

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
РАДІОЕЛЕКТРОНІКИ

ЗВІТ
до практичного завдання №1
з дисципліни «Архітектура програмного забезпечення»

Тема: Патерн проектування Strategy

Виконав: студент групи ПЗПІ-22-2
Д'яченко Микита

Харків – 2025

1 ІСТОРІЯ ЗМІН

№	Дата	Версія звіту	Опис змін та виправлень
1	13.06.2025	1.0	Зроблена презентація на тему: “Патерн проектування Strategy”

2 ЗАВДАННЯ

Підготувати доповідь на тему «Патерн проєктування Strategy».

Створити презентацію, приклад програмного коду, відеодоповідь та оформити звіт згідно з ДСТУ 3008:2015.

3 ОПИС ВИКОНАНОЇ РОБОТИ

У ході виконання практичного завдання було обрано патерн проєктування Strategy, який належить до поведінкових патернів. Було підготовлено презентацію, в якій детально розкрито суть патерну, його призначення, переваги, недоліки та сфери застосування. Також реалізовано приклад програмного коду на мові Python з демонстрацією використання різних стратегій. Проведено відеозапис доповіді з поясненням кожного етапу презентації. Звіт оформлено згідно з вимогами стандарту ДСТУ 3008:2015.

4 ВИСНОВКИ

У результаті виконання завдання було здобуто практичні навички застосування патернів проєктування на прикладі Strategy. Патерн дозволяє гнучко змінювати алгоритми без зміни структури програми, що відповідає принципам чистої архітектури та принципу відкритості/закритості. Отримані знання можуть бути використані для розробки масштабованого та підтримуваного програмного забезпечення.

