

Nama : M. Faris

Nim : 2311104017

### TP Modul 13

#### A. Contoh Penggunaan:

Observer pattern cocok digunakan dalam aplikasi cuaca, di mana terdapat objek pusat (misalnya *WeatherStation*) yang memantau kondisi cuaca, dan berbagai objek lain seperti aplikasi mobile, layar digital, atau sistem peringatan cuaca (*subscribers*) ingin mendapatkan pembaruan setiap kali data cuaca berubah.

#### B. Langkah-langkah Implementasi:

1. Tentukan objek *publisher* (subjek yang diamati) yang menyediakan metode untuk *subscribe*, *unsubscribe*, dan *notify*.
2. Buat antarmuka (*interface*) untuk *subscriber* dengan metode *update()*.
3. Objek-objek yang ingin berlangganan perubahan akan mengimplementasikan antarmuka tersebut.
4. Saat terjadi perubahan pada *publisher*, ia akan memanggil *update()* pada semua *subscriber* yang terdaftar.

#### C. Kelebihan dan Kekurangan:

Kelebihan:

- Mendukung prinsip Open/Closed: objek baru dapat ditambahkan tanpa mengubah kode yang sudah ada.
- Relasi antar objek bisa dibentuk secara dinamis saat runtime.

Kekurangan:

- Urutan notifikasi tidak terjamin (acak).
- Jika terlalu banyak *subscriber*, performa bisa menurun karena banyaknya notifikasi yang harus dikirim.

## 1. Source Kode :

```
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__":
```

```
    # The client code.
```

```
    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()
```

## 2. Output :

```
PS C:\KPL_M.Faris_2311104017_SE-07-01> & C:/User  
_DesignPatternImplementation/Observer.py  
Subject: Attached an observer.  
Subject: Attached an observer.  
  
Subject: I'm doing something important.  
Subject: My state has just changed to: 8  
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: 4  
Subject: Notifying observers...  
ConcreteObserverB: Reacted to the event
```

## 3. Penjelasan :

Jadi, di main ada objek subject dan dua observer, lalu mendaftarkan observer ke subject, menjalankan aksi yang mengubah state subject dan mengirimkan update ke observer, kemudian melepas satu observer dan menjalankan aksi lagi untuk melihat bahwa hanya observer yang tersisa yang mendapatkan notifikasi. Ini menggambarkan mekanisme dinamis subscribe dan unsubscribe pada Observer pattern.