Muhammad Ikhsan Al Hakim

2311104064

Laporan Tugas Pendahuluan Modul 13

1. Program.cs

```
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();
        }
    }
}
```

Penjelasan:

Class Program diatas merupakan implementasi dari Design Pattern Observer dalam C#, di mana Subject menyimpan state internal yang dapat berubah dan memiliki daftar Observer yang akan diberi notifikasi setiap kali state tersebut berubah. Interface ISubject mendefinisikan operasi untuk menambahkan, menghapus, dan memberi notifikasi kepada observer, sedangkan Observer menentukan metode Update yang akan dipanggil saat notifikasi terjadi. Kelas Subject menjalankan logika bisnis melalui metode SomeBusinessLogic yang mengubah state-nya secara acak, lalu memanggil NotifyO untuk memperbarui semua observer yang terdaftar. Dua observer konkret (ConcreteObserverA dan ConcreteObserverB) memberikan respons berdasarkan kondisi nilai State. Dalam metode Main, observer ditambahkan dan dihapus secara dinamis, menunjukkan bagaimana hubungan pengamat-diamati berjalan dalam praktik

2. Hasil Run