TUGAS PENDAHULUAN
KONSTRUKSI PERANGKAT LUNAK

PERTEMUAN 13

Design Pattern Implementation

Disusun Oleh :

Andera Singgih Pratama

2211104007

SE0601

Asisten Praktikum :

Naufal El Kamil Aditya Pratama Rahman

Imelda

Dosen Pengampu :

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

PROGRAM STUDI S1 SOFTWARE ENGINEERING

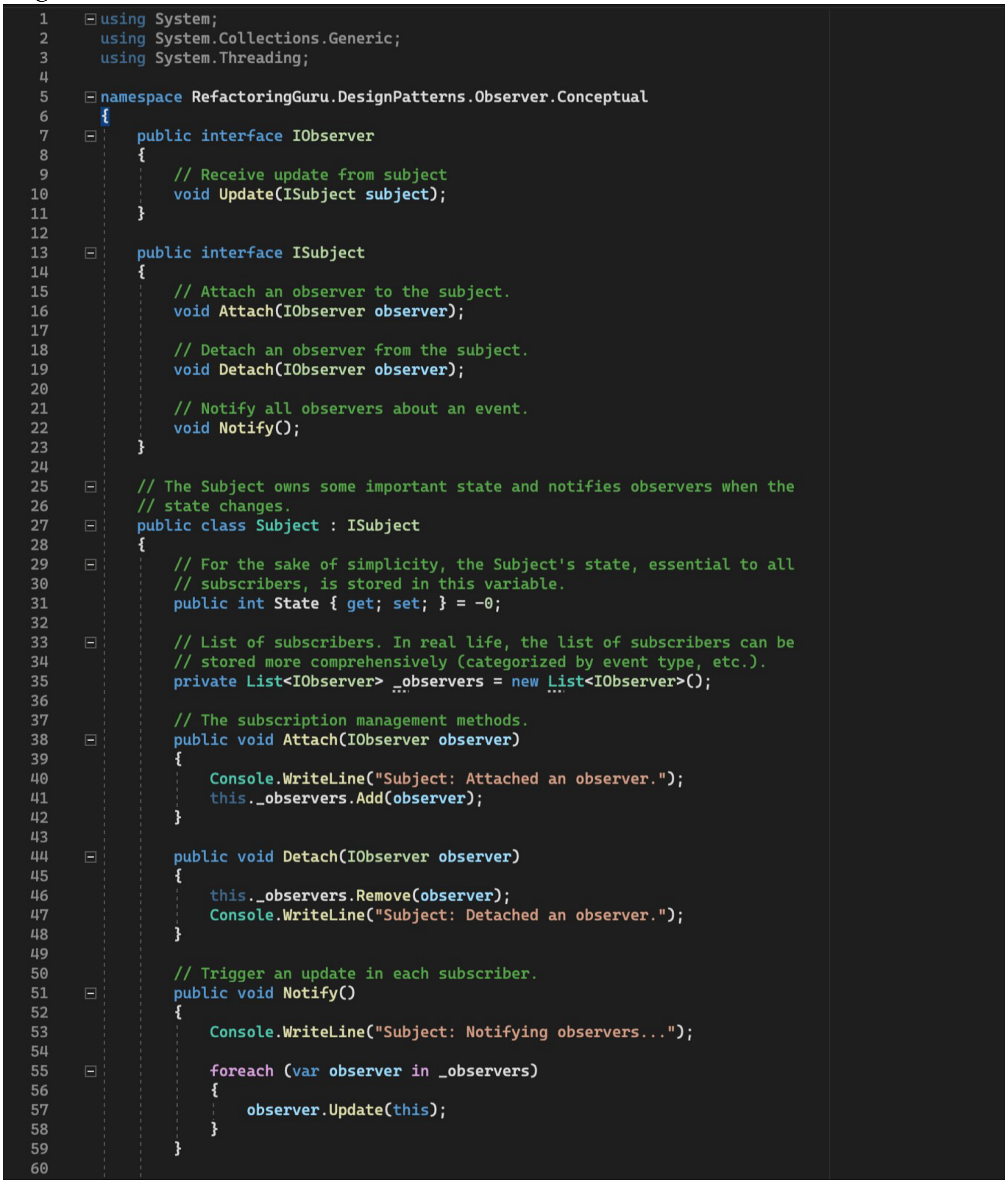
FAKULTAS INFORMATIKA

TELKOM UNIVERSITY PURWOKERTO

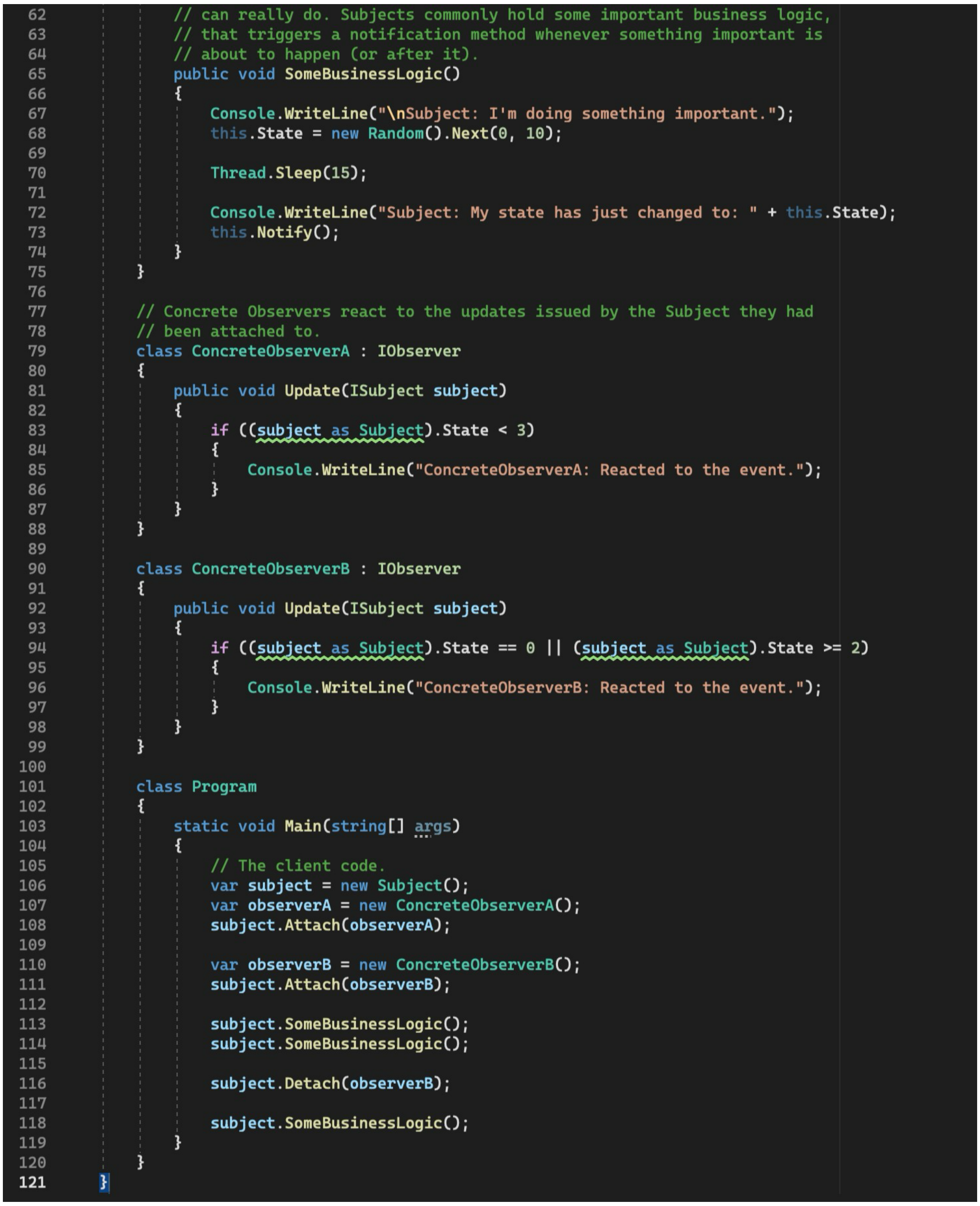
2025

Source Code

Program.cs

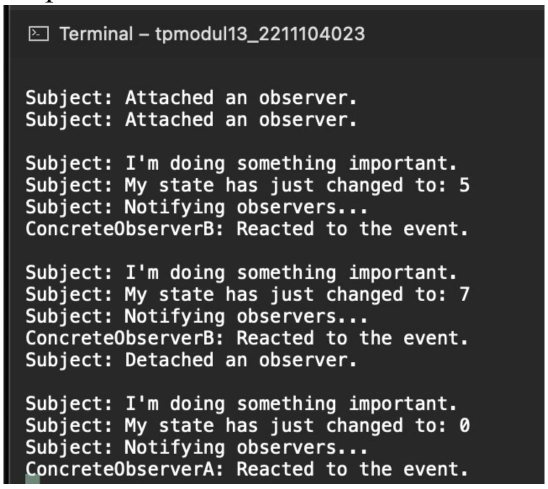


```
using System;
using System.Collections.Generic;
using System.Threading;

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);
            }
        }

```



```csharp
        // 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();
        }
    }
}
```

Output



```
Subject: Attached an observer.
Subject: Attached an observer.

Subject: I'm doing something important.
Subject: My state has just changed to: 5
Subject: Notifying observers...
ConcreteObserverB: Reacted to the event.

Subject: I'm doing something important.
Subject: My state has just changed to: 7
Subject: Notifying observers...
ConcreteObserverB: Reacted to the event.
Subject: Detached an observer.

Subject: I'm doing something important.
Subject: My state has just changed to: 0
Subject: Notifying observers...
