TP MODUL 13

Nama : Ghaza Zidane Nurraihan
Nim : 2311104038
Kelas : S1-SE07-01

Menjelaskan salah satu design pattern

A.)Berikan salah satu contoh kondisi dimana design pattern “Observer” dapat digunakan jawab:
contoh singkat penggunaan observer pattern: dalam aplikasi cuaca (weather app).
Dalam aplikasi cuaca, terdapat objek WeatherData yang bertindak sebagai publisher dan menyimpan data suhu, kelembapan, serta tekanan udara. Komponen UI seperti TemperatureDisplay, HumidityDisplay, dan PressureDisplay berperan sebagai subscriber yang ingin menerima notifikasi saat data cuaca berubah.

Ketika WeatherData menerima pembaruan dari API, ia akan memberi notifikasi ke semua observer yang terdaftar melalui metode notifyObservers(). Setiap observer kemudian akan memanggil metode update() untuk menampilkan data terbaru. Dengan pendekatan ini, relasi antar objek menjadi longgar (loosely coupled) dan dinamis, sesuai prinsip Open/Closed dalam OOP.

Contoh ini menggambarkan kondisi ideal untuk menerapkan Observer Pattern, yaitu saat banyak objek perlu merespons perubahan status satu objek pusat secara otomatis dan real-time tanpa menciptakan ketergantungan langsung antar objek.

B.) Berikan penjelasan singkat mengenai langkah-langkah dalam mengimplementasikan design pattern“Observer”
jawab:
a. Buat interface Observer
→ Berisi method update() untuk menerima notifikasi.

b. Buat interface Subject (Publisher)
→ Berisi method registerObserver(), removeObserver(), dan notifyObservers().

c. Implementasikan Subject konkret
→ Menyimpan daftar Observer dan memberi notifikasi saat ada perubahan.

d. Implementasikan Observer konkret
→ Menentukan respons saat menerima notifikasi.

e. Hubungkan Observer ke Subject
→ Daftarkan Observer menggunakan registerObserver().

f. Uji notifikasi
→ Saat Subject berubah, semua Observer akan dipanggil otomatis.

C.) Berikan kelebihan dan kekurangan dari design pattern “Observer”
Kelebihan:

- Rendah ketergantungan antara Subject dan Observer.
- Mudah dikembangkan (Open/Closed Principle).
- Update otomatis saat Subject berubah.
- Cocok untuk sistem notifikasi atau real-time.

Kekurangan:

- Sulit dilacak jika banyak Observer.
- Bisa menurunkan performa.
- Urutan notifikasi tidak terjamin.
- Risiko memory leak jika lupa unsubscribe.