ДОДАТОК Б

Відеозапис доповіді на YouTube

<https://www.youtube.com/watch?v=zr0pgpiaE3F6g>

00:00 Вступ. Актуальність патернів проєктування

01:30 Класифікація патернів. Визначення патерну Strategy

03:00 Приклад використання у реальному житті

04:30 Приклад коду на Python та пояснення його роботи

06:00 Переваги, недоліки та висновки

ДОДАТОК Б

Слайди презентації

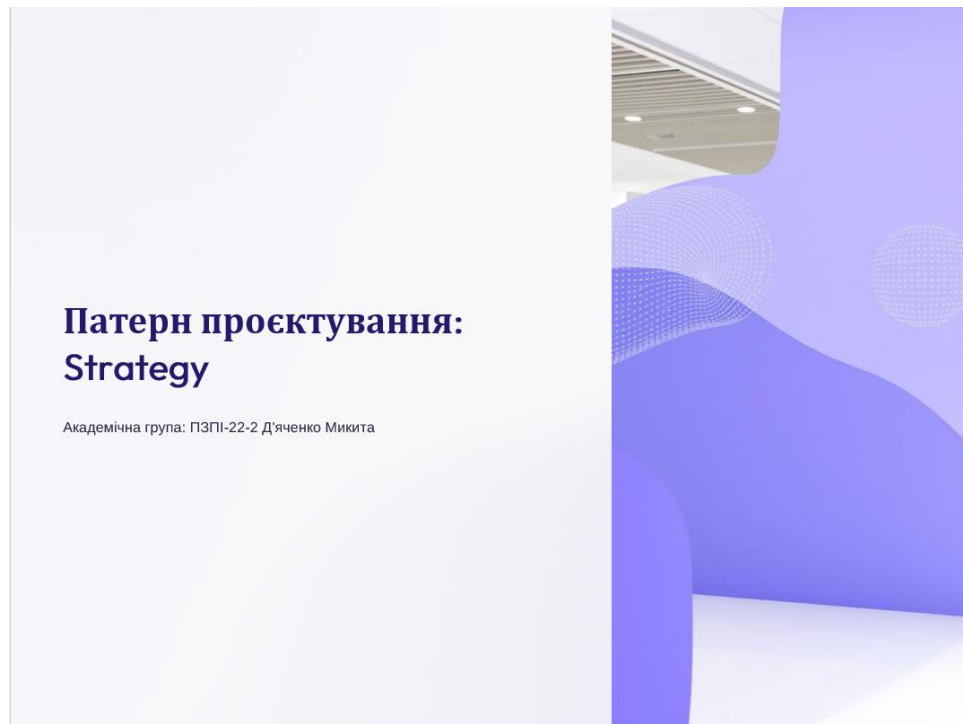


Рисунок Б.1 – Титульний слайд

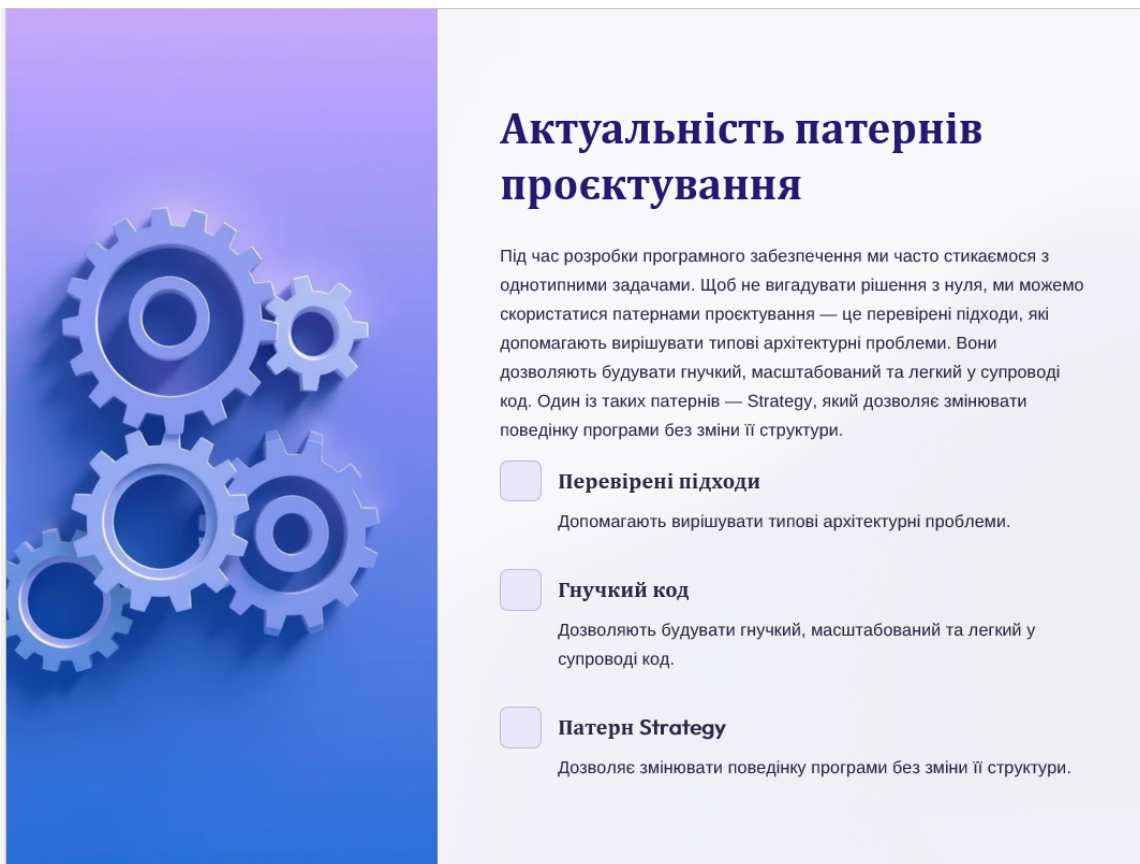


Рисунок Б.2 – Актуальність патернів проектування

Класифікація патернів GoF

У книзі «Design Patterns» від «банди чотирьох» (GoF) патерни поділяються на три великі групи:

Породжуючі (Creational)

Відповідають за створення об'єктів, наприклад Factory Method, Abstract Factory.

Структурні (Structural)

Описують, як компоненти програми зв'язуються між собою, наприклад Adapter, Composite.

Поведінкові (Behavioral)

Визначають способи взаємодії між об'єктами та передачу даних, наприклад Observer, State, і Strategy.

Strategy відноситься до поведінкових патернів і застосовується тоді, коли потрібно динамічно змінювати алгоритм під час виконання.

Рисунок Б.3 – Класифікація патернів GoF

Визначення патерну Strategy

Патерн Strategy дозволяє визначити сімейство алгоритмів, інкапсулювати кожен з них у окремий клас і зробити їх взаємозамінними. Іншими словами, це можливість «впровадити» в об'єкт змінювану поведінку, не змінюючи сам об'єкт. Це робиться через загальний інтерфейс.

