1. Що це за проєкт?

Це невелика **система керування завданнями**, яка реалізована як консольна програма.

Вона дозволяє:

- створити задачу (назва, дата, категорія, пріоритет);
- переглянути всі задачі;
- завершити задачу;
- при завершенні — викликати **сповіщення** (наприклад, через консоль).

Програма реалізує **чисту архітектуру**, побудовану на **трьох шаблонах проєктування** (Builder, Observer, Facade) і дотримується принципів SOLID.

2. Які шаблони проєктування використано?

Патерн	Для чого?	Де використано?
Builder *(генеративний)*	Поетапне створення об'єкта Task	task_system/builder.py
Observer *(поведінковий)*	Реакція на завершення задачі — сповіщення	task_system/observer.py (інтерфейс + TaskNotifier), manager.py (notify)
Facade *(структурний)*	Надання одного простого API до всієї системи	task_system/facade.py — TaskSystem

🛠️ Builder

Файл: task_system/builder.py

```
"""Builder pattern — step-by-step Task creation."""
from __future__ import annotations
import datetime as _dt
from .models import Task

class TaskBuilder:
    """Fluent builder for Task objects."""
```

```
def __init__(self) -> None:
    self.reset()

def reset(self) -> None:
    self._title: str = ""
    self._due_date: _dt.date = _dt.date.today()
    self._category: str = ""
    self._priority: str = "medium"

def set_title(self, title: str) -> "TaskBuilder":
    self._title = title
    return self

def set_due_date(self, due_date: _dt.date) -> "TaskBuilder":
    self._due_date = due_date
    return self

def set_category(self, category: str) -> "TaskBuilder":
    self._category = category
    return self

def set_priority(self, priority: str) -> "TaskBuilder":
    self._priority = priority
    return self

def build(self) -> Task:
    """Return new Task and reset builder."""
    task = Task(self._title, self._due_date, self._category, self._priority)
    self.reset()
    return task
```

📣 Observer

Файли:

- task_system/observer.py — інтерфейс + реалізація TaskNotifier
- manager.py — реалізація Subject

```python
"""Observer abstraction + concrete notifier."""
from __future__ import annotations
from abc import ABC, abstractmethod
from .models import Task

class Observer(ABC):
    """Observer interface."""
    @abstractmethod
    def update(self, task: Task) -> None: ...

class TaskNotifier(Observer):
    """Console notifier (could be replaced by e-mail, webhook, etc.)."""
    def update(self, task: Task) -> None:
        print(f"[NOTIFICATION] Task '{task.title}' marked as completed.")
```

main.py

```python
class TaskManager(Subject):

    """Stores tasks & triggers notifications."""
    def __init__(self) -> None:
        super().__init__()
        self._tasks: Dict[str, Task] = {}

    # ---- CRUD ----
    def add_task(self, task: Task) -> None:
        self._tasks[task.title] = task
```

```
    def complete_task(self, title: str) -> None:
        if title in self._tasks:
            task = self._tasks[title]
            if not task.completed:
                task.complete()
                self.notify(task)

    # ---- Queries ----
    def all_tasks(self) -> List[Task]:
        return list(self._tasks.values())
```

🏛 Facade

Файл: task_system/facade.py

```
"""Facade exposing simple high-level API."""
from __future__ import annotations
from typing import List
import datetime as _dt
from .builder import TaskBuilder
from .manager import TaskManager
from .observer import TaskNotifier
from .models import Task

class TaskSystem:
    """High-level façade that hides internal complexity."""
    def __init__(self) -> None:
        self._builder = TaskBuilder()
        self._manager = TaskManager()
        self._manager.attach(TaskNotifier())

    # ---- API ----
    def create_task(self, title: str, due_date: _dt.date,
                    category: str, priority: str = "medium") -> None:
        task = (
            self._builder.set_title(title)
            .set_due_date(due_date)
            .set_category(category)
            .set_priority(priority)
            .build()
        )
        self._manager.add_task(task)

    def complete_task(self, title: str) -> None:
```

```
        self._manager.complete_task(title)

    def show_tasks(self) -> None:
        for t in self.tasks():
            status = "✅" if t.completed else "❌"
            print(f"{t.title:20} | {t.due_date} | {t.category:10} | "
                  f"{t.priority.upper():6} | {status}")

    # ---- Exposed read-only data ----
    def tasks(self) -> List[Task]:
        return self._manager.all_tasks()
```

