Шаблони Проектування (Design Patterns)

Шаблони проектування — це загальні рекомендації та архітектурні концепції, які допомагають розробникам створювати програмне забезпечення, що є ефективним, легким для розуміння, та легко піддається розширенню та змінам. Це перевірені рішення для типових проблем, що виникають у процесі розробки.

Огляд Популярних Шаблонів

1. Singleton (Одинак)

- **Що це?** Гарантує, що клас має тільки один екземпляр у всій програмі.
- **Як виглядає?** Клас має приватний конструктор і статичний метод, який повертає єдиний екземпляр. Якщо екземпляр ще не існує, він створюється; інакше повертається вже існуючий.
- **Де використовують?** Для глобальних, унікальних ресурсів:
  - **Підключення до бази даних:** Щоб уникнути створення кількох з'єднань, які споживають пам'ять.
  - **Логгер:** Для централізованого запису подій в один файл.
  - **Менеджер налаштувань:** Для забезпечення єдиного доступу до конфігурації програми.

```
import datetime

# --- Реалізація ---
class ConfigManager:
    """Клас-одинак для керування налаштуваннями."""
    _instance = None  # Приватна змінна для зберігання єдиного екземпляра
    _initialized = False # Допоміжна змінна, щоб уникнути повторної ініціалізації

    def __new__(cls, *args, **kwargs):
        """
        Метод __new__ викликається перед __init__ і відповідає
        за створення нового екземпляра. Ми перехоплюємо його,
        щоб контролювати створення об'єкта.
        """
        if cls._instance is None:
            # Якщо екземпляра ще не існує, створюємо його.
            print("Створюється новий екземпляр ConfigManager.")
            cls._instance = super(ConfigManager, cls).__new__(cls)
        # Повертаємо єдиний екземпляр, незалежно від того, чи він щойно
створений, чи вже існує.
        return cls._instance
```

```
    def __init__(self, settings: dict):
        """
        Конструктор, який ініціалізує об'єкт.
        Викликається кожного разу, але ми перевіряємо, чи він вже був
ініціалізований.
        """
        if not self._initialized:
            self.settings = settings
            self._initialized = True
            print("ConfigManager успішно ініціалізовано.")

    def get_setting(self, key: str):
        """Повертає значення налаштування за ключем."""
        return self.settings.get(key)

    def update_setting(self, key: str, value):
        """Оновлює значення налаштування."""
        self.settings[key] = value
        print(f"Налаштування '{key}' оновлено до '{value}'.")

# --- Демонстрація використання ---
print("\n--- Демонстрація: Створюємо перший екземпляр ---")
config1 = ConfigManager({'app_name': 'MyAwesomeApp', 'version': '1.0'})
print(f"Назва додатку: {config1.get_setting('app_name')}")

print("\n--- Демонстрація: Спроба створити другий екземпляр ---")
# Тут конструктор буде викликаний знову, але __new__ поверне старий екземпляр
config2 = ConfigManager({'db_host': 'localhost'})
print(f"Другий об'єкт 'config2' спробував змінити налаштування, але його
'settings' не застосувалися.")
print(f"Назва додатку через config2: {config2.get_setting('app_name')}")

print("\n--- Перевірка: чи це один і той самий об'єкт? ---")
print(f"Чи є config1 і config2 одним об'єктом? {config1 is config2}")
print("Це один і той самий об'єкт, бо Singleton гарантує унікальність.")

print("\n--- Демонстрація: Оновлення налаштувань ---")
# Якщо оновити налаштування через один об'єкт, це відобразиться і на іншому.
config1.update_setting('version', '2.0')
print(f"Версія додатку через config1: {config1.get_setting('version')}")
print(f"Версія додатку через config2: {config2.get_setting('version')}")
```

