Observer Design Pattern

תבנית העיצוב Observer היא תבנית עיצוב מסוג התנהגות, שמאפשרת לאובייקטים (שנקרא **subject**) להודיע לאובייקטים אחרים (שנקראים **observers**) על שינויים של מצב מסויים.

לתבנית יש שני רכיבים עיקריים:

1. **Subject** - האובייקט שמנהל את הרשימה של ה-observers ומודיע להם על שינויים. הוא אחראי על רישום, מחיקה והודעה ל-observers. בדרך כלל נגדיר את הפונקציות notify, register, unregister שמאפשרות ל-observers להירשם ולהתרשם מה-subject.
2. **Observer** - האובייקט שמעוניין לקבל הודעות על שינויים ב-subject. כאשר הוא נרשם ל-subject, הוא מוסיף את עצמו לרשימת ה-observers של ה-subject ומגדיר פעולה שתבוצע כאשר הוא מקבל הודעה. בדרך כלל נגדיר אינטרפייס/מחלקה אבסטרקטית שמגדירה את הפעולה update שתבוצע כאשר ה-observer מקבל הודעה.

יתרונות

- התבנית מסירה את התלות בין ה-subject ל-observers, ומאפשרת להם להיות פחות תלויים זה בזה.
- שומרת על עקרון ה-Open/Closed, כאשר ניתן להוסיף observers חדשים ל-subject בקלות ובלי לשנות את ה-subject עצמו.
- גורמת לקוד להיות יותר נקי ומסודר, כאשר כל עניין של הודעה על שינוי מצב מופרד לפונקציות ומחלקות מיוחדות.
- גמישות - אפשר לתת לאובייקטים שונים טיפול שונה בהתראות.

דוגמא לשימוש ב- Observer

נניח שיש לנו מערכת חדשות שמפרסמת כתבות חדשות. נרצה שכאשר יש כתבה חדשה, כל המנויים על המערכת יקבלו הודעה על הכתבה החדשה.

אז כאן:

- **Subject** - המערכת חדשות. שומרת על רשימת המנויים ומודיעה להם על כתבה חדשה.
- **Observer** - המנויים על המערכת. כאשר יש כתבה חדשה, הם מקבלים הודעה על הכתבה החדשה.

java

```java
import java.util.ArrayList;
import java.util.List;

interface Subject {
    void registerObserver(Observer observer);
    void notifyObservers(String topic, String article);
}

interface Observer {
    void update(String topic, String article);
}

class NewsFeed implements Subject {
    private List<Observer> observers = new ArrayList<>();

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void notifyObservers(String topic, String article) {
        for (Observer observer : observers) {
            observer.update(topic, article);
        }
    }

    public void publishArticle(String topic, String article) {
```

```
        // Publish article logic...
        notifyObservers(topic, article);
    }
}

class User implements Observer {
    private String name;

    public User(String name) {
        this.name = name;
    }

    @Override
    public void update(String topic, String article) {
        System.out.println(
            name + " received notification: New article on " + topic + ": " + article
            );
    }
}

public class Main {
    public static void main(String[] args) {
        NewsFeed newsFeed = new NewsFeed();

        User user1 = new User("John");
        User user2 = new User("Mary");

        newsFeed.registerObserver(user1);
        newsFeed.registerObserver(user2);

        newsFeed.publishArticle("Technology", "AI advancements in healthcare");
        newsFeed.publishArticle("Sports", "NBA Finals highlights");
    }
}
```

python

הפעם ניתן דוגמה לשימוש כללי בתבנית העיצוב Observer בשפת Python. נגדיר מחלקה Subject שמנהלת את ה-observers שלה ומודיעה להם על שינויים, מחלקה Observer שמגדירה פעולה update שתבוצע כאשר הוא מקבל הודעה, ושני observers ספציפיים שמממשים את הפעולה update.

```
from abc import ABC, abstractmethod


class Subject:
    def __init__(self):
        self._observers = []

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

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

    def notify_all(self):
        for observer in self._observers:
            observer.update(self)


class Observer(ABC):
    @abstractmethod
    def update(self, subject):
        pass


class ConcreteObserverA(Observer):
    def update(self, subject):
        print("ConcreteObserverA: Reacted to the event")
```

```python
class ConcreteObserverB(Observer):
    def update(self, subject):
        print("ConcreteObserverB: Reacted to the event")


# Usage
subject = Subject()

observer_a = ConcreteObserverA()
observer_b = ConcreteObserverB()

subject.register(observer_a)
subject.register(observer_b)

subject.notify_all()  # Both observers will be notified

subject.unregister(observer_a)

subject.notify_all()  # Only observer_b will be notified
```