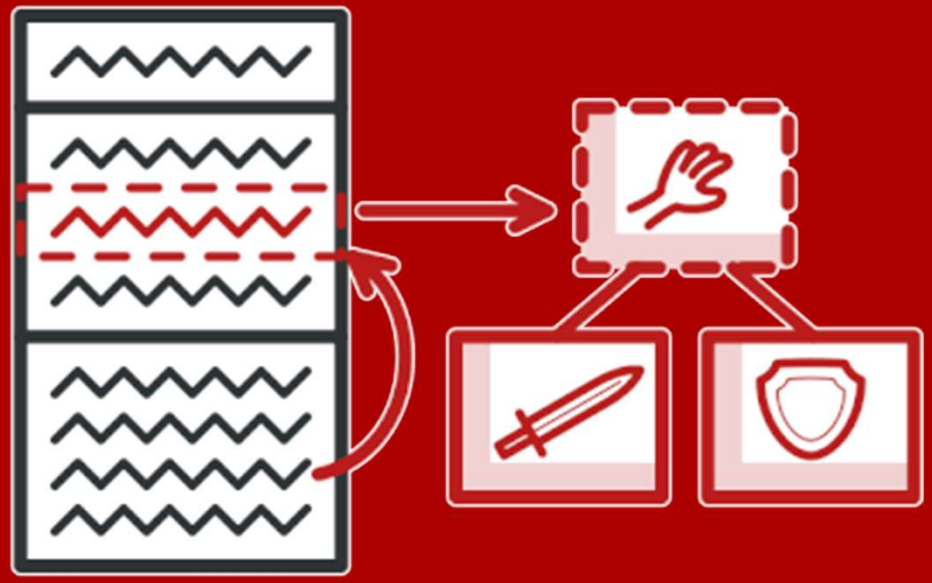
Патерн «Стратегія»