IMPLEMENTASI DAN PEMAHAMAN DESIGN PATTERN OBSERVER

A. Pada project yang telah dibuat sebelumnya, tambahkan kode yang mirip atau sama dengan contoh kode yang diberikan di halaman web tersebut

```python
from __future__ import annotations
from abc import ABC, abstractmethod
from random import randrange
from typing import List


class Subject(ABC):
    """
    The Subject interface declares a set of methods for managing subscribers.
    """

    @abstractmethod
    def attach(self, observer: Observer) -> None:
        """
        Attach an observer to the subject.
        """
        pass

    @abstractmethod
    def detach(self, observer: Observer) -> None:
        """
        Detach an observer from the subject.
        """
        pass

    @abstractmethod
    def notify(self) -> None:
        """
        Notify all observers about an event.
        """
        pass


class ConcreteSubject(Subject):
    """
    The Subject owns some important state and notifies observers when the state
    changes.
    """

    _state: int = None
    """
    For the sake of simplicity, the Subject's state, essential to all
    subscribers, is stored in this variable.
    """

    _observers: List[Observer] = []
    """
    List of subscribers. In real life, the list of subscribers can be stored
    more comprehensively (categorized by event type, etc.).
    """

    def attach(self, observer: Observer) -> None:
        print("Subject: Attached an observer.")
        self._observers.append(observer)

    def detach(self, observer: Observer) -> None:
        self._observers.remove(observer)

    """
    The subscription management methods.
    """

    def notify(self) -> None:
        """
        Trigger an update in each subscriber.
        """

        print("Subject: Notifying observers...")
        for observer in self._observers:
            observer.update(self)

    def some_business_logic(self) -> None:
        """
        Usually, the subscription logic is only a fraction of what a Subject can
        really do. Subjects commonly hold some important business logic, that
        triggers a notification method whenever something important is about to
        happen (or after it).
        """

        print("\nSubject: I'm doing something important.")
        self._state = randrange(0, 10)

        print(f"Subject: My state has just changed to: {self._state}")
        self.notify()


class Observer(ABC):
    """
    The Observer interface declares the update method, used by subjects.
    """

    @abstractmethod
    def update(self, subject: Subject) -> None:
        """
        Receive update from subject.
        """
        pass


"""
Concrete Observers react to the updates issued by the Subject they had been
attached to.
"""


class ConcreteObserverA(Observer):
    def update(self, subject: Subject) -> None:
        if subject._state < 3:
            print("ConcreteObserverA: Reacted to the event")


class ConcreteObserverB(Observer):
    def update(self, subject: Subject) -> None:
        if subject._state == 0 or subject._state >= 2:
            print("ConcreteObserverB: Reacted to the event")


if __name__ == "__main__":

    subject = ConcreteSubject()

    observer_a = ConcreteObserverA()
    subject.attach(observer_a)

    observer_b = ConcreteObserverB()
    subject.attach(observer_b)

    subject.some_business_logic()
    subject.some_business_logic()

    subject.detach(observer_a)

    subject.some_business_logic()
```

B. Jalankan program tersebut dan pastikan tidak ada error pada saat project dijalankan

```
[Running] python -u "d:\Semester4\PRAKTIKUM KPL\KPL_Ghaza Zidane Nurraihan_2311104038_SE0701\13_Disign pattern\TP_Disign
pattern_2311104038\13_disignpattern.py"
Subject: Attached an observer.
Subject: Attached an observer.

Subject: I'm doing something important.
Subject: My state has just changed to: 7
Subject: Notifying observers...
ConcreteObserverB: Reacted to the event

Subject: I'm doing something important.
Subject: My state has just changed to: 7
Subject: Notifying observers...
ConcreteObserverB: Reacted to the event

Subject: I'm doing something important.
Subject: My state has just changed to: 3
Subject: Notifying observers...
ConcreteObserverB: Reacted to the event

[Done] exited with code=0 in 0.084 seconds
```

C. Jelaskan tiap baris kode yang terdapat di bagian method utama atau “main”

penjelasannya

```
if __name__ == "__main__":
    """
```

Ini adalah entry point dari program.
Artinya: Jalankan kode di bawah ini hanya jika file ini dijalankan secara langsung (bukan diimpor sebagai modul).

```
subject = ConcreteSubject()
```

Membuat objek ConcreteSubject, yaitu **subjek** yang diamati oleh observer

```
observer_a = ConcreteObserverA()
subject.attach(observer_a)
```

Menambahkan observer_a ke daftar observer dari subject.
Menambahkan observer_a ke daftar observer dari subject.

```
observer_b = ConcreteObserverB()
subject.attach(observer_b)
```

Membuat objek ConcreteObserverB, observer lain yang akan merespons jika state subject bernilai 0 atau >= 2.
Menambahkan observer_b ke daftar observer dari subject.

```
subject.some_business_logic()
subject.some_business_logic()
```

Memanggil method yang:
Mengubah nilai state secara acak (randrange(0, 10)).
Menampilkan nilai state baru.
Memberi notifikasi ke semua observer.

Menjalankan lagi logika bisnis (seperti di atas), memungkinkan observer merespons jika kondisi terpenuhi.

```
subject.detach(observer_a)
```

Menghapus observer_a dari daftar observer, sehingga tidak akan diberi notifikasi lagi.

```
subject.some_business_logic()
```

Menjalankan logika bisnis sekali lagi.
Hanya observer_b yang akan diberi notifikasi karena observer_a sudah dilepas.