ММП 2025/2026

Викладач Канцедал Георгій Олегович

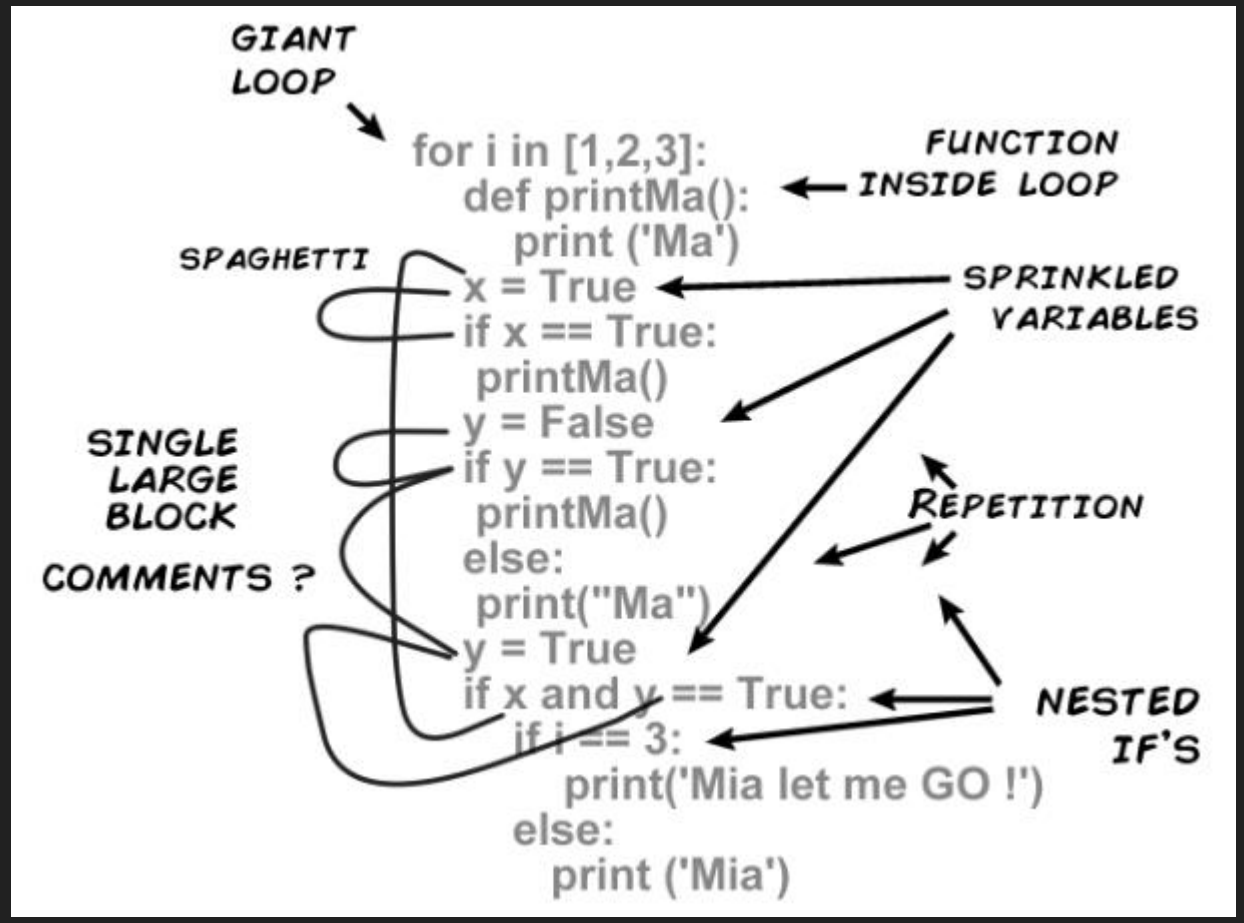
Патерни проектування

Отже ми вже взнали що пайтон підтримує майже всі доступні уяві парадигми і це створює небезпеку хаосу в коді. Однак завдяки принципам солід життя повинно ставати краще. Використання принципів програмування та поєднання парадигм призвело до появи стандартних патернів програмування.

Патерн проєктування — це **типове рішення** певної проблеми під час розробки програмної архітектури.