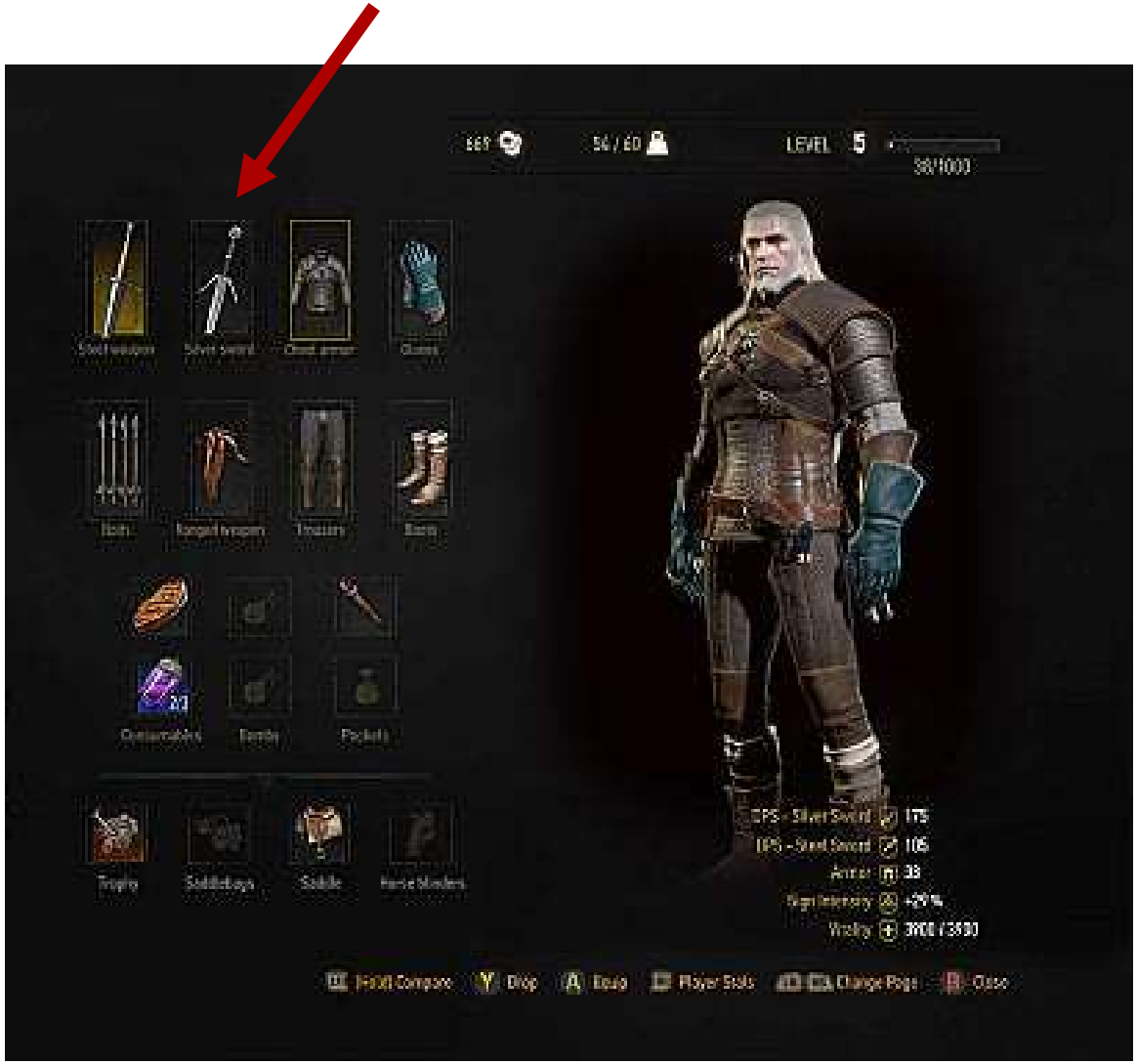
Ткаченко Андрій



Проблема

Уявіть, що ви робите гру, в якій гравець воює проти монстрів за допомогою різної зброї.

Кожна зброя наносить свою шкоду та має свої ефекти на монстрів.



Найбільш очевидне рішення

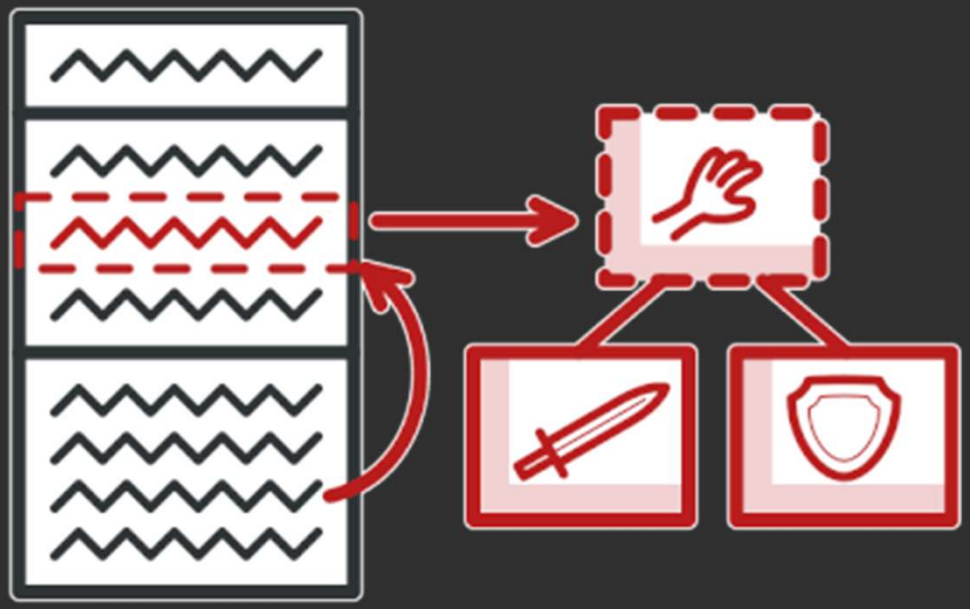
```java
public class SimplePlayer {

    private String weapon = "fist";

    public void attack(Monster monster) {
        switch (weapon) {
            case "fist":
                monster.damage(1);
                break;
            case "sword":
                monster.damage(10);
                break;
        }
    }

    public void setWeapon(String weapon) {
        this.weapon = weapon;
    }
}
```

Але...

