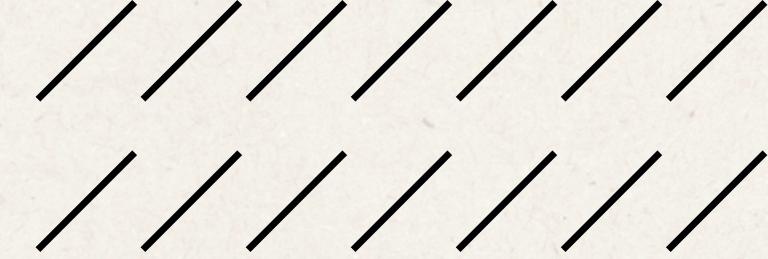


# **VISITOR**

Підготував Прокоф'єв Андрій

# Що таке **Visitor**?

Поведінковий патерн проектування, який дозволяє додавати до програми нові операції, не змінюючи класи об'єктів, над якими ці операції виконуються.

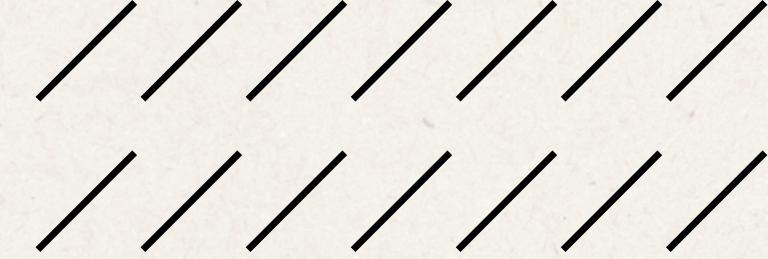


## Вирішує такі задачі :

Дозволяє легко  
розширювати  
функціонал

Відокремлює  
алгоритми від  
структур даних

Не перевантажує  
сторонніми  
методами



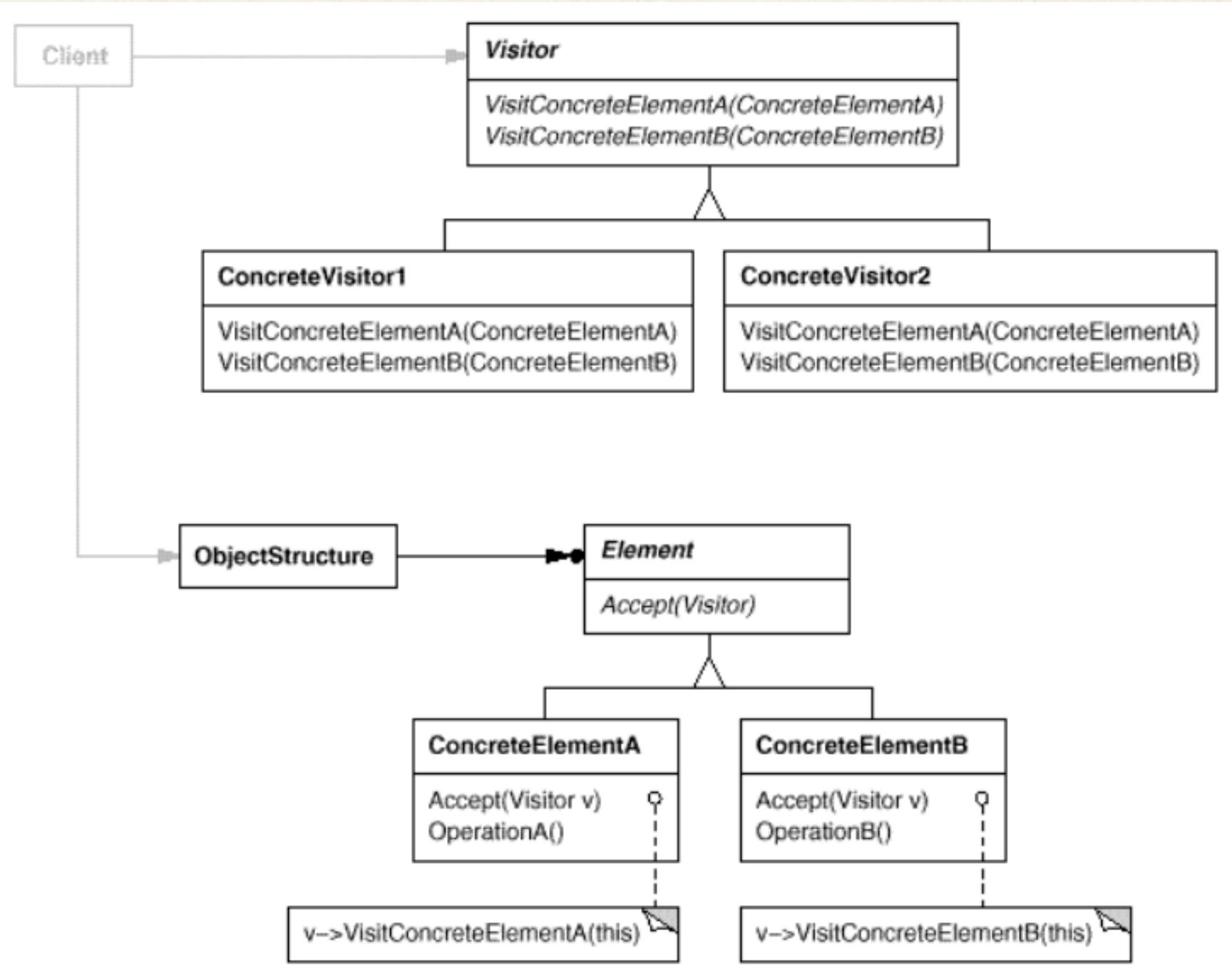
## Приклад з життя

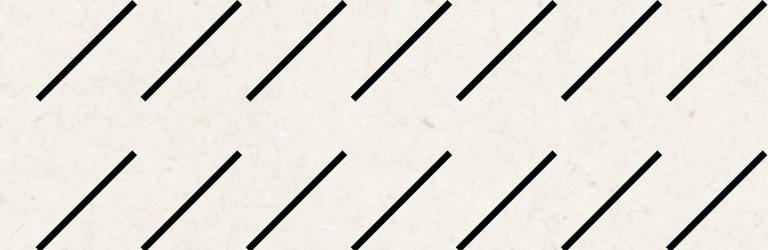
Податкова перевірка: Магазини лише приймають інспектора. Інспектор (Visitor) сам знає, що в аптекі перевіряти ліки, а в книжковому - звіти, не змінюючи роботу самих магазинів.

Ресторан: Страви це дані. Офіціант проходить повз них, щоб подати, а клієнт щоб з'сти. Логіка дій знаходиться в голові відвідувача, а не в складі страви.

# Рішення

1. Створити абстрактний клас `Visitor` із методами для відвідування кожного із конкретних елементів
2. Створити інтерфейс `Element` із методом `Accept(Visitor)`, який буде приймати відвідувача і виконувати необхідну операцію
3. Реалізувати цей інтерфейс в усіх класах, де потрібно додати новий функціонал. За потреби реалізувати додаткову логіку разом із викликом Відвідувача.
4. Для того щоб додати нову логіку до потрібного класу потрібно створити реалізацію класу `Visitor`. Для виклику методів використовуємо метод `Accept`





# Реалізація. Double Dispatch

В основі практичної реалізації моєго проєкту було використано механізм подвійної диспетчеризації. Через зв'язку методів `accept()` та `visit()`: під час виклику елемент передає посилання на самого себе (`this`) «у гості» до відвідувача.

Таке рішення було необхідним, оскільки в мові Java стандартне перевантаження методів базується на статичних типах під час компіляції. Натомість подвійна диспетчеризація дає системі можливість вже під час виконання програми точно визначити та застосувати специфічний алгоритм до кожного товару.