main.py

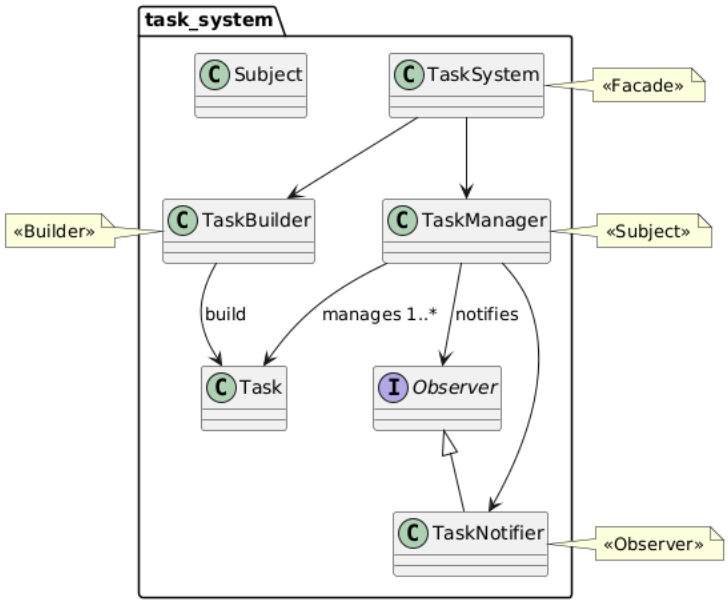
```
def main() -> None:
    ts = TaskSystem()

    ts.create_task("Завершити лабу", _dt.date(2025, 5, 30), "Навчання", "high")
    ts.create_task("Зробити бекап",   _dt.date(2025, 5, 28), "DevOps",    "medium")

    ts.show_tasks()          # до завершення
    ts.complete_task("Завершити лабу")
    ts.show_tasks()          # після завершення

if __name__ == "__main__":
    main()
```

Що перевіряється:

Тест-файл	Що перевіряє
test_builder.py	Створення задач через Builder, reset, нестандартні значення
test_manager.py	Додавання, завершення, повторне завершення, дублі
test_observer.py	Чи викликається update, чи працює сповіщення
test_facade.py	Створення через Facade, правильна взаємодія
test_full_flow.py	Створення → завершення → перевірка стану
test_integration.py	Робота з кількома задачами, стан після дій

UML-діаграма

Файл diagrams/uml_design.png



- **TaskSystem** — <>, з’єднаний із TaskBuilder та TaskManager.
- **TaskManager** — <>, який керує задачами та підписниками.
- **Observer** — інтерфейс для TaskNotifier, який виводить повідомлення.
- **Стрілки** показують залежності між класами.
- **Стереотипи** (<<Facade>>, <<Observer>>, <<Builder>>) показують роль класів у патернах.

1. Навіщо тут ABC і abstractmethod?

Ми використовуємо ABC (Abstract Base Class), щоб створити **інтерфейс Observer**, який:

- **гарантує**, що всі підкласи реалізують метод update(self, task) — це обов'язкова частина патерну Observer;
- дозволяє TaskManager працювати з **будь-яким Observer**, а не лише з TaskNotifier;
- дотримується **принципу DIP (Dependency Inversion Principle)**: залежність іде не від реалізації, а від абстракції.

Це робить наш код **розширюваним** і **тестованим**.

Наприклад, ми можемо створити EmailNotifier(Observer) — і нічого не змінювати в TaskManager.

Якщо не використовувати ABC, то хтось може випадково створити клас-нащадок без update(), і ми не помітимо цього до моменту виконання програми (runtime error замість compile-time перевірки).

2. Які принципи архітектури ми тут використали? Чим вони відрізняються від патернів?

Відповідь:

Патерни проектування (Design Patterns) — це **локальні рішення** для структурування частини коду (Builder, Observer, Facade).

Архітектурні принципи — це **вищий рівень**, який визначає структуру, залежності, модульність, масштабованість.

Принципи архітектури, які ми використали:

Принцип	Як реалізовано
Single Responsibility (SRP)	У кожного класу одна роль: Builder створює, Manager зберігає й керує, Observer сповіщає
Separation of Concerns (SoC)	Дані (Task) не змішуються з логікою (TaskManager), API винесене в окремий TaskSystem
Encapsulation	Усі деталі реалізації приховані за TaskSystem — клієнт не бачить, як усе побудовано всередині
Loose Coupling	Через Observer — Manager нічого не знає про сповіщення; Facade приховує залежності
Open/Closed Principle (OCP)	Новий Observer або Task тип — додається без змін у існуючий код
Dependency Inversion (DIP)	TaskManager залежить від абстрактного Observer, а не конкретного класу

Відмінність:

- Патерн = готовий шаблон вирішення часткової задачі (як створити, як повідомити, як приховати складність).
- Архітектурний принцип = правило побудови всієї системи (розділення відповідальності, слабке зв’язування, інкапсуляція).

3. Навіщо в кожному тесті self.assertEqual(...)?

Відповідь:

self.assertEqual(...) — це **перевірка результату** (assertion), яка підтверджує, що код працює правильно.

У кожному юніт-тесті ми створюємо об’єкт (наприклад, Task) і перевіряємо, що результат відповідає очікуванню.

Це — **серце будь-якого тесту**.