Такий підхід погано масштабується та складний для подальших змін під час розробки

```
public void attack(Monster monster) {
    switch (weapon) {
        case "fist":
            monster.damage(1);
            break;
        case "sword":
            monster.damage(10);
            break;
        case "fireSword":
            monster.damage(10);
            monster.setOnFire();
            break;
        case "hammer":
            monster.damage(7);
            monster.stun();
            break;
        case "winterSpear":
            monster.damage(5);
            monster.freeze();
            break;
    }
}
```

Для цього нам потрібен патерн «Стратегія»!

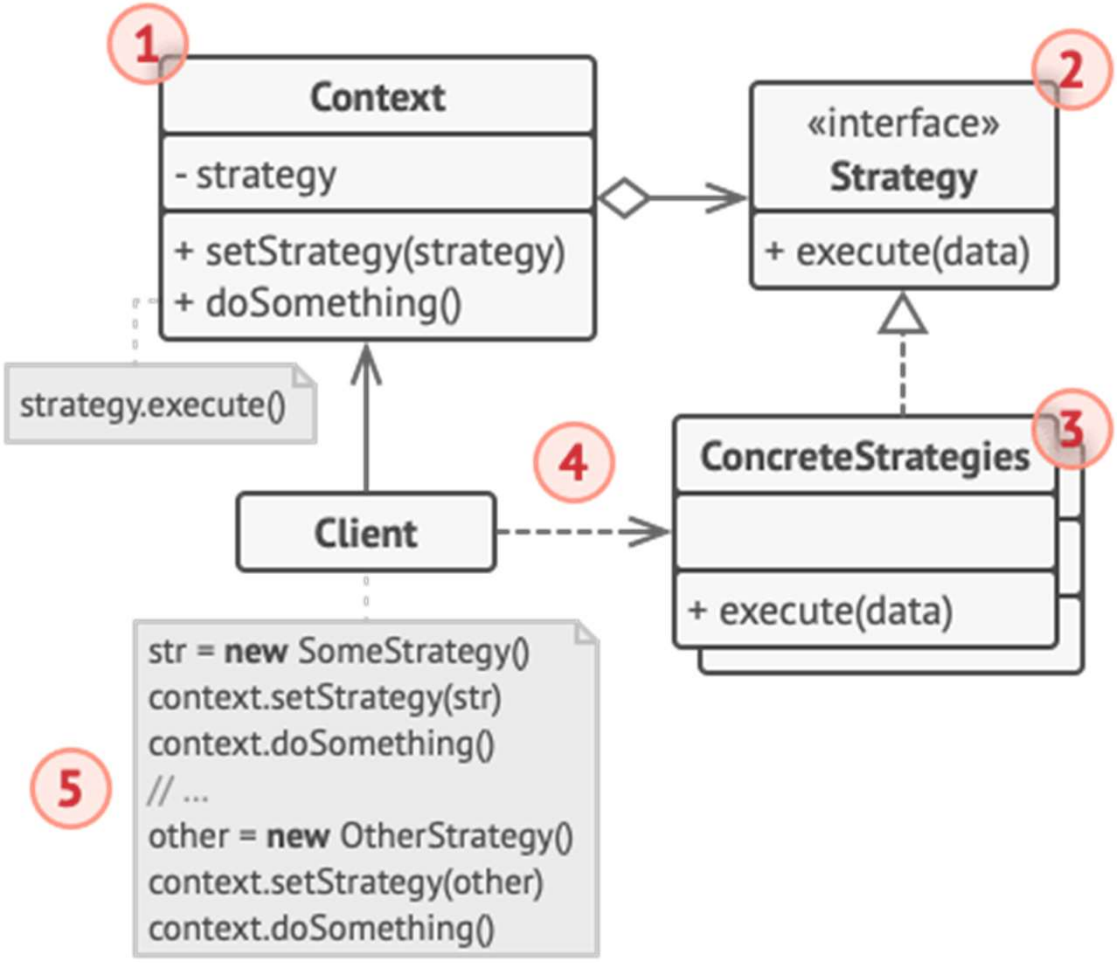


То що таке патерн «Стратегія»?