```
1 usage  ↗ Prokofiev Andrii
@Override
public void accept(Visitor visitor) {
    visitor.visit( electronics: this);
}
```

```
3 usages  ↗ Prokofiev Andrii
public void applyVisitor(Visitor visitor) {
    for (Item item : items) {
        item.accept(visitor);
    }
}
```

# Stateful TaxVisitor

Це тип відвідувача, який здатний зберігати інформацію під час обходу структури об'єктів, тобто накопичує результат

Як це реалізовано в TaxVisitor:

- Клас містить приватне поле totalTax, яке ініціалізується нулем.
- Кожного разу, коли викликається метод visit(Electronics) або visit(Food), відвідувач обчислює податок для конкретного товару і додає його до загальної суми.
- Після того, як кошик (Cart) завершить обхід усіх елементів, ми звертаємося до відвідувача за фінальним результатом через метод getTotalTax().

```
3 usages  ↗ Prokofiev Andrii
public class TaxVisitor implements Visitor {
    3 usages
    private double totalTax = 0;

    1 usage  ↗ Prokofiev Andrii
    @Override
    public void visit(Electronics electronics) {
        double tax = electronics.getBasePrice() * 0.20;
        totalTax += tax;
        System.out.println(STR."Податок: \{electronics.getModel()\}: \{tax\}");
    }

    1 usage  ↗ Prokofiev Andrii
    @Override
    public void visit(Food food) {
        double tax = (food.getWeight() * food.getBasePrice()) * 0.05;
        totalTax += tax;
        System.out.println(STR."Податок (\{food.getWeight()\}кг): \{tax\}");
    }

    1 usage  ↗ Prokofiev Andrii
    public double getTotalTax() { return totalTax; }
}
```

# Stateless JsonExportVisitor

Це тип відвідувача, який не зберігає інформацію про попередні кроки під час обходу структури об'єктів, тобто кожна операція є незалежною.

