# Паттерн “стратегия”

Бокай Иван Андреевич
23.Б08

# Определение

Стратегия – это поведенческий паттерн проектирования, определяющий семейство схожих алгоритмов и помещающий каждый в отдельный класс.

# Проблема

Представим, что в приложении необходимо реализовать расчёт стоимости доставки. Доставка может быть разная: почтой, курьером, в пункты выдачи и т.д.

Реализация всех алгоритмов расчёта стоимости в одном классе неизбежно приведёт к росту этого класса и большой if-else (или switch-case) конструкции

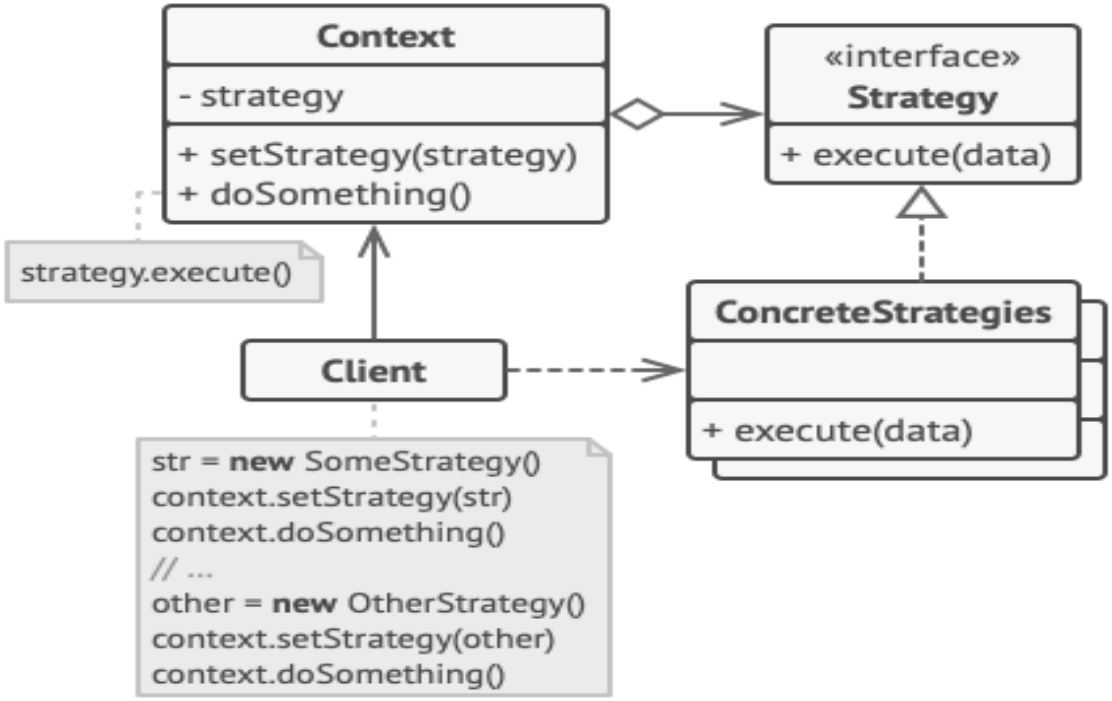
# Проблема (C++)

```
class DeliveryService {
private:
    int calculate_courier(int weight) { return 10 + 5 * weight; }

    int calculate_post(int weight) { return 5 + 2 * weight; }

    int calculate_express(int weight) { return 20 + 3 * weight; }

public:
    int calculate(int weight, string type) {
        // Может быть заменено на switch-case, но дело это не спасает
        if (type == "courier") {
            return calculate_courier(weight);
        } else if (type == "post") {
            return calculate_post(weight);
        } else if (type == "express") {
            return calculate_express(weight);
        }
        return -1;
    }
};
```

# Решение

- Создать абстрактный класс стратегии
- Вынести схожие алгоритмы в отдельные классы (стратегии)
- Изначальный класс будет играть роль контекста, делегируя исполнение стратегиям
- Извне выбирается конкретная стратегия и запрашивается результат.
- Изначальный класс вызывает выполнение стратегии, получает результат и отдаёт его пользователю

# Структура


Context
- strategy
+ setStrategy(strategy)
+ doSomething()
«interface»
Strategy
+ execute(data)
strategy.execute()
Client
ConcreteStrategies
+ execute(data)
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()

# Где(когда) применять?

- Когда вам нужно использовать разные вариации какого-то алгоритма внутри одного объекта
- Когда есть множество похожих классов, отличающихся только некоторым поведением
- Когда вы не хотите обнажать детали реализации алгоритмов для других классов.
- Когда различные вариации алгоритмов реализованы в виде развесистого условного оператора.

# Этапы реализации

1. Определение проблемного алгоритма
2. Создание интерфейса, описывающего алгоритм
3. Вариации алгоритма – отдельные классы, реализующие этот интерфейс
4. В классе контекста – поле для хранения текущей стратегии и метод для изменения
5. Клиенты подают объект-стратегию, чтобы контекст вёл себя определённым образом

# Реализация (C++)

Интерфейс, описывающий алгоритм:

```
class ShippingStrategy {
public:
    virtual ~ShippingStrategy() = default;
    virtual int calculate(int weight) const = 0;
};
```

# Реализация (C++)

Отдельные классы алгоритмов:

```
class CourierStrategy : public ShippingStrategy {
public:
    int calculate(int weight) const override { return 10 + 5 * weight; }
};

class PostStrategy : public ShippingStrategy {
public:
    int calculate(int weight) const override { return 5 + 2 * weight; }
};
```

# Реализация (C++)

Класс контекста:

```
class DeliveryService {
private:
    std::unique_ptr<ShippingStrategy> strategy_;

public:
    DeliveryService(std::unique_ptr<ShippingStrategy>&& strategy)
        : strategy_(std::move(strategy)) {}

    void set_strategy(std::unique_ptr<ShippingStrategy>&& strategy) {
        strategy_ = std::move(strategy);
    }

    int calculate(int weight) { return strategy_->calculate(weight); }
};
```

# Использование (C++)

```cpp
int main() {
    int weight = 10;
    DeliveryService ds(std::make_unique<CourierStrategy>());
    std::cout << ds.calculate(weight);
    ds.set_strategy(std::make_unique<PostStrategy>());
    std::cout << ds.calculate(weight);
    return 0;
}
```

# Преимущества

- Замена алгоритмов на лету
- Изолирует алгоритмы от других классов
- Переход от наследования к делегированию
- Реализует принципы SOLID (open-closed principle)

# Недостатки

- Усложнение программы, за счёт создания множества классов

- Клиент должен знать, что делает каждая стратегия

# Источники

https://refactoring.guru/ru

Спасибо за внимание!