«Стратегія» – це поведінковий патерн проектування, який визначає сімейство схожих алгоритмів і розміщує кожен з них у власному класі.

Після цього алгоритми можна заміняти один на інший прямо під час виконання програми.

Якщо більш конкретно



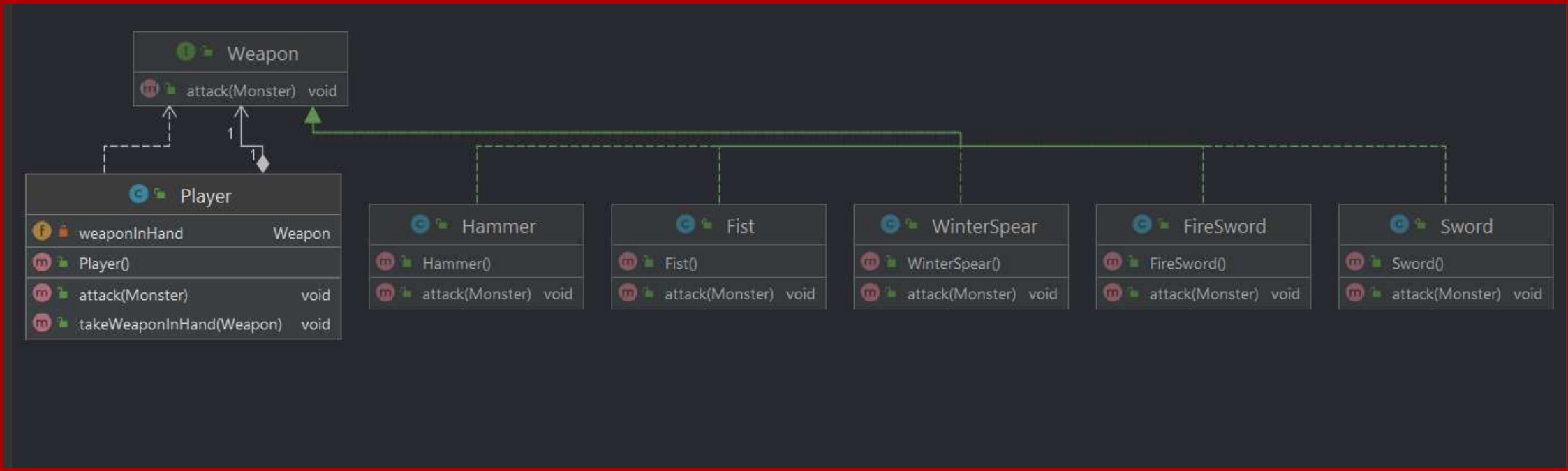
```java
public class Player {

    private Weapon weaponInHand = new Fist();

    public void attack(Monster monster){
        weaponInHand.attack(monster);
    }

    public void takeWeaponInHand(Weapon weapon){
        weaponInHand = weapon;
    }
}
```

```java
public class Sword implements Weapon {
    @Override
    public void attack(Monster monster) {
        monster.damage(10);
    }
}
```

В нашій грі це буде виглядати так...

- Гравець = Контекст
- Зброя = Стратегія

Приклад


Weapon
attack(Monster) void
1
1
Player
weaponInHand Weapon
Player()
attack(Monster) void
takeWeaponInHand(Weapon) void
Hammer
Hammer()
attack(Monster) void
Fist
Fist()
attack(Monster) void
WinterSpear
WinterSpear()
attack(Monster) void
FireSword
FireSword()
attack(Monster) void
Sword
Sword()
attack(Monster) void