Основні особливості патернів:

- Не є готовим кодом, а лише концепцією, яку потрібно адаптувати під конкретні потреби.
- Допомагає стандартизувати підхід до вирішення типових завдань у розробці ПЗ.
- Полегшує підтримку та масштабування коду, роблячи його більш зрозумілим для інших розробників.



Історія патернів проектування

Як з’явилася концепція патернів?

- **Кристофер Александер** — перший, хто описав концепцію патернів, але не в програмуванні! У 1977 році він написав книгу *«A Pattern Language»*, де розглядав **архітектурні рішення** для проєктування міст і будівель. У книзі було запропоновано **«мову шаблонів»** для архітектури, де **кожен патерн вирішував конкретну проблему** (наприклад, «Якої висоти повинні бути вікна?» або «Скільки зелених зон потрібно в мікрорайоні?»). Ця ідея надихнула програмістів!

У 1994 році **Еріх Гамма, Річард Хелм, Ральф Джонсон і Джон Вліссидес** написали книгу: *«Design Patterns: Elements of Reusable Object-Oriented Software»*.

- Вона містила **23 основні патерни** для **об'єктно-орієнтованого програмування (ООП)**.
- Оригінальна назва книги була занадто довгою, тому її стали називати **«book by the Gang of Four»**, або просто **«GoF book»**.

Види патернів проєктування

Патерни проєктування поділяються на **три основні категорії**:

- **Порождувальні (Creational)** – відповідають за створення об'єктів.
- **Структурні (Structural)** – визначають, як компоненти взаємодіють між собою.
- **Поведенкові (Behavioral)** – описують способи взаємодії між об'єктами.

Основні рівні патернів

- **ідіоми (Idioms)** – найпростіший рівень, характерний для конкретної мови програмування.
- **Патерни проєктування (Design Patterns)** – вирішують типові проблеми ООП.
- **Архітектурні патерни (Architectural Patterns)** – формують структуру всієї програми.

Види патернів проєктування

Ідіоми (Idioms) – найнижчий рівень

- Це **стилі написання коду**, які характерні для конкретної мови програмування.
- Не є універсальними – працюють лише в рамках однієї мови.

Приклади:

- **Python:** використання with open() для роботи з файлами.
- **C++:** RAII (Resource Acquisition Is Initialization) – автоматичне управління ресурсами.
- **JavaScript:** використання замикань (closure).

Патерни проєктування (Design Patterns) – середній рівень

- Універсальні рішення для побудови гнучкої та масштабованої архітектури.
- Використовуються в ООП, незалежно від конкретної мови.

Приклади:

- **Порождувальні** – Singleton, Factory Method, Builder.
- **Структурні** – Adapter, Decorator, Facade.
- **Поведенкові** – Observer, Strategy, Command.

Види патернів проєктування

Архітектурні патерни (Architectural Patterns) – найвищий рівень

- Патерни, які визначають **структуру всієї системи**.
- Використовуються в **великих проєктах** та **системній архітектурі**.

Приклади:

- **MVC (Model-View-Controller)** – розділення логіки додатку на модель, представлення та контролер.
- **MVVM (Model-View-ViewModel)** – покращена версія MVC для UI/UX.
- **Microservices (Мікросервіси)** – розбиття системи на незалежні сервіси.
- **Event-Driven Architecture (Подієво-орієнтована архітектура)** – система реагує на події.
- **Layered Architecture (Багаторівнева архітектура)** – поділ системи на рівні (UI, бізнес-логіка, база даних).

Одинак (Singleton)

Одинак (Singleton) — це **порождувальний патерн**, який:

- **Гарантує, що у класу є тільки один екземпляр.**
- **Надає глобальну точку доступу до цього екземпляра.**

Переваги:

✔ Зручний для реалізації **глобального доступу** (наприклад, для керування ресурсами, логування, конфігурації).

✔ Гарантує, що існує **тільки один екземпляр** об’єкта.

Недоліки:

 Порушує модульність – клас, який залежить від Singleton, важко використовувати в інших програмах.

 Ускладнює тестування – при написанні юніт-тестів доводиться **емулювати одинак**, що робить тестування менш гнучким.

 Може стати точкою вузького місця (bottleneck) – якщо багато частин коду звертаються до Singleton, він може стати **«глобальною змінною»** з поганими наслідками для продуктивності.