ConcreteObserverA: Reacted to the event.
```

Penjelasan Program

Program ini adalah **implementasi dari *Observer Design Pattern*** dalam bahasa **C#**, yang merupakan bagian dari pola desain perilaku (behavioral design pattern). Pola ini memungkinkan suatu **objek (Subject)** untuk memberi tahu **objek-objek lain (Observers)** saat terjadi perubahan **status** tanpa perlu mengetahui siapa observer tersebut. Tujuannya Untuk **memisahkan hubungan satu-ke-banyak** antara objek, sehingga ketika **satu objek berubah**, maka semua objek lain yang "mengamati" (observer) akan **diberi tahu secara otomatis**.

Struktur Program Interface

```
public interface IObserver
{
    void Update(ISubject subject);
}
public interface ISubject
{     void Attach(IObserver
observer);     void Detach(IObserver
observer);     void Notify();
}
```

- `IObserver`: Interface yang wajib diimplementasikan oleh semua observer. Mereka akan menerima pembaruan dari `ISubject`.
- `ISubject`: Interface yang wajib diimplementasikan oleh objek yang diamati. Memiliki metode untuk:
    - o `Attach`: Menambahkan observer o `Detach`:

Menghapus observer o `Notify`: Memberi tahu semua

observer **Subject (** yang diamati **)** `public class`

```
Subject : ISubject

{     public int State { get; set; } = -0;     private

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

