TUGAS PENDAHULUAN
KONSTRUKSI PERANGKAT LUNAK

MODUL XIII

DESIGN PATTERN IMPLEMENTATION

Disusun Oleh :

Atika Aji Hadiyani

2211104003

SE-06-01

Dosen Pengampu :

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

PROGRAM STUDI S1 SOFTWARE ENGINEERING

TELKOM UNIVERSITY PURWOKERTO

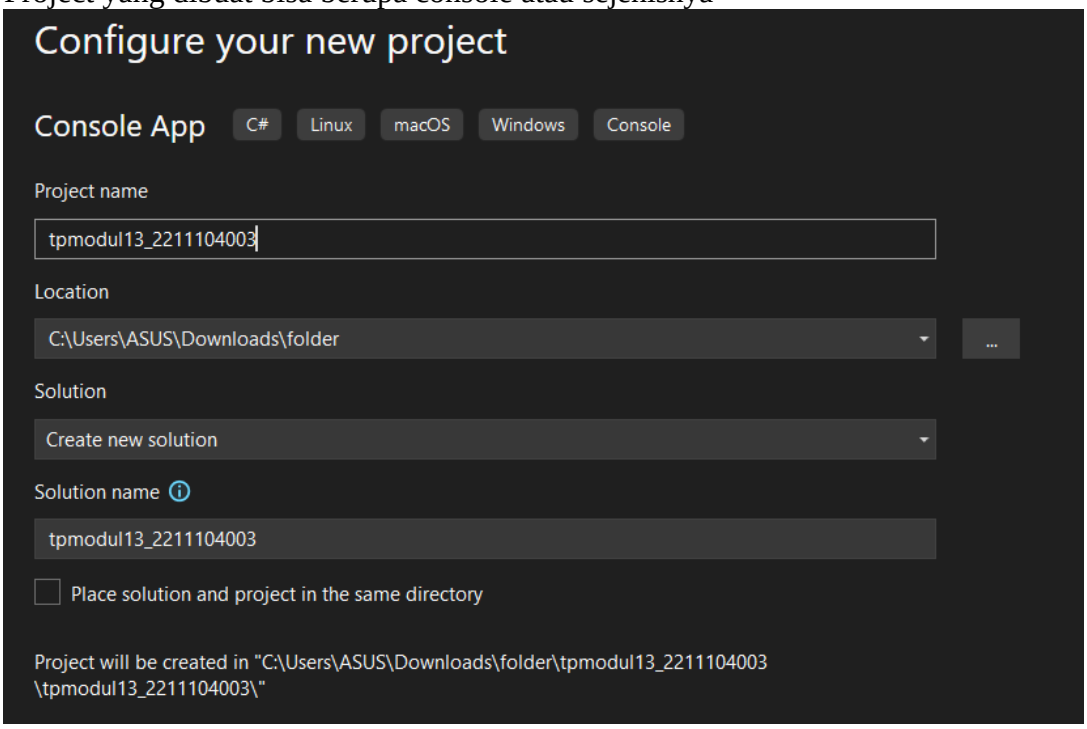
TUGAS PENDAHULUAN 13

1. MEMBUAT PROJECT GUI BARU

Buka IDE misalnya dengan Visual Studio

A. Misalnya menggunakan Visual Studio, buatlah project baru dengan nama tpmodul113_NIM

B. Project yang dibuat bisa berupa console atau sejenisnya



2. MENJELASKAN SALAH SATU DESIGN PATTERN

Buka halaman web https://refactoring.guru/design-patterns/catalog kemudian baca design pattern dengan nama “Observer”, dan jawab pertanyaan berikut ini (dalam Bahasa Indonesia):

A. Berikan salah satu contoh kondisi dimana design pattern “Observer” dapat digunakan?

Jawab:

Observer pattern dapat digunakan pada aplikasi berita atau cuaca, di mana terdapat satu objek utama (misalnya, pusat data cuaca) yang memberitahu banyak tampilan atau komponen (observer) saat terjadi perubahan data. Misalnya, ketika suhu berubah, semua tampilan suhu di aplikasi akan diperbarui secara otomatis tanpa harus saling mengetahui satu sama lain.

B. Berikan penjelasan singkat mengenai langkah-langkah dalam mengimplementasikan design pattern “Observer” .

Jawab:

- **Buat interface Observer** yang memiliki metode seperti update() untuk menerima notifikasi.
- **Buat interface Subject (Publisher)** yang memiliki metode untuk attach(), detach(), dan notifyObservers().
- **Implementasikan Subject** dalam kelas konkret yang menyimpan data dan daftar observer.
- **Implementasikan Observer** dalam kelas yang ingin mendapatkan notifikasi.
- Saat data di Subject berubah, panggil notifyObservers() untuk memberitahu semua Observer.

C. Berikan kelebihan dan kekurangan dari design pattern “Observer”

Jawab:

Kelebihan:

- **Loose coupling**: Subject dan Observer tidak saling tergantung secara langsung, sehingga memudahkan pemeliharaan dan pengembangan.
- **Konsistensi data**: Semua observer akan otomatis mendapatkan informasi terbaru saat terjadi perubahan pada subject.
- **Fleksibel**: Mudah menambah atau menghapus observer tanpa mengubah kode subject.

Kekurangan:

- **Dapat menimbulkan kompleksitas** saat terlalu banyak observer, sehingga sulit dilacak.
- **Masalah performa**: Jika banyak observer dan pembaruan sering terjadi, dapat menurunkan performa sistem.
- **Notifikasi tidak terkendali**: Jika tidak dikelola dengan baik, bisa terjadi notifikasi berlebihan atau loop pemanggilan.