2. Factory Method (Фабричний метод)

- **Що це?** Визначає метод для створення об'єктів у базовому класі, але дозволяє підкласам вирішувати, який конкретний клас інстанціювати.
- **Як виглядає?** Є абстрактний Creator (Творець) з factory_method, який повертає об'єкт. Його підкласи (ConcreteCreator) реалізують цей метод, повертаючи конкретні Product (Продукт).
- **Де використовують?** Коли потрібно гнучко створювати об'єкти, не прив'язуючись до конкретного типу:
    - **Керування різними типами об'єктів:** Наприклад, програма може створювати різні види документів (PDF, DOCX) залежно від налаштувань, не змінюючи основний код.

```
# --- Реалізація ---
from abc import ABC, abstractmethod

# Інтерфейс продукту, який буде створювати фабрика
class Notification(ABC):
    """Абстрактний клас-продукт. Визначає загальний інтерфейс."""
    @abstractmethod
    def send(self, message: str):
        pass

# Конкретні продукти
class SMSNotification(Notification):
    """Конкретний продукт: SMS-повідомлення."""
    def send(self, message: str):
        print(f"Надсилаю SMS: '{message}'")

class EmailNotification(Notification):
    """Конкретний продукт: Email-повідомлення."""
    def send(self, message: str):
        print(f"Надсилаю Email: '{message}'")

# Інтерфейс творця (фабрики)
class NotificationFactory(ABC):
    """Абстрактний клас-творець (фабрика). Визначає фабричний метод."""
    @abstractmethod
    def create_notification(self) -> Notification:
        """Фабричний метод, що має бути реалізований в підкласах."""
        pass

    def notify(self, message: str):
        """Метод, який використовує продукт, створений фабричним методом."""
```

```
        notification = self.create_notification()
        notification.send(message)

# Конкретні творці
class SMSFactory(NotificationFactory):
    """Конкретна фабрика для створення SMS-повідомлень."""
    def create_notification(self) -> Notification:
        return SMSNotification()

class EmailFactory(NotificationFactory):
    """Конкретна фабрика для створення Email-повідомлень."""
    def create_notification(self) -> Notification:
        return EmailNotification()

# --- Демонстрація використання ---
def client_code(factory: NotificationFactory, message: str):
    """
    Клієнтський код, який працює з фабрикою, не знаючи, який саме
    конкретний тип повідомлення вона створить.
    """
    print("Клієнт: Я використовую фабрику, щоб надіслати повідомлення.")
    factory.notify(message)

print("\n--- Демонстрація: Надсилання SMS ---")
sms_factory = SMSFactory()
client_code(sms_factory, "Привіт! Це тестове SMS.")

print("\n--- Демонстрація: Надсилання Email ---")
email_factory = EmailFactory()
client_code(email_factory, "Subject: Тестовий лист\nBody: Привіт! Це тестовий
email.")
```

3. Observer (Спостерігач)

- **Що це?** Дозволяє одному об'єкту (Subject, або Observable) сповіщати інші об'єкти (Observers) про зміни свого стану.
- **Як виглядає?** Subject має методи для реєстрації, видалення та сповіщення Observer'ів. Кожен Observer має метод update(), який викликається Subject при зміні стану.
- **Де використовують?** Для реалізації системи сповіщень:
    - **GUI-компоненти:** Кнопка може бути Subject, а вікно — Observer. При натисканні кнопки, вікно отримує сповіщення і оновлюється.
    - **Системи подій:** Коли подія в одній частині системи повинна запускати дії

в інших, не пов'язаних частинах.

```python
class Subject:
    """Об'єкт, що сповіщає про зміни."""
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        self._observers.append(observer)

    def detach(self, observer):
        self._observers.remove(observer)

    def notify(self, message):
        for observer in self._observers:
            observer.update(message)

class Observer(ABC):
    """Інтерфейс спостерігача."""
    @abstractmethod
    def update(self, message):
        pass

class ConcreteObserver(Observer):
    """Конкретний спостерігач, який реагує на повідомлення."""
    def __init__(self, name):
        self.name = name

    def update(self, message):
        print(f"Спостерігач '{self.name}' отримав повідомлення: {message}")

# Приклад використання
subject = Subject()
observer_a = ConcreteObserver("Observer A")
observer_b = ConcreteObserver("Observer B")

subject.attach(observer_a)
subject.attach(observer_b)

subject.notify("Стан змінився!")
```

4. Decorator (Декоратор)

- **Що це?** Дозволяє динамічно приєднувати нові обов'язки до об'єкта,

обгортаючи його іншими об'єктами-декораторами.

