# TUGAS PENDAHULUAN
# KONSTRUKSI PERANGKAT LUNAK

## PERTEMUAN 10

**Disusun Oleh :**

**Muhammad Abdul Aziz**

**2211104026**

**SE0601**

**Asisten Praktikum :**

**Naufal El Kamil Aditya Pratama Rahman**

**Imelda**

**Dosen Pengampu :**

**Yudha Islami Sulistya, S.Kom., M.Cs.**

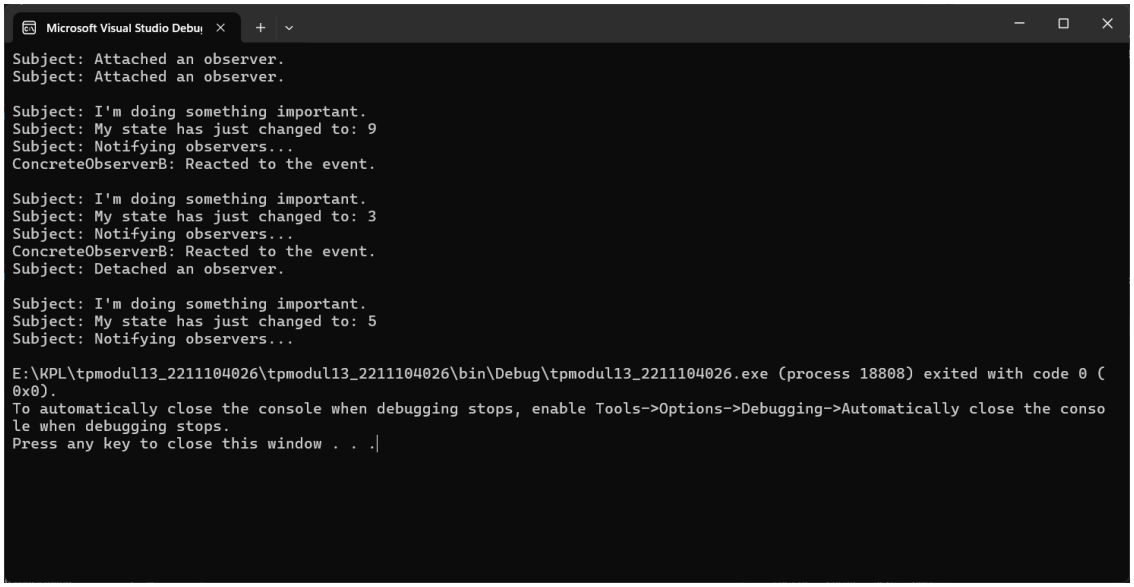
**PROGRAM STUDI S1 SOFTWARE ENGINEERING**

**FAKULTAS INFORMATIKA**

**TELKOM UNIVERSITY PURWOKERTO**

**2025**

1. **Screenshot hasil run**




2. **Penjelasan singkat dari kode implementasi yang dibuat (beserta screenshot dari potongan source code yang dijelaskan).**

   Design Pattern

   **A. Contoh kondisi penggunaan Observer Pattern:**

   Observer pattern dapat digunakan pada **sistem notifikasi**, misalnya:
   - Aplikasi berita yang memberitahu pengguna saat ada berita baru.
   - Sistem monitoring cuaca yang mengupdate suhu di berbagai lokasi.

   **B. Langkah-langkah implementasi Observer Pattern:**
   1. Buat **Subject** (publisher) yang menyimpan list dari **Observers** (subscriber).
   2. Subject menyediakan method untuk:
      - Menambahkan observer.
      - Menghapus observer.
      - Memberi notifikasi ke semua observer saat ada perubahan.
   3. Buat interface IObserver untuk observer.
   4. Implementasikan class-class observer yang menerapkan IObserver.
   5. Subject akan memanggil Update() milik observer saat ada perubahan data.

   **C. Kelebihan dan Kekurangan:**

   **Kelebihan:**
   - Mengikuti prinsip **Loose Coupling** (tidak bergantung langsung).
   - **Mudah diperluas**, tinggal tambah observer baru.
   - Cocok untuk **event-driven systems**.

   **Kekurangan:**
   - Bisa sulit dilacak jika terlalu banyak observer.
   - Risiko **memory leaks** jika observer tidak dihapus dengan benar.

Source Code :

Program.cs

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

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

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

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

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

    // The Subject owns some important state and notifies observers when the
    // state changes.
    4 references
    public class Subject : ISubject
    {
        // For the sake of simplicity, the Subject's state, essential to all
        // subscribers, is stored in this variable.
        5 references
        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.
        3 references
        public void Attach(IObserver observer)
        {
            Console.WriteLine("Subject: Attached an observer.");
            this._observers.Add(observer);
        }

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

        // Trigger an update in each subscriber.
        2 references
        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).
        3 references
        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.
    1 reference
    class ConcreteObserverA : IObserver
    {
        2 references
        public void Update(ISubject subject)
        {
            if ((subject as Subject).State < 3)
            {
                Console.WriteLine("ConcreteObserverA: Reacted to the event.");
```

```
            }
        }
    }

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

    0 references
    class Program
    {
        0 references
        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

- Isubject (Publisher Interface)

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

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

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

- Ini adalah antarmuka dari publisher (si pemberi notifikasi).
- Menyediakan method untuk menambahkan, menghapus, dan memberi notifikasi ke observer.

- Subject (Publisher)

```
public class Subject : ISubject
{
    // For the sake of simplicity, the Subject's state, essential to all
    // subscribers, is stored in this variable.
    5 references
    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.
    3 references
    public void Attach(IObserver observer)
    {
        Console.WriteLine("Subject: Attached an observer.");
        this._observers.Add(observer);
    }

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

    // Trigger an update in each subscriber.
    2 references
    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).
    3 references
    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();
    }
}
```

- Menyimpan data penting (State).
- Menyimpan daftar observer dalam list.
- Saat terjadi perubahan state, akan memanggil Notify() ke semua observer.

- IObserver (Observer Interface)

```
public interface IObserver
{
    // Receive update from subject
    3 references
    void Update(ISubject subject);
}
```

- Setiap observer harus mengimplementasikan method Update().

- ConcreteObserverA dan ConcreteObserverB

```
class ConcreteObserverA : IObserver
{
    2 references
    public void Update(ISubject subject)
    {
        if ((subject as Subject).State < 3)
        {
            Console.WriteLine("ConcreteObserverA: Reacted to the event.");
        }
    }
}

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

  - Mereka adalah implementasi nyata dari observer.
  - Mereka akan melakukan sesuatu berdasarkan kondisi dari subject.State.
- Program.Main() (Client)

```
class Program
{
    0 references
    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();
    }
}
```

  - Di sinilah semuanya dijalankan.
  - Program membuat objek subject dan menambahkan observer ke dalamnya.
  - SomeBusinessLogic() mengubah State lalu memicu Notify() ke observer.