3. IMPLEMENTASI DAN PEMAHAMAN DESIGN PATTERN OBSERVER

Buka halaman web berikut https://refactoring.guru/design-patterns/observer dan scroll ke bagian “Code Examples”, pilih kode yang akan dilihat misalnya C# dan ikuti langkah-langkah berikut:

A. Pada project yang telah dibuat sebelumnya, tambahkan kode yang mirip atau sama dengan contoh kode yang diberikan di halaman web tersebut
B. Jalankan program tersebut dan pastikan tidak ada error pada saat project dijalankan
C. Jelaskan tiap baris kode yang terdapat di bagian method utama atau “main”

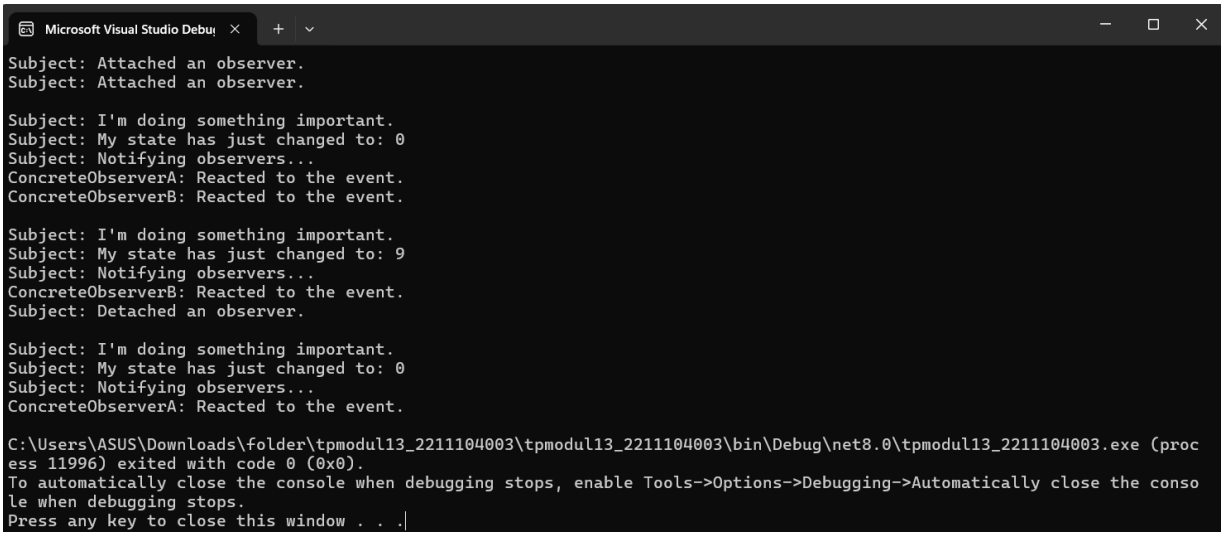
Source Code:

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

namespace ObserverPatternExample
{
    // Interface Observer
    8 references
    public interface IObserver
    {
        3 references
        void Update(ISubject subject);
    }

    // Interface Subject (Publisher)
    4 references
    public interface ISubject
    {
        3 references
        void Attach(IObserver observer);
        2 references
        void Detach(IObserver observer);
        2 references
        void Notify();
    }

    // Concrete Subject
    4 references
    public class Subject : ISubject
    {
        5 references
        public int State { get; set; } = 0;

        private List<IObserver> _observers = new List<IObserver>();

        3 references
        public void Attach(IObserver observer)
        {
            Console.WriteLine("Subject: Attached an observer.");
            _observers.Add(observer);
        }

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

```
        2 references
        public void Notify()
        {
            Console.WriteLine("Subject: Notifying observers...");
            foreach (var observer in _observers)
            {
                observer.Update(this);
            }
        }

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

    // Concrete Observer B
    1 reference
    public 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.");
            }
        }
    }
```

```
    // Main Program
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            var subject = new Subject();

            var observerA = new ConcreteObserverA();
            subject.Attach(observerA);

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

            subject.SomeBusinessLogic();  // Observer A dan B merespon
            subject.SomeBusinessLogic();  // Observer A dan B merespon

            subject.Detach(observerB);    // Observer B tidak lagi merespon

            subject.SomeBusinessLogic();  // Hanya Observer A yang merespon
        }
    }
}
```

Hasil:



Penjelasan bagian method utama (main):

- var subject = new Subject();
  Baris ini membuat instance dari Subject, yaitu objek pusat (publisher) yang menyimpan sebuah *state* dan bertugas memberi tahu para *observer* saat terjadi perubahan nilai.

- var observerA = new ConcreteObserverA();
  subject.Attach(observerA);
  a. Baris pertama mendefinisikan observerA, yaitu entitas yang ingin memantau setiap pembaruan pada subject.
  b. Baris kedua menambahkan observerA ke dalam daftar observer milik subject, sehingga ia akan menerima notifikasi jika ada perubahan pada state subject.

- var observerB = new ConcreteObserverB();
  subject.Attach(observerB);
  a. Di sini dibuat observerB, yaitu observer lain yang juga ingin memperoleh informasi perubahan dari subject.
  b. Selanjutnya, observerB didaftarkan ke subject agar juga mendapatkan pemberitahuan saat state subject berubah.

- subject.SomeBusinessLogic();
  a. Method ini memicu logika bisnis yang terdapat di dalam subject.
  b. Dalam prosesnya, subject akan secara acak memodifikasi state-nya, lalu memberi tahu seluruh observer yang terdaftar—dalam hal ini observerA dan observerB.

- subject.Detach(observerB);
  Baris ini digunakan untuk mencabut observerB dari daftar observer milik subject. Setelah ini, observerB tidak akan menerima pemberitahuan lagi setiap kali state subject berubah.