Haza Zaidan Zidna Fann

2311104056

SISE 0702

https://github.com/hazazaidan/KPL_Haza_Zaidan_Zidna_Fann_2311104056_SISE0702

Tp Modul 13

A. Contoh kondisi dimana design pattern "Observer" dapat digunakan

Salah satu contoh kondisi adalah **notifikasi status build produk**. Dalam konteks kode Builder di atas, misalnya kita ingin memberikan notifikasi otomatis (misalnya ke UI atau log sistem) setiap kali bagian dari produk selesai dibangun (PartA, PartB, atau PartC). Maka, kita bisa menerapkan **Observer Pattern** agar setiap observer (misalnya UI, logger, atau monitor) mendapatkan update tanpa mengubah ConcreteBuilder.

B. Penjelasan langkah-langkah implementasi Observer Pattern

Berikut langkah-langkah umum untuk menerapkan Observer Pattern:

1. **Buat interface IObserver**: Berisi method Update() yang akan dipanggil saat ada perubahan.
2. **Buat interface ISubject**: Berisi method untuk menambahkan, menghapus, dan memberi notifikasi ke observer (Attach(), Detach(), dan Notify()).
3. **Subject konkret (ConcreteBuilder dimodifikasi)**: Menyimpan daftar observer, dan memanggil Notify() saat ada perubahan (misalnya setelah memanggil BuildPartA()).
4. **Observer konkret**: Implementasi IObserver, seperti ConsoleLogger yang menampilkan log ke konsol setiap ada update.

```
public interface IObserver
{
    void Update(string message);
}


public interface ISubject
{
    void Attach(IObserver observer);
    void Detach(IObserver observer);
    void Notify(string message);
}
public class ConcreteBuilder : IBuilder, ISubject
{
```

```
private Product _product = new Product();
private List<IObserver> _observers = new List<IObserver>();

public void Attach(IObserver observer)
{
   _observers.Add(observer);
}

public void Detach(IObserver observer)
{
   _observers.Remove(observer);
}

public void Notify(string message)
{
   foreach (var observer in _observers)
   {
      observer.Update(message);
   }
}

public void BuildPartA()
{
   this._product.Add("PartA1");
   Notify("PartA1 has been built.");
}

public void BuildPartB()
{
   this._product.Add("PartB1");
   Notify("PartB1 has been built.");
```

```
    }

    public void BuildPartC()
    {
        this._product.Add("PartC1");
        Notify("PartC1 has been built.");
    }

    // Other methods remain the same...
}
public class ConsoleLogger : IObserver
{
    public void Update(string message)
    {
        Console.WriteLine("Observer Log: " + message);
    }
}
```

C. Kelebihan dan Kekurangan Observer Pattern

Kelebihan:

1. **Low coupling**: Observer dan Subject tidak saling bergantung secara langsung.
2. **Mudah dikembangkan**: Menambahkan observer baru tidak mengubah kode Subject.
3. **Real-time update**: Cocok untuk sistem yang butuh update otomatis (event-driven).