1

Визначення сімейства алгоритмів

Інкапсулювати кожен з них у окремий клас і зробити їх взаємозамінними.

2

Впровадження змінюваної поведінки

Не змінюючи сам об'єкт, через загальний інтерфейс.

3

Приклад: розрахунок знижки

За акцією, за кількістю покупок, за купоном — і застосовувати їх, не змінюючи основну логіку.



Рисунок Б.4 – Визначення патерну Strategy

Структура патерну Strategy

Учасники патерну:

Strategy (Інтерфейс стратегії)

Описує загальний метод, який мають реалізувати всі алгоритми.

ConcreteStrategy (Конкретна стратегія)

Клас, який реалізує конкретний варіант алгоритму.

Context (Контекст)

Об'єкт, який використовує певну стратегію. Він не знає, яку саме реалізацію використовує, лише викликає метод через інтерфейс.

Цей підхід дозволяє нам легко додавати нові стратегії без зміни вже написаного коду.



Рисунок Б.5 – Структура патерну Strategy



Приклад використання у житті

Уявімо, що ми розробляємо застосунок доставки товарів. Користувач може обрати один із варіантів:



**Доставка
поштою**



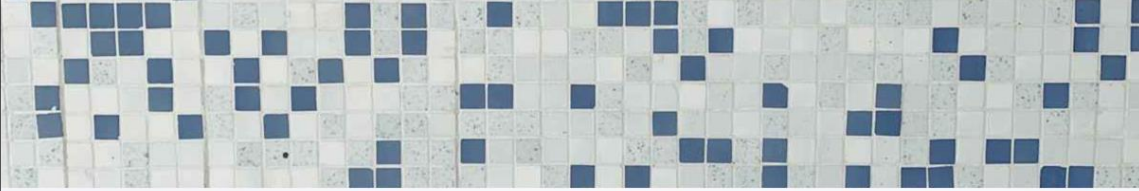
Кур'єром



Самовивіз

Кожен з варіантів має власний алгоритм розрахунку вартості й часу. Ми можемо реалізувати кожну стратегію в окремому класі, а система буде динамічно перемикається між ними в залежності від вибору користувача. Так ми не дублюємо код і легко додаємо нові методи доставки у майбутньому.

Рисунок Б.6 – Приклад використання у житті



Простий приклад реалізації на Python

```
class Strategy:
    def execute(self, a, b):
        pass
class Add(Strategy):
    def execute(self, a, b):
        return a + b
class Subtract(Strategy):
    def execute(self, a, b):
        return a - b

class Context:
    def __init__(self, strategy):
        self.strategy = strategy
    def set_strategy(self, strategy):
        self.strategy = strategy
    def execute(self, a, b):
        return self.strategy.execute(a, b)
```

У цьому прикладі є інтерфейс Strategy, дві реалізації Add і Subtract, а також клас Context, який використовує одну з цих стратегій. Ми можемо змінювати поведінку Context без його перегисування.

Інтерфейс Strategy

Визначає метод `execute`.

Реалізації Add та Subtract

Конкретні стратегії для додавання та віднімання.

Клас Context

Використовує обрану стратегію для виконання операції.




Рисунок Б.7 – Простий приклад реалізації на Python

Переваги та недоліки

Переваги:

- Можна легко додавати нові алгоритми без зміни існуючого коду.
- Кожен алгоритм ізольований у своєму класі — легше тестувати та підтримувати.
- Сприяє дотриманню принципів SOLID, зокрема відкритості/закритості (Open/Closed Principle).

Недоліки:

- Зростає кількість класів — кожна стратегія потребує окремого класу.
- Контексту потрібно знати, яку стратегію застосувати — це може вимагати додаткової логіки або конфігурації.

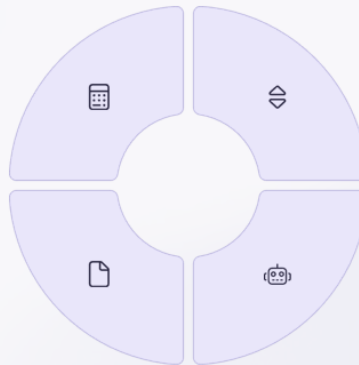
Рисунок Б.8 – Переваги та недоліки

Застосування патерну Strategy

Strategy можна застосовувати в багатьох ситуаціях:

**Розрахунок податків,
доставки або знижок**
Коли є кілька варіантів розрахунку.

Обробка файлів
Різні стратегії стиснення або
шифрування.



