**Nama : Dimastian Aji Wibowo**

**NIM : 2311104058**

# TP MODUL 13

## A. Contoh kondisi penggunaan Observer

Observer design pattern sangat berguna ketika suatu objek perlu memberitahukan perubahan statusnya kepada objek lain tanpa harus mengetahui secara detail siapa yang menerima notifikasi tersebut. Sebagai contoh, pola Observer dapat digunakan pada aplikasi notifikasi email ketika ada update baru di suatu aplikasi. Ketika terjadi pembaruan konten di aplikasi, objek Subject akan mengirimkan notifikasi kepada semua Observer yang terdaftar, seperti pengguna yang telah berlangganan untuk menerima notifikasi tersebut. Dengan demikian, setiap perubahan pada Subject akan secara otomatis diinformasikan kepada seluruh Observer.

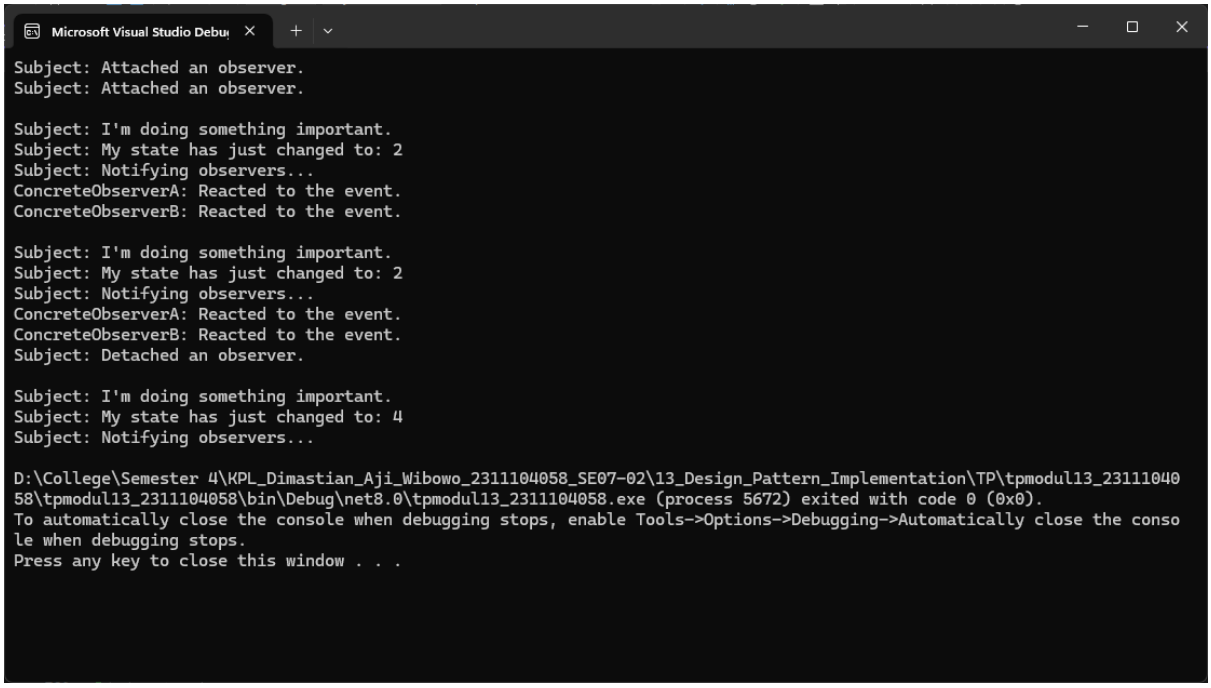
## B. Langkah-langkah implementasi Observer

Langkah pertama dalam mengimplementasikan Observer adalah menentukan objek Subject yang akan bertanggung jawab untuk mengirimkan notifikasi saat terjadi perubahan. Setelah itu, dibuat sebuah interface Observer yang memiliki metode Update() yang akan dipanggil saat notifikasi dikirimkan. Kemudian, setiap objek yang ingin menerima notifikasi harus mengimplementasikan interface Observer tersebut dan mendefinisikan metode Update() sesuai kebutuhan. Terakhir, Subject akan mengelola daftar Observer yang terdaftar untuk menerima notifikasi dan mengirimkan notifikasi tersebut ketika terjadi perubahan kondisi atau status pada Subject. Dengan cara ini, Observer akan selalu mendapatkan informasi terbaru secara otomatis tanpa harus melakukan pengecekan secara manual.

## C. Kelebihan dan kekurangan Observer

Observer design pattern memiliki beberapa kelebihan, salah satunya adalah dapat mengurangi ketergantungan antar objek, sehingga objek-objek tersebut dapat beroperasi secara independen. Selain itu, Observer mendukung arsitektur event-driven, yang memungkinkan objek untuk merespons suatu peristiwa atau perubahan status secara otomatis. Namun, pola ini juga memiliki kekurangan, yaitu jika tidak dikelola dengan baik, dapat terjadi memory leak akibat banyaknya Observer yang terdaftar namun tidak dihapus ketika tidak lagi dibutuhkan. Kondisi ini dapat memperlambat kinerja aplikasi, terutama jika jumlah Observer semakin bertambah seiring waktu.