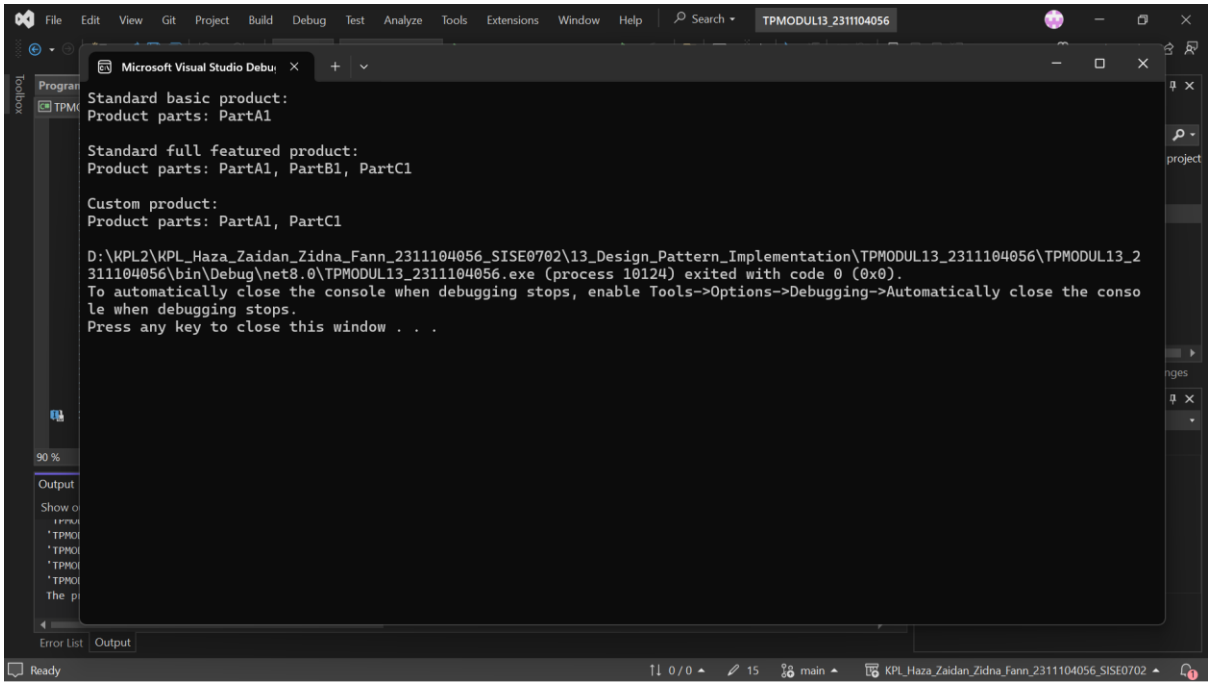
Kekurangan:

1. **Sulit dilacak**: Saat observer terlalu banyak, alur update bisa membingungkan.
2. **Potensi masalah performa**: Jika banyak observer, Notify() bisa jadi lambat.
3. **Masalah memory leak**: Jika lupa memanggil Detach(), observer bisa tidak terhapus dari memori.

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

namespace RefactoringGuru.DesignPatterns.Builder.Conceptual
{
    // The Builder interface specifies methods for creating the different parts
    // of the Product objects.
    public interface IBuilder
    {
        void BuildPartA();

        void BuildPartB();

        void BuildPartC();
    }

    // The Concrete Builder classes follow the Builder interface and provide
    // specific implementations of the building steps. Your program may have
    // several variations of Builders, implemented differently.
    public class ConcreteBuilder : IBuilder
    {
        private Product _product = new Product();

        // A fresh builder instance should contain a blank product object, which
        // is used in further assembly.
        public ConcreteBuilder()
        {
            this.Reset();
        }

        public void Reset()
        {
            this._product = new Product();
        }

        // All production steps work with the same product instance.
        public void BuildPartA()
        {
            this._product.Add("PartA1");
        }

        public void BuildPartB()
        {
            this._product.Add("PartB1");
        }

        public void BuildPartC()
        {
            this._product.Add("PartC1");
        }

        // Concrete Builders are supposed to provide their own methods for
        // retrieving results. That's because various types of builders may
        // create entirely different products that don't follow the same
        // interface. Therefore, such methods cannot be declared in the base
        // Builder interface (at least in a statically typed programming
        // language).
        //
        // Usually, after returning the end result to the client, a builder
        // instance is expected to be ready to start producing another product.
        // That's why it's a usual practice to call the reset method at the end
        // of the `GetProduct` method body. However, this behavior is not
        // mandatory, and you can make your builders wait for an explicit reset
        // call from the client code before disposing of the previous result.
        public Product GetProduct()
        {
            Product result = this._product;

            this.Reset();

