Marvel Sanjaya Setiawan

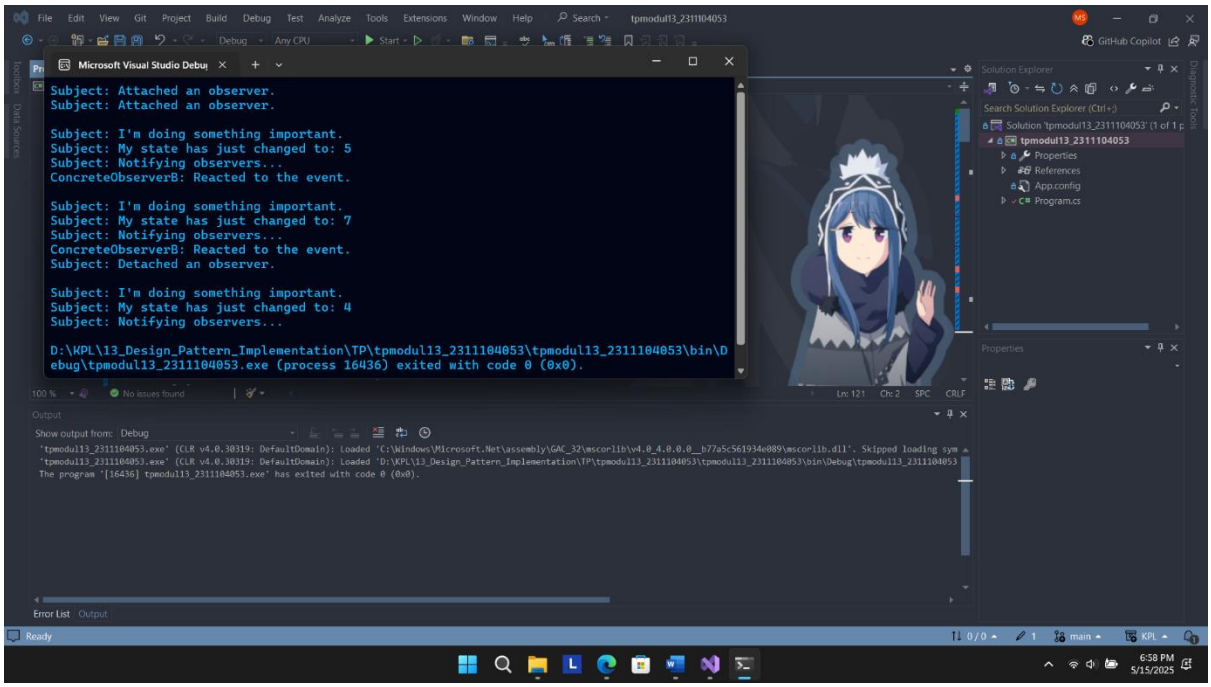
2311104053

SE07-02

TP MODUL 13

Link github:
https://github.com/Meph1sto14/KPL_Marvel_Sanjaya_Setiawan_2311104053_SE07-02/tree/b89577c370d82e2a74c09ec51ad431470578c6f9/13_Design_Pattern_Implementation/TP/tpmodul13_2311104053

Hasil run:



Program.cs

```
1. using System;
2. using System.Collections.Generic;
3. using System.Threading;
4.
5. namespace RefactoringGuru.DesignPatterns.Observer.Conceptual
6. {
7.     public interface IObserver
8.     {
9.         // Receive update from subject
10.        void Update(ISubject subject);
11.    }
12.
13.    public interface ISubject
14.    {
15.        // Attach an observer to the subject.
16.        void Attach(IObserver observer);
17.
18.        // Detach an observer from the subject.
19.        void Detach(IObserver observer);
```

```
20.
21.         // Notify all observers about an event.
22.         void Notify();
23.     }
24.
25.     // The Subject owns some important state and notifies observers when the
26.     // state changes.
27.     public class Subject : ISubject
28.     {
29.         // For the sake of simplicity, the Subject's state, essential to all
30.         // subscribers, is stored in this variable.
31.         public int State { get; set; } = -0;
32.
33.         // List of subscribers. In real life, the list of subscribers can be
34.         // stored more comprehensively (categorized by event type, etc.).
35.         private List<IObserver> _observers = new List<IObserver>();
36.
37.         // The subscription management methods.
38.         public void Attach(IObserver observer)
39.         {
40.             Console.WriteLine("Subject: Attached an observer.");
41.             this._observers.Add(observer);
42.         }
43.
44.         public void Detach(IObserver observer)
45.         {
46.             this._observers.Remove(observer);
47.             Console.WriteLine("Subject: Detached an observer.");
48.         }
49.
50.         // Trigger an update in each subscriber.
51.         public void Notify()
52.         {
53.             Console.WriteLine("Subject: Notifying observers...");
54.
55.             foreach (var observer in _observers)
56.             {
57.                 observer.Update(this);
58.             }
59.         }
60.
61.         // Usually, the subscription logic is only a fraction of what a Subject
62.         // can really do. Subjects commonly hold some important business logic,
63.         // that triggers a notification method whenever something important is
64.         // about to happen (or after it).
65.         public void SomeBusinessLogic()
66.         {
67.             Console.WriteLine("\nSubject: I'm doing something important.");
68.             this.State = new Random().Next(0, 10);
69.
70.             Thread.Sleep(15);
71.
72.             Console.WriteLine("Subject: My state has just changed to: " + this.State);
73.             this.Notify();
74.         }
75.     }
76.
77.     // Concrete Observers react to the updates issued by the Subject they had
78.     // been attached to.
79.     class ConcreteObserverA : IObserver
80.     {
81.         public void Update(ISubject subject)
82.         {
83.             if ((subject as Subject).State < 3)
84.             {
85.                 Console.WriteLine("ConcreteObserverA: Reacted to the event.");
86.             }
87.         }
88.     }
```

```
89.
90.     class ConcreteObserverB : IObserver
91.     {
92.         public void Update(ISubject subject)
93.         {
94.             if ((subject as Subject).State == 0 || (subject as Subject).State >= 2)
95.             {
96.                 Console.WriteLine("ConcreteObserverB: Reacted to the event.");
97.             }
98.         }
99.     }
100.
101.    class Program
102.    {
103.        static void Main(string[] args)
104.        {
105.            // The client code.
106.            var subject = new Subject();
107.            var observerA = new ConcreteObserverA();
108.            subject.Attach(observerA);
109.
110.            var observerB = new ConcreteObserverB();
111.            subject.Attach(observerB);
112.
113.            subject.SomeBusinessLogic();
114.            subject.SomeBusinessLogic();
115.
116.            subject.Detach(observerB);
117.
118.            subject.SomeBusinessLogic();
119.        }
120.    }
121. }
122.
```