Одинак (Singleton)

```python
class Singleton:
    _instance = None  # Приватна змінна для збереження єдиного екземпляра

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

# Використання
singleton1 = Singleton()
singleton2 = Singleton()

print(singleton1 is singleton2)  # ✅ True (це один і той самий об'єкт)
```

```python
def singleton(cls):
    instances = {}

    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]

    return get_instance

@singleton
class Config:
    def __init__(self):
        self.settings = {}

# Використання
config1 = Config()
config2 = Config()

print(config1 is config2)  # ✅ True (це один і той самий екземпляр)
```

Адаптер (Adapter)

Адаптер (Adapter) — це **структурний патерн**, який дозволяє **«подружити»** несумісні об'єкти, тобто змінити їхній інтерфейс так, щоб вони могли працювати разом.

Коли використовувати патерн Адаптер?

- Коли потрібно використовувати **об'єкт з несумісним інтерфейсом**, але ви не можете (або не хочете) змінювати його код.
- Коли ви працюєте з **застарілим кодом або сторонніми бібліотеками**, які мають інший інтерфейс.
- Коли необхідно **об'єднати новий код зі старою системою** без значних змін.

```python
class PrinterAdapter:
    """Адаптер, який перетворює інтерфейс старого принтера на сучасний."""
    def __init__(self, old_printer):
        self.old_printer = old_printer

    def print_text(self, text):
        """Перетворює виклик на метод старого класу."""
        self.old_printer.old_print(text)

# Використання:
old_printer = OldPrinter()
adapted_printer = PrinterAdapter(old_printer)

print_document(adapted_printer, "Hello")  # ✅ Тепер працює через адаптер
```

Міст (Bridge)

Міст (Bridge) — це **структурний патерн**, який **відокремлює абстракцію від її реалізації**, дозволяючи розвивати їх **незалежно** одна від одної.

Коли використовувати патерн Міст?

- **Якщо в класі з'являється надто багато підкласів** – краще розділити його на дві ієрархії.
- **Якщо абстракція та реалізація змінюються незалежно** – наприклад, є **різні види пристроїв** і **різні способи їхнього управління**.
- **Якщо хочете уникнути жорсткої прив'язки між абстракцією та реалізацією**, щоб легко їх змінювати.

```python
class TV:
    def on(self):
        print("Телевізор увімкнено")

    def off(self):
        print("Телевізор вимкнено")

class Radio:
    def on(self):
        print("Радіо увімкнено")

    def off(self):
        print("Радіо вимкнено")

# Дублювання коду в кожному пульті!
class RemoteForTV:
    def __init__(self, device: TV):
        self.device = device

    def toggle_power(self):
        print("Клац! Перемикаємо живлення на телевізорі")
        self.device.on()

class RemoteForRadio:
    def __init__(self, device: Radio):
        self.device = device

    def toggle_power(self):
        print("Клац! Перемикаємо живлення на радіо")
        self.device.on()
```

Міст (Bridge)

Рішення через патерн Міст

- Ми розділимо клас на дві **ієрархії**:
- **Абстракція** (Remote) – пульт управління.
- **Реалізація** (Device) – конкретні пристрої

```python
class Device:
    def on(self):
        pass

    def off(self):
        pass

# Реалізація для телевізора
class TV(Device):
    def on(self):
        print("Телевізор увімкнено")

    def off(self):
        print("Телевізор вимкнено")

# Реалізація для радіо
class Radio(Device):
    def on(self):
        print("Радіо увімкнено")

    def off(self):
        print("Радіо вимкнено")

# Абстракція "Пульт"
class Remote:
    def __init__(self, device: Device):
        self.device = device  # ✅ Передаємо пристрій як залежність

    def toggle_power(self):
        print("Клац! Перемикаємо живлення")
        self.device.on()

# Додаткове розширення – "Розумний пульт"
class SmartRemote(Remote):
    def mute(self):
        print("Пристрій вимкнено на беззвучний режим")
```

Компоновщик (Composite)

Компоновщик (Composite) — це **структурний патерн**, який дозволяє працювати з групою об'єктів так само, як із єдиним об'єктом. **Основна ідея** – організувати об'єкти у **деревоподібну структуру** та надати єдиний інтерфейс для роботи з **одиничними об'єктами та групами об'єктів**.

Коли використовувати патерн Компоновщик?