Особлива можливість реалізації в Java через enum

```java
public enum Strategy {

    STRATEGY_A {
        @Override
        void execute() {
            System.out.println("Executing strategy A");
        }
    },

    STRATEGY_B {
        @Override
        void execute() {
            System.out.println("Executing strategy B");
        }
    };

    abstract void execute();
}
```

```java
public class Context {

    public static void main(String[] args) {
        Context useStrategy = new Context();
        useStrategy.perform(Strategy.STRATEGY_A);
        useStrategy.perform(Strategy.STRATEGY_B);
    }

    private void perform(Strategy strategy) { strategy.execute(); }
}
```

Переваги і недоліки

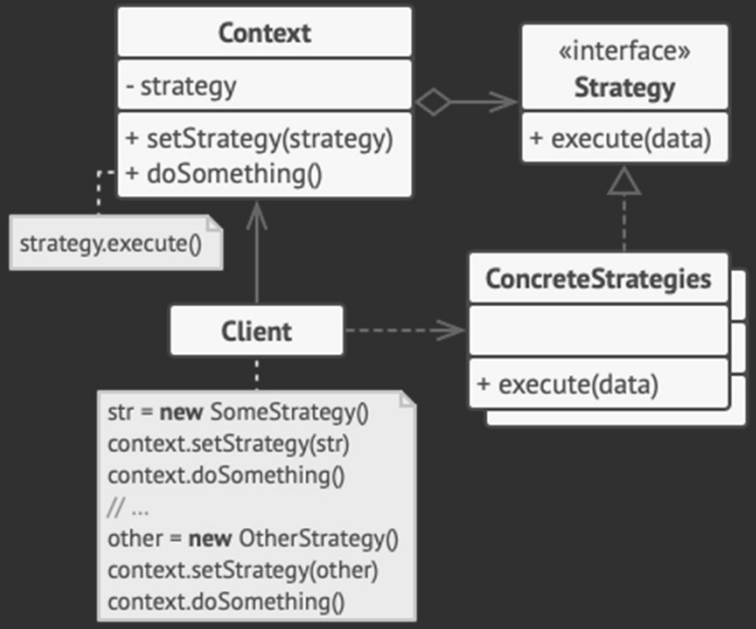
Переваги

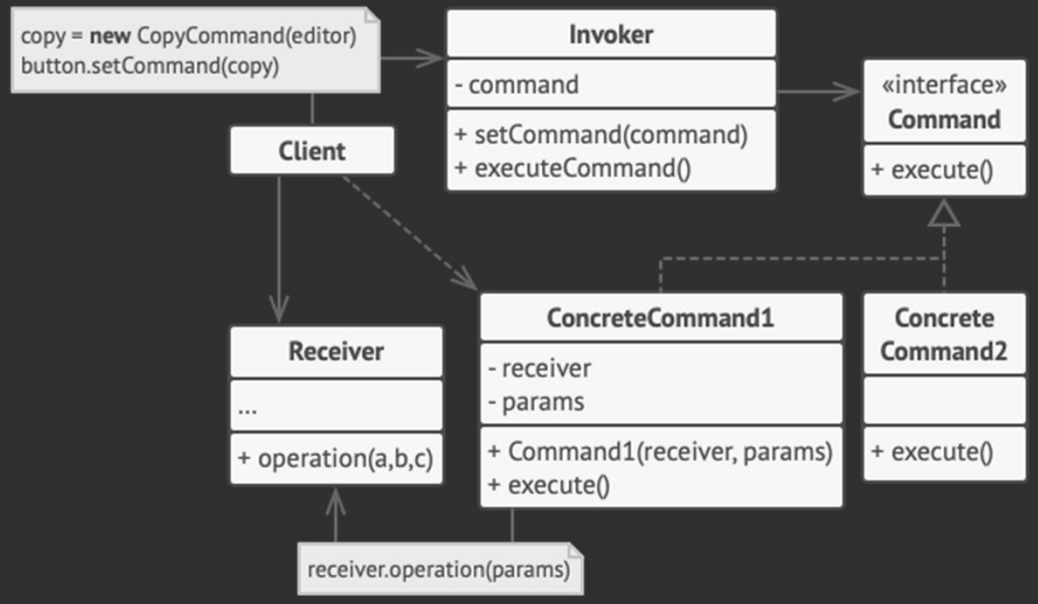
- ✓ Швидка (гаряча) заміна алгоритмів на ходу
- ✓ Ізолює код і алгоритми від інших класів
- ✓ Заміна спадкування делегуванням, що дає більше гнучкості
- ✓ Реалізація принципу відкритості/закритості (буква O в принципах SOLID)