MENJELASKAN SALAH SATU DESIGN PATTERN

A. Berikan salah satu contoh kondisi dimana design pattern “Observer” dapat digunakan
B. Berikan penjelasan singkat mengenai langkah-langkah dalam mengimplementasikan design pattern “Observer”
C. Berikan kelebihan dan kekurangan dari design pattern “Observer”

Jawaban:

A. Berikut dibawah ini adalah salah satu contoh kondisi
   Bayangkan Anda memiliki dua jenis objek: Pelanggan dan Toko. Pelanggan sangat tertarik dengan merek produk tertentu (misalnya, model baru iPhone) yang akan segera tersedia di toko.

   Pelanggan dapat mengunjungi toko setiap hari dan memeriksa ketersediaan produk. Namun, saat produk masih dalam perjalanan, sebagian besar perjalanan ini tidak ada gunanya.

   Di sisi lain, toko dapat mengirim banyak email (yang mungkin dianggap spam) ke semua pelanggan setiap kali produk baru tersedia. Hal ini akan menyelamatkan beberapa pelanggan dari perjalanan yang tidak ada habisnya ke toko. Pada saat yang sama, hal ini akan mengganggu pelanggan lain yang tidak tertarik dengan produk baru.

Sepertinya kita mengalami konflik. Entah pelanggan membuang waktu untuk memeriksa ketersediaan produk atau toko membuang-buang sumber daya untuk memberi tahu pelanggan yang salah.

B. Penjelasan singkat mengenai langkah-langkah implementasi Observer Pattern:
   1. Pisahkan logika bisnis → Identifikasi bagian yang bertindak sebagai publisher dan bagian yang akan menjadi subscriber.

   2. Buat antarmuka pelanggan (Observer) → Mendeklarasikan metode Update() untuk menerima notifikasi dari publisher.

   3. Buat antarmuka penerbit (Subject) → Mendeklarasikan metode untuk menambahkan, menghapus, dan memberi tahu subscriber.

   4. Kelola daftar langganan → Menyimpan daftar subscriber dan menerapkan metode langganan. Bisa diwarisi oleh penerbit konkret atau dikelola secara terpisah.

   5. Buat penerbit konkret → Mengimplementasikan logika yang mengubah state dan memanggil Notify() saat ada perubahan.

   6. Implementasikan notifikasi pelanggan → Subscriber menerima update dan bereaksi terhadap perubahan state penerbit.

   7. Daftarkan pelanggan ke penerbit → Client code membuat pelanggan, menambahkannya ke penerbit, dan memantau bagaimana mereka merespons perubahan.

C. Kelebihan:
   1. Prinsip Terbuka/Tertutup. Anda dapat memperkenalkan kelas pelanggan baru tanpa harus mengubah kode penerbit (dan sebaliknya jika ada antarmuka penerbit).

   2. Anda dapat membuat relasi antar objek pada saat runtime.

   Kekurangan:
   1. Pelanggan diberitahukan secara acak.

Penjelasan tiap baris kode yang terdapat di bagian method utama atau “main”

1. var subject = new Subject();
   - Membuat instance dari Subject, yang berfungsi sebagai publisher atau objek yang diamati.

2. var observerA = new ConcreteObserverA();
   - Membuat instance dari ConcreteObserverA, yaitu salah satu observer yang akan bereaksi terhadap perubahan state Subject.

3. subject.Attach(observerA);
    - Menambahkan observerA ke daftar subscriber Subject, sehingga akan menerima notifikasi saat state berubah.

4. var observerB = new ConcreteObserverB();
    - Membuat instance dari ConcreteObserverB, observer lain yang memiliki logika reaksi berbeda dari ConcreteObserverA.

5. subject.Attach(observerB);
    - Menambahkan observerB ke daftar subscriber Subject.

6. subject.SomeBusinessLogic();
    - Memanggil metode yang mengubah state Subject secara acak dan memberi tahu semua observer tentang perubahan tersebut.

7. subject.SomeBusinessLogic();
    - Memanggil metode yang sama sekali lagi untuk memicu perubahan state lain dan memberitahu observer.

8. subject.Detach(observerB);
    - Menghapus observerB dari daftar subscriber, sehingga tidak akan menerima update lagi.

9. subject.SomeBusinessLogic();
    - Memanggil metode bisnis lagi untuk melihat bagaimana observer yang tersisa (observerA) bereaksi tanpa observerB.