## Hasil Run




## Source Code

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace RefactoringGuru.DesignPatterns.Observer.Conceptual
{
    public interface IObserver
    {
        // Receive update from subject
        void Update(ISubject subject);
    }

    public interface ISubject
    {
        // Attach an observer to the subject.
        void Attach(IObserver observer);

        // Detach an observer from the subject.
        void Detach(IObserver observer);

        // Notify all observers about an event.
        void Notify();
    }

    // The Subject owns some important state and notifies observers when the
    // state changes.
    public class Subject : ISubject
    {
        // For the sake of simplicity, the Subject's state, essential to all
        // subscribers, is stored in this variable.
        public int State { get; set; } = -0;

        // List of subscribers. In real life, the list of subscribers can be
        // stored more comprehensively (categorized by event type, etc.).
        private List<IObserver> _observers = new List<IObserver>();

        // The subscription management methods.
        public void Attach(IObserver observer)
        {
            Console.WriteLine("Subject: Attached an observer.");
```

```
            this._observers.Add(observer);
        }

        public void Detach(IObserver observer)
        {
            this._observers.Remove(observer);
            Console.WriteLine("Subject: Detached an observer.");
        }

        // Trigger an update in each subscriber.
        public void Notify()
        {
            Console.WriteLine("Subject: Notifying observers...");

            foreach (var observer in _observers)
            {
                observer.Update(this);
            }
        }

        // 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).
        public void SomeBusinessLogic()
        {
            Console.WriteLine("\nSubject: I'm doing something important.");
            this.State = new Random().Next(0, 10);

            Thread.Sleep(15);

            Console.WriteLine("Subject: My state has just changed to: " + this.State);
            this.Notify();
        }
    }

    // Concrete Observers react to the updates issued by the Subject they had
    // been attached to.
    class ConcreteObserverA : IObserver
    {
        public void Update(ISubject subject)
        {
            if ((subject as Subject).State < 3)
            {
                Console.WriteLine("ConcreteObserverA: Reacted to the event.");
            }
        }
    }

    class ConcreteObserverB : IObserver
    {
        public void Update(ISubject subject)
        {
            if ((subject as Subject).State == 0 || (subject as Subject).State >= 2)
            {
                Console.WriteLine("ConcreteObserverB: Reacted to the event.");
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // The client code.
            var subject = new Subject();
            var observerA = new ConcreteObserverA();
            subject.Attach(observerA);

            var observerB = new ConcreteObserverB();
            subject.Attach(observerB);

            subject.SomeBusinessLogic();
            subject.SomeBusinessLogic();

            subject.Detach(observerB);
```

```
            subject.SomeBusinessLogic();
        }
    }
}
```

Kode ini menerapkan Desain Pattern Observer, yang memungkinkan objek (Subject) untuk menginformasikan perubahan statusnya kepada objek lain (Observer) yang terdaftar tanpa mengikatkan diri pada implementasi detail dari objek tersebut.

1. IObserver adalah antarmuka yang mendefinisikan metode Update(), yang digunakan oleh objek pengamat untuk menerima pemberitahuan ketika status pada Subject berubah.
2. ISubject adalah antarmuka untuk objek yang akan dipantau oleh Observer. Antarmuka ini memiliki metode Attach(), Detach(), dan Notify(), yang mengelola daftar Observer dan memberi tahu mereka jika ada perubahan.
3. Subject adalah kelas yang mengimplementasikan antarmuka ISubject. Kelas ini memiliki status State yang menyimpan informasi yang penting untuk para Observer. Metode Attach() dan Detach() digunakan untuk menambah atau menghapus Observer dari daftar. Metode Notify() digunakan untuk memberi tahu semua Observer yang terdaftar tentang perubahan status.
4. ConcreteObserverA dan ConcreteObserverB adalah implementasi dari antarmuka IObserver yang masing-masing merespons pembaruan berdasarkan kondisi status tertentu. ConcreteObserverA merespons perubahan ketika status kurang dari 3, sementara ConcreteObserverB merespons ketika status bernilai 0 atau lebih dari atau sama dengan 2.
5. Dalam kelas Program, objek Subject dibuat, dan dua Observer (ConcreteObserverA dan ConcreteObserverB) ditambahkan ke dalamnya. Ketika metode SomeBusinessLogic() dipanggil pada objek Subject, statusnya berubah secara acak, dan kedua Observer menerima pemberitahuan tentang perubahan tersebut melalui metode Notify(). Kemudian, salah satu Observer dihapus, dan pembaruan berikutnya hanya diterima oleh yang tersisa.

Desain ini berguna ketika ada banyak objek yang perlu diberi tahu tentang perubahan status objek lainnya tanpa perlu mengetahui atau mengontrol bagaimana objek tersebut diimplementasikan.