- **Як виглядає?** Існують загальний компонент та його конкретна реалізація. Декоратори наслідують загальний компонент і містять посилання на об'єкт, який вони прикрашають.
- **Де використовують?** Коли потрібно додати нові функції без зміни існуючого коду:
  - **Додавання функціональності:** Наприклад, до текстового редактора можна додати функції перевірки орфографії, підсвічування синтаксису, форматування тексту, динамічно обгортаючи його різними декораторами.

```
# Компонент (інтерфейс)
class Coffee(ABC):
    @abstractmethod
    def get_cost(self):
        pass

    @abstractmethod
    def get_ingredients(self):
        pass

# Конкретний компонент
class SimpleCoffee(Coffee):
    def get_cost(self):
        return 5
    def get_ingredients(self):
        return "Кава"

# Декоратор (абстрактний клас)
class CoffeeDecorator(Coffee):
    def __init__(self, coffee):
        self._decorated_coffee = coffee

    def get_cost(self):
        return self._decorated_coffee.get_cost()

    def get_ingredients(self):
        return self._decorated_coffee.get_ingredients()

# Конкретні декоратори
class MilkDecorator(CoffeeDecorator):
    def get_cost(self):
        return super().get_cost() + 2
    def get_ingredients(self):
```

```
        return super().get_ingredients() + ", Молоко"

class SugarDecorator(CoffeeDecorator):
    def get_cost(self):
        return super().get_cost() + 1
    def get_ingredients(self):
        return super().get_ingredients() + ", Цукор"

# Приклад використання
my_coffee = SimpleCoffee()
print(f"Просто кава: вартість {my_coffee.get_cost()}, склад:
{my_coffee.get_ingredients()}")

my_coffee = MilkDecorator(my_coffee)
print(f"Кава з молоком: вартість {my_coffee.get_cost()}, склад:
{my_coffee.get_ingredients()}")

my_coffee = SugarDecorator(my_coffee)
print(f"Кава з молоком і цукром: вартість {my_coffee.get_cost()}, склад:
{my_coffee.get_ingredients()}")
```

5. Adapter (Адаптер)

- **Що це?** Дозволяє об'єктам з несумісними інтерфейсами працювати разом.
- **Як виглядає?** Клас-адаптер містить у собі об'єкт, який потрібно "адаптувати", і перетворює його інтерфейс до того, який очікує клієнт.
- **Де використовують?** Для інтеграції:
    - **Робота зі старим кодом (legacy):** Коли потрібно використовувати старий компонент у новій системі, яка очікує інший інтерфейс.
    - **Інтеграція зовнішніх бібліотек:** Коли ви хочете використовувати бібліотеку, яка має незручний для вас інтерфейс.

```
# Несумісний клас (Legacy)
class OldLogger:
    def log_message(self, text):
        print(f"OLD_LOGGER: {text}")

# Інтерфейс, який очікує клієнт
class NewLogger(ABC):
    @abstractmethod
    def log(self, text):
        pass
```

```
# Клас-Адаптер
class OldLoggerAdapter(NewLogger):
    def __init__(self, old_logger_instance):
        self._old_logger = old_logger_instance

    def log(self, text):
        self._old_logger.log_message(f"Адаптовано: {text}")

# Приклад використання
def client_code(logger_instance: NewLogger):
    logger_instance.log("Це повідомлення від клієнта.")

old_logger = OldLogger()
adapter = OldLoggerAdapter(old_logger)
client_code(adapter)
```

6. Strategy (Стратегія)

- **Що це?** Дозволяє визначати сімейство алгоритмів, поміщати їх у окремі класи та робити об'єкти взаємозамінними.
- **Як виглядає?** Є загальний інтерфейс Strategy, який реалізують ConcreteStrategy (конкретні алгоритми). Об'єкт Context містить посилання на Strategy і делегує йому виконання завдання.
- **Де використовують?** Коли потрібно вибрати один з кількох схожих алгоритмів:
    - **Сортування:** Можна мати різні стратегії сортування (швидке, бульбашкове) і вибирати потрібну під час виконання.
    - **Розрахунок знижок:** Залежно від акції, можна застосувати різні стратегії розрахунку знижки.

```
class SortStrategy(ABC):
    @abstractmethod
    def sort(self, data):
        pass

class BubbleSortStrategy(SortStrategy):
    def sort(self, data):
        print("Використовую бульбашкове сортування.")
        return sorted(data)

class QuickSortStrategy(SortStrategy):
    def sort(self, data):
```

```
        print("Використовую швидке сортування.")
        return sorted(data)

class Context:
    def __init__(self, strategy: SortStrategy):
        self._strategy = strategy

    def set_strategy(self, strategy: SortStrategy):
        self._strategy = strategy

    def do_some_sorting(self, data):
        return self._strategy.sort(data)

# Приклад використання
data_to_sort = [3, 1, 4, 1, 5, 9, 2, 6]
context = Context(BubbleSortStrategy())
print(f"Початкові дані: {data_to_sort}")
print(f"Відсортовані дані: {context.do_some_sorting(data_to_sort)}")

context.set_strategy(QuickSortStrategy())
print(f"Відсортовані дані з іншою стратегією:
{context.do_some_sorting(data_to_sort)}")
```