- **Коли потрібно працювати з ієрархічними структурами (деревами).**
- **Коли клієнтський код повинен працювати однаково як із простими, так і зі складними об'єктами.**
- **Коли потрібно динамічно будувати об'єкти та групувати їх у колекції.**

```python
class Component(ABC):
    """
    Базовий інтерфейс для всіх об'єктів у композиції.
    """

    @abstractmethod
    def operation(self) -> str:
        pass


class Leaf(Component):
    """
    Лист – це кінцевий елемент дерева, який не має дочірніх елементів.
    """

    def operation(self) -> str:
        return "Лист"


class Composite(Component):
    """
    Композит містить у собі кілька компонентів (листів або інших композитів).
    """

    def __init__(self) -> None:
        self._children: List[Component] = []

    def add(self, component: Component) -> None:
        """Додає підкомпонент у композицію."""
        self._children.append(component)

    def remove(self, component: Component) -> None:
        """Видаляє підкомпонент із композиції."""
        self._children.remove(component)

    def operation(self) -> str:
        """Виконує операцію для всіх дочірніх компонентів."""
        results = [child.operation() for child in self._children]
        return f"Гілка({'+'.join(results)})"


def client_code(component: Component) -> None:
    """
    Клієнтський код працює однаково як із простими, так і зі складними об'єктами.
    """

    print(f"RESULT: {component.operation()}")
```

Компоновщик (Composite)

Патерн **Компоновщик (Composite)** використовується **в усіх задачах, що пов’язані з побудовою деревоподібних структур**.

Найпоширеніші сценарії:

- **Ієрархічні об'єкти** – наприклад, **файлова система (папки та файли)**.
- **Складені елементи GUI** – кнопки, панелі, вікна, які можуть містити інші елементи.
- **Графові структури** – сцени в іграх, організаційні діаграми.

Ознаки використання патерну Компоновщик

- **Якщо з об'єктів будується древовидна структура**.
- **Якщо з усіма елементами (і окремими, і груповими) працюють через один і той же інтерфейс**.
- **Якщо потрібно дозволити клієнту працювати з групами об'єктів так само, як із одиночними об'єктами**.

```python
class Component(ABC):
    """
    Базовий інтерфейс для всіх об'єктів у композиції.
    """

    @abstractmethod
    def operation(self) -> str:
        pass


class Leaf(Component):
    """
    Лист – це кінцевий елемент дерева, який не має дочірніх елементів.
    """

    def operation(self) -> str:
        return "Лист"


class Composite(Component):
    """
    Композит містить у собі кілька компонентів (листів або інших композитів).
    """

    def __init__(self) -> None:
        self._children: List[Component] = []

    def add(self, component: Component) -> None:
        """Додає підкомпонент у композицію."""
        self._children.append(component)

    def remove(self, component: Component) -> None:
        """Видаляє підкомпонент із композиції."""
        self._children.remove(component)

    def operation(self) -> str:
        """Виконує операцію для всіх дочірніх компонентів."""
        results = [child.operation() for child in self._children]
        return f"Гілка({'+'.join(results)})"


def client_code(component: Component) -> None:
    """
    Клієнтський код працює однаково як із простими, так і зі складними об'єктами.
    """

    print(f"RESULT: {component.operation()}")
```

Декоратор (Decorator)

Декоратор (Decorator) — це **структурний патерн**, який дозволяє **додавати нову поведінку об'єктам на льоту**, обгортаючи їх у спеціальні об'єкти-обгортки (**wrapper**).

Коли використовувати патерн Декоратор?

- **Коли потрібно динамічно додавати нові можливості об'єкту без зміни його коду.**
- **Коли є кілька незалежних доповнень, і їх можна комбінувати у будь-якому порядку.**
- **Коли не можна (або не хочеться) використовувати спадкування для додавання поведінки.**

```python
from abc import ABC, abstractmethod

# Базовий інтерфейс
class Coffee(ABC):
    """Інтерфейс для всіх типів кави."""

    @abstractmethod
    def cost(self) -> float:
        pass

# Реалізація базового об'єкта
class SimpleCoffee(Coffee):
    """Звичайна кава без добавок."""

    def cost(self) -> float:
        return 5

# Базовий клас Декоратора
class CoffeeDecorator(Coffee):
    """Абстрактний клас-декоратор, який розширює поведінку кави."""

    def __init__(self, coffee: Coffee):
        self._coffee = coffee

    def cost(self) -> float:
        return self._coffee.cost()

# Декоратори
class MilkDecorator(CoffeeDecorator):
    """Додає молоко до кави."""

    def cost(self) -> float:
        return super().cost() + 2

class SugarDecorator(CoffeeDecorator):
    """Додає цукор до кави."""

    def cost(self) -> float:
        return super().cost() + 1
```

Декоратор (Decorator)

Декоратор (Decorator) — це **структурний патерн**, який дозволяє **додавати нову поведінку об'єктам на льоту**, обгортаючи їх у спеціальні об'єкти-обгортки (**wrapper**).