Недоліки

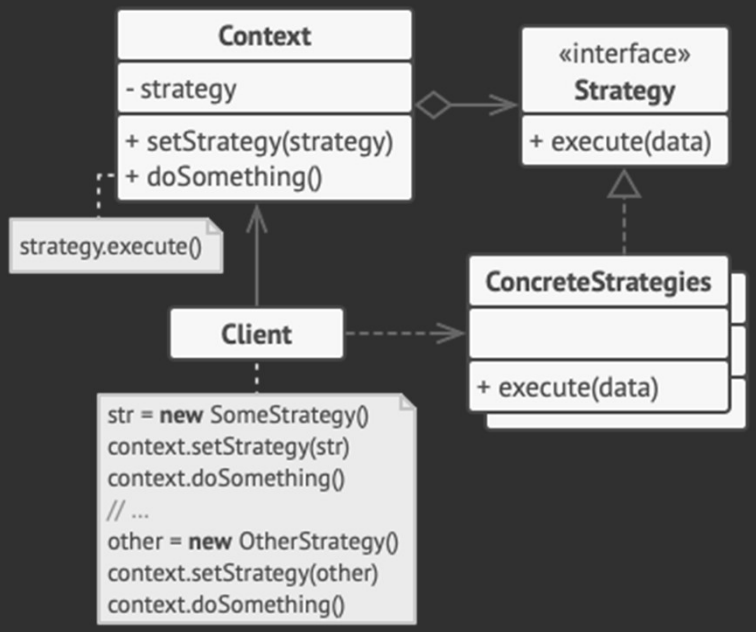
- ❖ Ускладнення програми через створення та використання додаткових класів.
- ❖ Користувач має чітко розуміти різницю між різними стратегіями, щоб завжди вибирати потрібну.

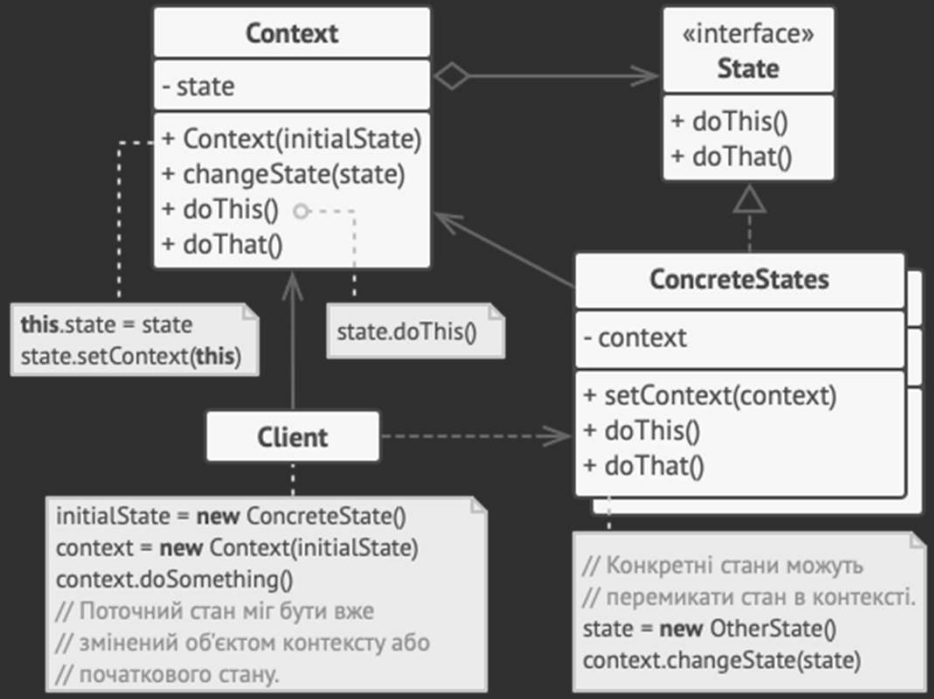
«Стратегія» та «Команда»