     public void Attach(IObserver observer) { ...

}     public void Detach(IObserver observer) { ...

}     public void Notify() { ... }

     public void

SomeBusinessLogic()

    {

        // Melakukan logika bisnis dan mengubah State
        // Lalu memberi tahu observer

    }

}
```

- `State`: Nilai yang menjadi perhatian para observer.

- `SomeBusinessLogic()`: Melakukan sesuatu (contoh: ubah nilai state menjadi acak 0–9), lalu memanggil `Notify()` untuk memberitahu semua observer.

Obseever (yang mengamati) `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.");

        }

    }

}
```

Kedua observer ini punya **logika reaksi berbeda** tergantung nilai dari `Subject.State`.

Main Program `class Program`

```
{     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();  // Kedua observer bereaksi sesuai
kondisi         subject.SomeBusinessLogic();


        subject.Detach(observerB);    // observerB tidak akan diberi tahu
lagi          subject.SomeBusinessLogic();  // Hanya observerA yang
bereaksi
    }
}
```

Kesimpulan

Program ini adalah **contoh penerapan pola Observer**, berguna saat kamu ingin satu objek (Subject) bisa memperbarui banyak objek (Observers) saat terjadi perubahan, **tanpa ketergantungan langsung** satu sama lain. Cocok digunakan di aplikasi GUI, sistem event, atau sistem notifikasi real-time.