Сортування
Вибір між різними алгоритмами
сортування.

**Штучний інтелект у
відеоіграх**
Зміна поведінки ворогів.

Усі ці приклади об'єднує одна ідея — є кілька варіантів поведінки, які можна інкапсулювати і змінювати динамічно.

Рисунок Б.9 – Застосування патерну Strategy

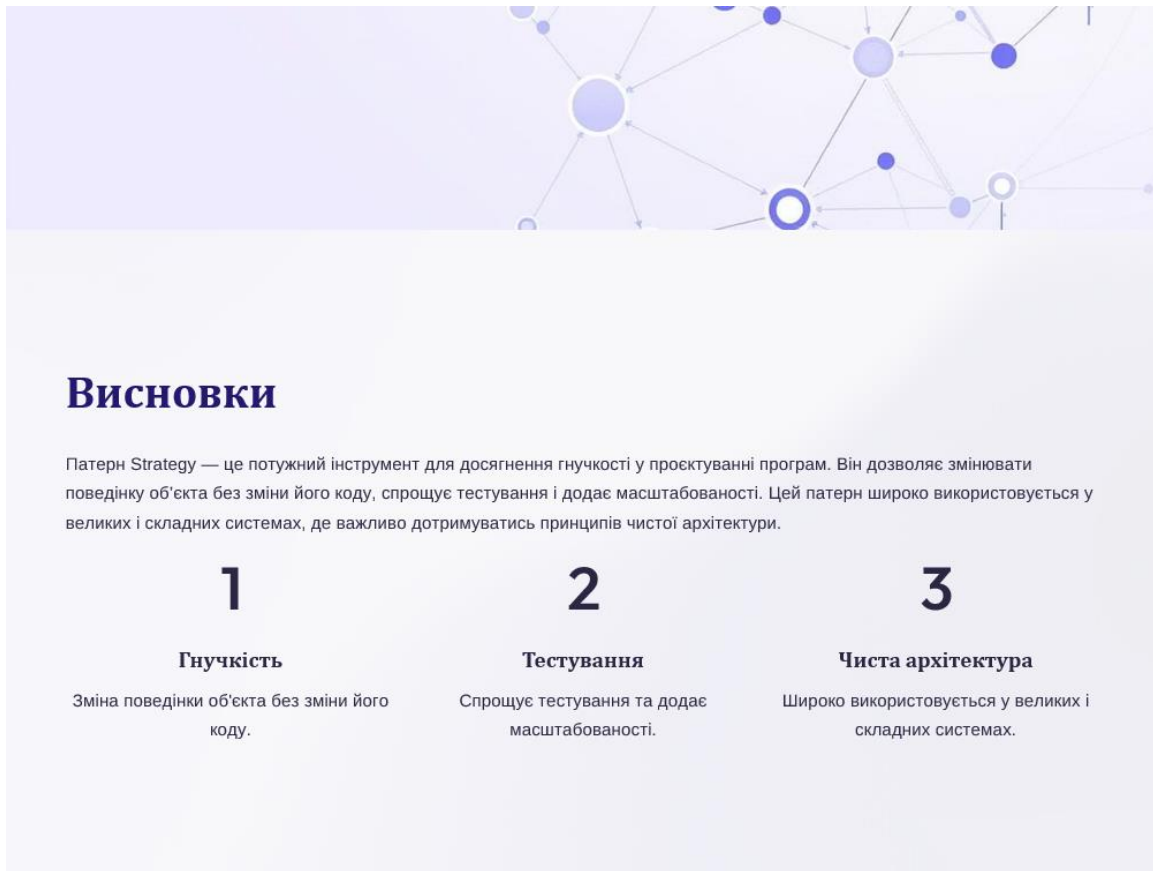


Рисунок Б.10 – Висновки

ДОДАТОК В

Приклад коду на Python

```
1  class Strategy:
2      def execute(self, a, b):
3          pass
4
5  class Add(Strategy):
6      def execute(self, a, b):
7          return a + b
8
9  class Subtract(Strategy):
10     def execute(self, a, b):
11         return a - b
12
13 class Context:
14     def __init__(self, strategy):
15         self.strategy = strategy
16
17     def set_strategy(self, strategy):
18         self.strategy = strategy
19
20     def execute(self, a, b):
21         return self.strategy.execute(a, b)
```