7. Command (Команда)

- **Що це?** Інкапсулює запит як об'єкт, що дозволяє параметризувати клієнта різними запитами, ставити їх у чергу або логувати.
- **Як виглядає?** Є загальний інтерфейс Command з методом execute(). Конкретні команди (ConcreteCommand) реалізують цей метод, виконуючи певну дію над об'єктом Receiver.
- **Де використовують?** Коли потрібно реалізувати відкладене виконання або скасування дій:
    - **Історія дій (undo/redo):** Кожна дія (наприклад, "копіювати", "вставити") інкапсулюється в об'єкт Command, що дозволяє зберігати їх історію та скасовувати.
    - **Макроси:** Комбінація кількох команд в одну.

```
# Receiver
class Light:
    def turn_on(self):
        print("Світло увімкнено.")
    def turn_off(self):
```

```
        print("Світло вимкнено.")

# Command (інтерфейс)
class Command(ABC):
    @abstractmethod
    def execute(self):
        pass

# Конкретні команди
class TurnOnLightCommand(Command):
    def __init__(self, light: Light):
        self._light = light
    def execute(self):
        self._light.turn_on()

class TurnOffLightCommand(Command):
    def __init__(self, light: Light):
        self._light = light
    def execute(self):
        self._light.turn_off()

# Invoker
class RemoteControl:
    def __init__(self):
        self._command = None
    def set_command(self, command: Command):
        self._command = command
    def press_button(self):
        self._command.execute()

# Приклад використання
light = Light()
remote = RemoteControl()

on_command = TurnOnLightCommand(light)
remote.set_command(on_command)
remote.press_button()

off_command = TurnOffLightCommand(light)
remote.set_command(off_command)
remote.press_button()
```

8. State (Стан)

- **Що це?** Дозволяє об'єкту змінювати свою поведінку, коли змінюється його внутрішній стан. Зовні виглядає так, ніби змінився сам клас об'єкта.
- **Як виглядає?** Клас Context посилається на об'єкт State (Стан). Різні ConcreteState змінюють поведінку Context, і можуть змінювати його поточний стан.
- **Де використовують?** Коли поведінка об'єкта залежить від його стану:
  - **Ігрові персонажі:** Персонаж може мати стани "біжить", "стоїть", "б'ється", і його поведінка (наприклад, швидкість руху) залежить від поточного стану.
  - **Документ:** Документ може бути в стані "чернетка", "на рецензії", "опублікований", і доступні дії залежать від цього стану.

```
class State(ABC):
    @abstractmethod
    def handle(self):
        pass

class ConcreteStateA(State):
    def handle(self):
        print("Об'єкт знаходиться в стані A.")

class ConcreteStateB(State):
    def handle(self):
        print("Об'єкт знаходиться в стані B.")

class Context:
    def __init__(self, state: State):
        self._state = state

    def set_state(self, state: State):
        self._state = state

    def request(self):
        self._state.handle()

# Приклад використання
context = Context(ConcreteStateA())
context.request()

context.set_state(ConcreteStateB())
context.request()
```