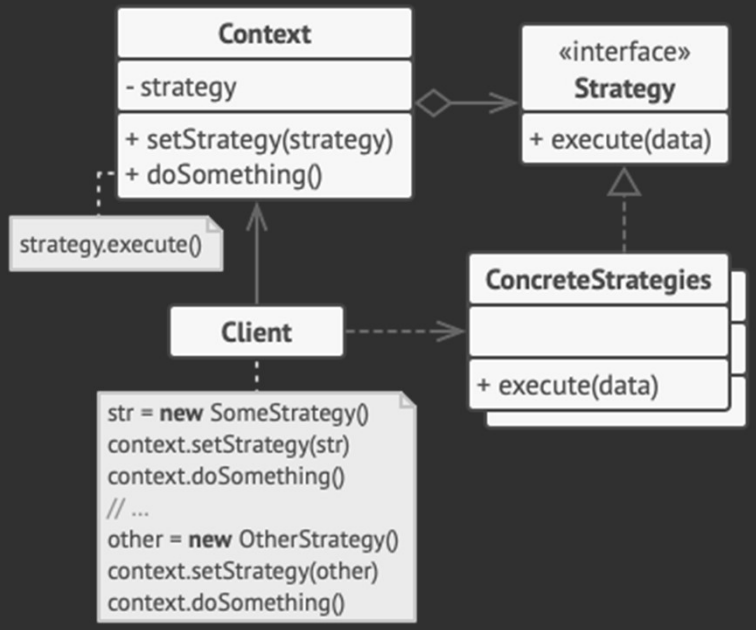


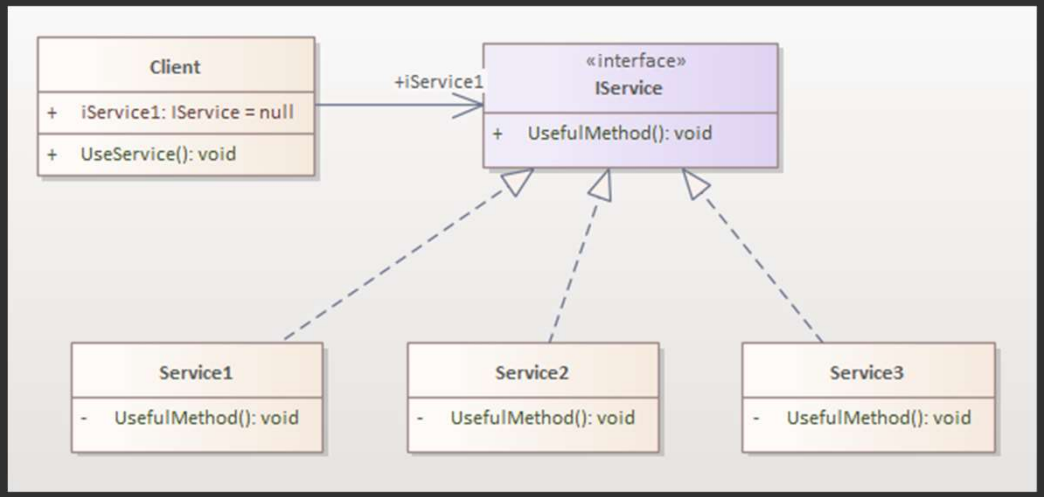
«Стратегія» та «Стан»





«Стратегія» та «Впровадження залежностей»





Порівняння із іншими

- **Стратегія змінює поведінку об’єкта «зсередини», а Декоратор змінює його «ззовні».**
- **Міст, Стратегія та Стан (а також трохи і Адаптер) мають схожі структури класів — усі вони побудовані за принципом «композиції», тобто делегування роботи іншим об’єктам.**