Як це реалізовано в Export-відвідувачах:

- Клас не має внутрішніх змінних-акумуляторів. Його задача – виконати миттєву дію над поточним об'єктом.
- Кожного разу, коли викликається метод visit(Electronics) або visit(Food), відвідувач просто конвертує дані елемента або проводить миттєвий аудит безпеки.
- Результат роботи відвідувача ми бачимо одразу (наприклад, вивід у консоль або файл). Оскільки стан не накопичується, один і той самий об'єкт відвідувача можна безпечно використовувати повторно для різних кошиків.

```
3 usages  ↗ Prokofiev Andrii
public class TaxVisitor implements Visitor {
    3 usages
    private double totalTax = 0;

    1 usage  ↗ Prokofiev Andrii
    @Override
    public void visit(Electronics electronics) {
        double tax = electronics.getBasePrice() * 0.20;
        totalTax += tax;
        System.out.println(STR."Податок: \{electronics.getModel()\}: \{tax\}");
    }

    1 usage  ↗ Prokofiev Andrii
    @Override
    public void visit(Food food) {
        double tax = (food.getWeight() * food.getBasePrice()) * 0.05;
        totalTax += tax;
        System.out.println(STR."Податок (\{food.getWeight()\}кг): \{tax\}");
    }

    1 usage  ↗ Prokofiev Andrii
    public double getTotalTax() { return totalTax; }
}
```

# Об'єктна структура та взаємодія

Об'єктна структура є контейнером для елементів і забезпечує інтерфейс для відвідувача, щоб він міг обійти всі компоненти системи. У нашому проекті цю роль виконує клас **Cart**.

- Кошик не знає, які саме операції виконуються над товарами, лише зберігає список та надавати доступ.
- Метод **applyVisitor** дозволяє одним викликом застосувати будь-який новий алгоритм до всіх товарів у списку.
- Ми можемо додавати нові типи відвідувачів, не змінюючи код класу Cart.

```
3 usages  ↗ Prokofiev Andrii
public class Cart {
    2 usages
    private final List<Item> items = new ArrayList<>();

    4 usages  ↗ Prokofiev Andrii
    public void addItem(Item item) {
        items.add(item);
    }

    3 usages  ↗ Prokofiev Andrii
    public void applyVisitor(Visitor visitor) {
        for (Item item : items) {
            item.accept(visitor);
        }
    }
}
```

# Переваги та недоліки

- |  |   |
|--|---|
| <ul style="list-style-type: none"><li><b>01</b> Забезпечення принципу єдиної відповідальності</li><li><b>02</b> Забезпечення принципу відкритості і закритості</li><li><b>03</b> Відвідувач може зберігати корисну інформацію при роботі з об'єктами</li></ul> | <ul style="list-style-type: none"><li><b>01</b> Треба оновлювати усіх відвідувачів при додаванні класу, який можна відвідати</li><li><b>02</b> Відвідувач може не мати потрібного доступу до приватних полів</li><li><b>03</b> Може порушувати інкапсуляцію</li></ul> |
|--|---|

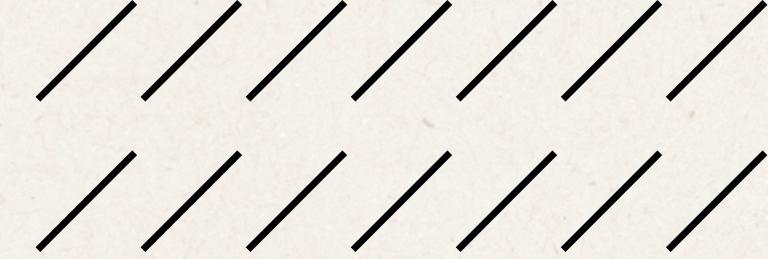
# Коли **Visitor** доречний?

## **ТРЕБА, ЯКЩО:**

- Робота зі складними структурами
- Очищення класів від стороннього коду
- Вибіркові операції в ієрархії
- Розширення без модифікації

## **НЕ ТРЕБА, ЯКЩО:**

- Часті зміни в ієрархії класів об'єктів
- Проста структура або логіка
- Порушення інкапсуляції
- Коли об'єкти та алгоритми нерозривно пов'язані



# Джерела

- <https://refactoring.guru/design-patterns/visitor>
- <https://en.cppreference.com/w/cpp/utility/variant/visit>
- [https://en.wikipedia.org/wiki/Double\\_dispatch](https://en.wikipedia.org/wiki/Double_dispatch)