9. MVC (Модель-Вид-Контролер)

- **Що це?** Архітектурний шаблон, що розділяє програму на три компоненти:
  - **Model (Модель):** Керує даними та бізнес-логікою.
  - **View (Вид):** Відповідає за відображення даних.
  - **Controller (Контролер):** Обробляє ввід користувача та координує взаємодію між моделлю та видом.
- **Як виглядає?** Controller приймає дані від користувача, оновлює Model, а потім View відображає оновлену Model.
- **Де використовують?** У розробці веб-додатків та десктопних програм з інтерфейсом користувача:
  - **Веб-фреймворки:** Django, Flask, Ruby on Rails побудовані на основі або схожих архітектурних шаблонах.

```
# --- Реалізація ---

# 1. Model (Модель): Містить дані та бізнес-логіку
class ToDoListModel:
    """Модель даних для списку завдань."""
    def __init__(self):
        self._tasks = []

    @property
    def tasks(self):
        return self._tasks

    def add_task(self, task: str):
        """Додає нове завдання до списку."""
        self._tasks.append({'name': task, 'completed': False})

    def complete_task(self, index: int):
        """Позначає завдання як виконане."""
        if 0 <= index < len(self._tasks):
            self._tasks[index]['completed'] = True

    def remove_task(self, index: int):
        """Видаляє завдання зі списку."""
        if 0 <= index < len(self._tasks):
            del self._tasks[index]

# 2. View (Вид): Відповідає за відображення даних
class ToDoListView:
    """Вид, який відображає список завдань у консолі."""
    def display_tasks(self, tasks: list):
```

```
        """Відображає всі завдання зі списку."""
        print("\n--- Список завдань ---")
        if not tasks:
            print("Список порожній.")
        else:
            for i, task in enumerate(tasks):
                status = "[✓]" if task['completed'] else "[ ]"
                print(f"{i+1}. {status} {task['name']}")
        print("-----------------------")

    def display_message(self, message: str):
        """Виводить інформаційне повідомлення."""
        print(f"Повідомлення: {message}")

# 3. Controller (Контролер): Керує взаємодією
class ToDoListController:
    """
    Контролер, який обробляє ввід користувача та оновлює
    модель і вид.
    """
    def __init__(self, model, view):
        self._model = model
        self._view = view

    def add_new_task(self, task_name: str):
        """Додає нове завдання через модель."""
        self._model.add_task(task_name)
        self._view.display_message(f"Додано завдання: '{task_name}'")
        self.update_view()

    def complete_existing_task(self, index: int):
        """Позначає завдання як виконане через модель."""
        self._model.complete_task(index - 1)
        self._view.display_message(f"Завдання #{index} позначено як виконане.")
        self.update_view()

    def remove_existing_task(self, index: int):
        """Видаляє завдання через модель."""
        self._model.remove_task(index - 1)
        self._view.display_message(f"Завдання #{index} видалено.")
        self.update_view()

    def update_view(self):
        """Змушує вид відобразити актуальні дані з моделі."""
```

```
        self._view.display_tasks(self._model.tasks)

# --- Демонстрація використання ---
print("\n--- Демонстрація: Запуск додатку To-Do ---")
model = ToDoListModel()
view = ToDoListView()
controller = ToDoListController(model, view)

# Початкове відображення
controller.update_view()

# Додавання завдань
controller.add_new_task("Купити хліб")
controller.add_new_task("Вивчити шаблони проектування")

# Позначення завдання як виконаного
controller.complete_existing_task(2)

# Видалення завдання
controller.remove_existing_task(1)
```

10. Composite (Компонувальник)

- **Що це?** Дозволяє об'єктам-композитам та окремим об'єктам оброблятися клієнтами однаковим чином. Це спрощує роботу з ієрархічними структурами, такими як дерева.
- **Як виглядає?** Є загальний інтерфейс Component, який реалізують Leaf (листок, окремий об'єкт) та Composite (композит, контейнер для інших компонентів).
- **Де використовують?** Коли потрібно працювати з ієрархічними даними:
  - **Файлова система:** Папки можуть містити як файли (листи), так і інші папки (композити).
  - **Графічні інтерфейси:** Панель може містити кнопки, текстові поля та інші панелі.

```
class Component(ABC):
    def __init__(self, name):
        self.name = name
    @abstractmethod
    def operation(self):
        pass
```

```python
# Листок
class File(Component):
    def operation(self):
        return f"Файл '{self.name}'"

# Композит
class Directory(Component):
    def __init__(self, name):
        super().__init__(name)
        self.children = []

    def add(self, component):
        self.children.append(component)

    def operation(self):
        results = [f"Директорія '{self.name}'"]
        for child in self.children:
            results.append(child.operation())
        return "\n".join(results)

# Приклад використання
root = Directory("root")
home = Directory("home")
user_dir = Directory("user_dir")
file1 = File("document.txt")
file2 = File("photo.jpg")

user_dir.add(file1)
user_dir.add(file2)
home.add(user_dir)
root.add(home)

print(root.operation())
```