            return result;
        }
    }

    // It makes sense to use the Builder pattern only when your products are
    // quite complex and require extensive configuration.
    //
    // Unlike in other creational patterns, different concrete builders can
    // produce unrelated products. In other words, results of various builders
    // may not always follow the same interface.
    public class Product
    {
        private List<object> _parts = new List<object>();

        public void Add(string part)
        {
            this._parts.Add(part);
        }

        public string ListParts()
        {
            string str = string.Empty;

            for (int i = 0; i < this._parts.Count; i++)
            {
                str += this._parts[i] + ", ";
            }

            str = str.Remove(str.Length - 2); // removing last ",c"

            return "Product parts: " + str + "\n";
        }
    }

    // The Director is only responsible for executing the building steps in a
    // particular sequence. It is helpful when producing products according to a
    // specific order or configuration. Strictly speaking, the Director class is
    // optional, since the client can control builders directly.
    public class Director
    {
        private IBuilder _builder;

        public IBuilder Builder
        {
            set { _builder = value; }
        }

        // The Director can construct several product variations using the same
        // building steps.
        public void BuildMinimalViableProduct()
        {
            this._builder.BuildPartA();
        }

        public void BuildFullFeaturedProduct()
        {
            this._builder.BuildPartA();
            this._builder.BuildPartB();
            this._builder.BuildPartC();
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // The client code creates a builder object, passes it to the
            // director and then initiates the construction process. The end
            // result is retrieved from the builder object.
            var director = new Director();
            var builder = new ConcreteBuilder();
            director.Builder = builder;

            Console.WriteLine("Standard basic product:");
            director.BuildMinimalViableProduct();
            Console.WriteLine(builder.GetProduct().ListParts());

            Console.WriteLine("Standard full featured product:");
            director.BuildFullFeaturedProduct();
            Console.WriteLine(builder.GetProduct().ListParts());

            // Remember, the Builder pattern can be used without a Director
            // class.
            Console.WriteLine("Custom product:");
            builder.BuildPartA();
            builder.BuildPartC();
            Console.Write(builder.GetProduct().ListParts());
        }
    }
}
```

Kode tersebut adalah implementasi dari **Builder Design Pattern** dalam C#. Berikut penjelasan singkatnya:

1. **Interface IBuilder:**
    - Menyediakan metode untuk membuat bagian-bagian produk (BuildPartA(), BuildPartB(), BuildPartC()).
2. **Class ConcreteBuilder:**
    - Implementasi konkret dari IBuilder.
    - Memiliki atribut _product yang merupakan instance dari Product.
    - Metode BuildPartA(), BuildPartB(), BuildPartC() menambahkan bagian tertentu ke _product.
    - GetProduct() mengembalikan produk yang telah selesai dibangun.
    - Reset() digunakan untuk menginisialisasi ulang produk.
3. **Class Product:**
    - Menyimpan daftar bagian (_parts).
    - ListParts() digunakan untuk mencetak semua bagian yang telah ditambahkan.
4. **Class Director:**
    - Mengelola urutan langkah pembuatan produk.
    - Dapat membangun produk minimal (BuildMinimalViableProduct()) atau produk penuh (BuildFullFeaturedProduct()).
5. **Class Program (Main method):**
    - Membuat instance dari Director dan ConcreteBuilder.
    - Menampilkan hasil produk standar, produk lengkap, dan produk kustom yang dibangun melalui Builder Pattern.


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search
TPMODUL13_2311104056
Microsoft Visual Studio Debu
Standard basic product:
Product parts: PartA1
Standard full featured product:
Product parts: PartA1, PartB1, PartC1
Custom product:
Product parts: PartA1, PartC1
D:\KPL2\KPL_Haza_Zaidan_Zidna_Fann_2311104056_SISE0702\13_Design_Pattern_Implementation\TPMODUL13_2311104056\TPMODUL13_2311104056\bin\Debug\net8.0\TPMODUL13_2311104056.exe (process 10124) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
Output
Error List
Output
Ready
main
KPL_Haza_Zaidan_Zidna_Fann_2311104056_SISE0702