Коли використовувати патерн Декоратор?

- **Коли потрібно динамічно додавати нові можливості об'єкту без зміни його коду.**
- **Коли є кілька незалежних доповнень, і їх можна комбінувати у будь-якому порядку.**
- **Коли не можна (або не хочеться) використовувати спадкування для додавання поведінки.**

```python
from abc import ABC, abstractmethod

# Базовий інтерфейс
class Coffee(ABC):
    """Інтерфейс для всіх типів кави."""

    @abstractmethod
    def cost(self) -> float:
        pass

# Реалізація базового об'єкта
class SimpleCoffee(Coffee):
    """Звичайна кава без добавок."""

    def cost(self) -> float:
        return 5

# Базовий клас Декоратора
class CoffeeDecorator(Coffee):
    """Абстрактний клас-декоратор, який розширює поведінку кави."""

    def __init__(self, coffee: Coffee):
        self._coffee = coffee

    def cost(self) -> float:
        return self._coffee.cost()

# Декоратори
class MilkDecorator(CoffeeDecorator):
    """Додає молоко до кави."""

    def cost(self) -> float:
        return super().cost() + 2

class SugarDecorator(CoffeeDecorator):
    """Додає цукор до кави."""

    def cost(self) -> float:
        return super().cost() + 1
```

Фасад (Facade)

Фасад (Facade) — це **структурний патерн**, який надає **спрощений інтерфейс** до складної системи об'єктів, бібліотеки або фреймворку.

Коли використовувати патерн Фасад?

- **Коли система занадто складна для прямого використання (багато взаємозалежних класів).**
- **Коли потрібно приховати складні API бібліотек або фреймворків.**
- **Коли необхідно створити єдину точку входу для роботи з підсистемою.**

```python
class AudioDecoder:
    """Декодує аудіофайли в потрібний формат."""
    def decode(self, file):
        print(f"Декодуємо {file}")

class AudioPlayer:
    """Відтворює аудіофайли."""
    def play(self, decoded_file):
        print(f"Відтворюємо {decoded_file}")

class Equalizer:
    """Обробляє звук через еквалайзер."""
    def apply_settings(self, decoded_file):
        print(f"Застосовуємо налаштування еквалайзера до {decoded_file}")

# Використання без фасаду:
decoder = AudioDecoder()
player = AudioPlayer()
equalizer = Equalizer()

file = "song.mp3"
decoder.decode(file)
equalizer.apply_settings(file)
player.play(file)
```

Легковаговик (Flyweight)

Легковаговик (Flyweight) — це **структурний патерн**, який допомагає **зменшити використання пам’яті**, розділяючи **спільний стан** між багатьма об'єктами.

Коли використовувати патерн Легковаговик?

- **Якщо у програмі створюється велика кількість об'єктів з однаковими даними** (наприклад, символи в текстовому редакторі).
- **Якщо потрібно оптимізувати використання пам'яті** за рахунок розділення стану між об'єктами.
- **Якщо велика частина інформації в об'єктах повторюється і її можна зберігати окремо**.

```python
class TreeType:
    """Клас, що містить спільний стан (легковаговик)."""

    def __init__(self, species, color, texture):
        self.species = species
        self.color = color
        self.texture = texture

    def __str__(self):
        return f"{self.species} ({self.color}, {self.texture})"


class TreeFactory:
    """Фабрика, що кешує створені типи дерев."""

    _tree_types = {}

    @staticmethod
    def get_tree_type(species, color, texture):
        """Перевіряє, чи вже є такий тип дерева, і повертає його, або створює новий."""
        key = (species, color, texture)
        if key not in TreeFactory._tree_types:
            TreeFactory._tree_types[key] = TreeType(species, color, texture)
        return TreeFactory._tree_types[key]


class Tree:
    """Клас дерева, що містить лише унікальний стан (координати)."""

    def __init__(self, tree_type: TreeType, x, y):
        self.tree_type = tree_type  # Посилання на спільний об'єкт
        self.x = x
        self.y = y

    def display(self):
        print(f"Дерево {self.tree_type} розташоване за координатами ({self.x}, {self.y})")


# Створюємо ліс із Легковаговиком
forest = []
for x in range(100):
    for y in range(100):
        tree_type = TreeFactory.get_tree_type("Дуб", "Зелений", "Груба")
        forest.append(Tree(tree_type, x, y))

# Виводимо одне дерево
forest[0].display()
```

