat,
        kUnknown,
    };

    virtual ~PushMessage() = default;
    virtual Type type() const = 0;  // Типы сообщений
    virtual const char* data() const = 0;  // Полезная информация в сообщении
};
class PushHandler {
public:
    explicit PushHandler(std::unique_ptr<PushHandler> next) : next_(std::move(next)) {}
    virtual ~PushHandler() = default;
    void receivePush(const PushMessage& msg) {
        if (handlePush(msg)) {  // Если текущий обработчик принял сообщение, то считаем его обработанным
            return;
        }
        if (!next_) {  // Если обработчиков больше нет, то оповещаем о том, что сообщение осталось необработанным
            throw std::runtime_error("Error: Message should be handled!");
        }
        next_->receivePush(msg);  // Иначе передаём сообщение следующему обработчику
    }

private:
    virtual bool handlePush(const PushMessage& msg) = 0;  // В наследниках реализуем логику по обработке сообщений

    std::unique_ptr<PushHandler> next_;
};
```

Реализация обработчиков

```
class NewsHandler : public PushHandler {
public:
    using PushHandler::PushHandler;
private:
    bool handlePush(const PushMessage& msg) override {
        if (msg.type() != PushMessage::Type::kNews) {
            return false;
        }
        std::cout << "Breaking news! " << msg.data() << std::endl;
        return true;
    }
};
class WeatherHandler : public PushHandler {
public:
    using PushHandler::PushHandler;
private:
    bool handlePush(const PushMessage& msg) override {
        if (msg.type() != PushMessage::Type::kWeather) {
            return false;
        }
        std::cout << "Weather forecast: " << msg.data() << std::endl;
        return true;
    }
};
class ChatHandler : public PushHandler {
public:
    using PushHandler::PushHandler;
private:
    bool handlePush(const PushMessage& msg) override {
        if (msg.type() != PushMessage::Type::kChat) {
            return false;
        }
        std::cout << "New message received: " << msg.data() << std::endl;
        return true;
    }
};
```

Реализация сообщений

```
class NewsMessage : public PushMessage {
public:
    Type type() const override { return Type::kNews; }
    const char* data() const override {
        return "New C++ standard is published!";
    }
};

class WeatherMessage : public PushMessage {
public:
    Type type() const override { return Type::kWeather; }
    const char* data() const override {
        return "It is expected -25C degrees tomorrow.";
    }
};

class ChatMessage : public PushMessage {
public:
    Type type() const override { return Type::kChat; }
    const char* data() const override {
        return "Hello, how are you?";
    }
};

class UnkownMessage : public PushMessage {
public:
    Type type() const override { return Type::kUnknown; }
    const char* data() const override {
        return nullptr;
    }
};
```

Реализация цепочки обработчиков

```
int main() {
    auto news_handler = std::make_unique<NewsHandler>(nullptr);
    auto weather_handler = std::make_unique<NewsHandler>(std::move(news_handler));
    auto chat_handler = std::make_unique<ChatHandler>(std::move(weather_handler));

    try {
        chat_handler->receivePush(ChatMessage());
        chat_handler->receivePush(NewsMessage());
        chat_handler->receivePush(UnkownMessage());
    } catch (const std::exception& e) {
        std::cout << e.what() << std::endl;
    }
}
```

New message received: Hello, how are you?

Breaking news! New C++ standard is published!

Error: Message should be handled!

Задавайте вопросы и пишите отзыв о лекции

Иван Поляков
Разработчик Go/C++ в инфраструктуре поиска в Авито