Замісник (Proxy)

Замісник (Proxy) — це об'єкт, який **виступає посередником** між клієнтом і реальним об'єктом. **Основна ідея** – **перехоплювати виклики** клієнта, виконувати додаткові функції (**кешування, контроль доступу, логування**) та передавати запит до реального об'єкта.

Коли використовувати патерн Замісник?

- **Коли необхідно додати контроль доступу** – наприклад, перевіряти права користувача перед передачею запиту.
- **Коли потрібно кешувати результати запитів** – зменшення навантаження на реальний об'єкт.
- **Коли необхідно відкладене створення (lazy initialization)** – об'єкт створюється лише під час першого виклику.
- **Коли потрібно вести логування або збір статистики**.

```python
class Video:
    """Клас відеофайлу, який завантажує файл з диска."""

    def __init__(self, filename):
        self.filename = filename
        print(f"Завантаження відео {filename}...")

    def play(self):
        print(f"Відтворення відео {self.filename}.")

class VideoProxy:
    """Замісник для відео, який виконує відкладене завантаження."""

    def __init__(self, filename):
        self.filename = filename
        self._video = None  # Об'єкт ще не створений

    def play(self):
        """Лише під час першого виклику створюється реальний об'єкт Video."""
        if self._video is None:
            self._video = Video(self.filename)  # Завантаження тільки тут!
        self._video.play()

# Використання:
video = VideoProxy("movie.mp4")  # ✅ Відео ще НЕ завантажується
video.play()  # ✅ Тепер відбувається завантаження та відтворення
video.play()  # ✅ Завантаження не відбувається вдруге, тільки відтворення
```

Ланцюг обов'язків (Chain of Responsibility)

Ланцюг обов'язків (Chain of Responsibility) — це **поведінковий патерн**, який дозволяє **послідовно передавати запит між обробниками**, поки один із них не зможе його обробити. **Основна ідея** – кожен обробник **або обробляє запит, або передає його далі по ланцюгу**.

Коли використовувати патерн Ланцюг обов'язків?

- **Якщо необхідно уникнути жорсткої прив’язки між відправником і отримувачем запиту**.
- **Якщо є кілька потенційних обробників запиту, і їх порядок має бути гнучким**.
- **Якщо потрібно дати можливість кільком обробникам працювати над одним запитом** (наприклад, у випадку системи логування або безпеки).

```python
from abc import ABC, abstractmethod

class Handler(ABC):
    """Базовий клас обробника у ланцюгу відповідальності."""

    def __init__(self, next_handler=None):
        self._next_handler = next_handler

    @abstractmethod
    def handle(self, level, message):
        """Метод, який повинен бути реалізований у підкласах."""
        pass

class InfoLogger(Handler):
    """Обробник для рівня INFO."""

    def handle(self, level, message):
        if level == "INFO":
            print(f"[INFO]: {message}")
        elif self._next_handler:
            self._next_handler.handle(level, message)

class WarningLogger(Handler):
    """Обробник для рівня WARNING."""

    def handle(self, level, message):
        if level == "WARNING":
            print(f"[WARNING]: {message}")
        elif self._next_handler:
            self._next_handler.handle(level, message)

class ErrorLogger(Handler):
    """Обробник для рівня ERROR."""

    def handle(self, level, message):
        if level == "ERROR":
            print(f"[ERROR]: {message}")
        elif self._next_handler:
            self._next_handler.handle(level, message)

# Побудова ланцюга обробників:
logger_chain = InfoLogger(WarningLogger(ErrorLogger()))
```

Кінець

.

Більше про патерни https://refactoring.guru/ru/design-patterns/python