

**TUGAS PENDAHULUAN  
KONSTRUKSI PERANGKAT LUNAK**

**MODUL XIII  
DESIGN PATTERN IMPLEMENTATION**



**Disusun Oleh:**

**Aorinka Anendya Chazanah / 2211104013**

**S1 SE-06-01**

**Dosen Pengampu:**

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

**PROGRAM STUDI S1 SOFTWARE ENGINEERING**

**FAKULTAS INFORMATIKA**

**TELKOM UNIVERSITY PURWOKERTO**

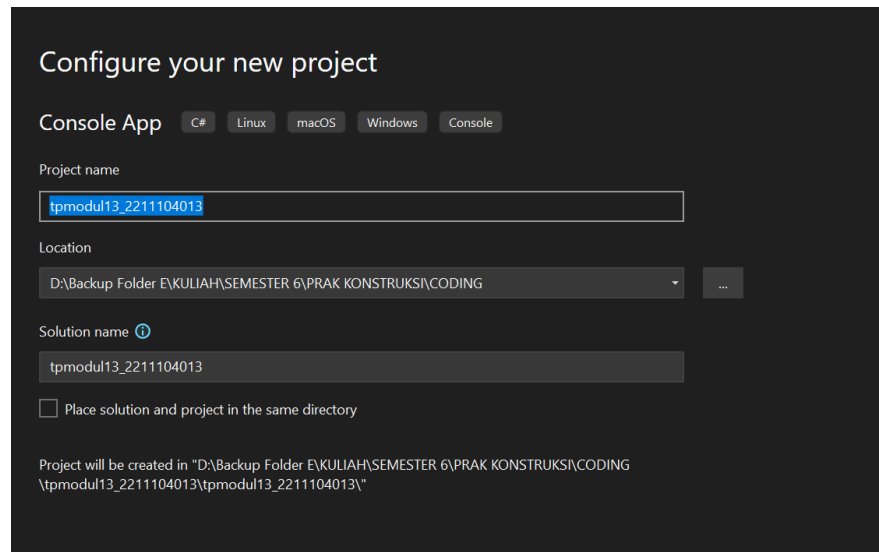
**2025**

## TUGAS PENDAHULUAN

### 1. MEMBUAT PROJECT GUI

Buka IDE misalnya dengan Visual Studio

- A. Misalnya menggunakan Visual Studio, buatlah project baru dengan nama tpmodul13\_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

**Jawaban:**

Salah satu contoh kondisi penggunaan Observer adalah pada sistem notifikasi aplikasi e-commerce. Misalnya, pengguna ingin mendapatkan pemberitahuan saat harga suatu produk turun. Dalam kasus ini:

1. Produk bertindak sebagai *publisher*, karena status harganya dapat berubah.
2. Pengguna yang ingin diberi tahu saat harga turun bertindak sebagai *subscriber*.

Dengan Observer pattern, pengguna bisa *subscribe* ke produk tertentu. Ketika harga produk berubah, sistem akan otomatis memberi tahu semua pengguna yang telah mendaftar (*subscribe*) ke produk tersebut. Ini memungkinkan pemberitahuan yang efisien dan hanya dikirim ke pihak yang relevan.

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

a. **Pisahkan logika bisnis**

Tentukan bagian program yang bertindak sebagai *publisher* (sumber perubahan) dan yang sebagai *subscriber* (yang akan merespons perubahan).

b. **Buat interface subscriber**

Interface ini minimal memiliki satu metode, seperti `update()`, yang akan dipanggil oleh *publisher* saat terjadi perubahan.

c. **Buat interface publisher**

Interface ini berisi metode untuk menambahkan, menghapus, dan memberi tahu *subscriber*, seperti `subscribe()`, `unsubscribe()`, dan `notify()`.

d. **Implementasikan daftar subscriber**

Biasanya dibuat dalam bentuk list atau map, bisa ditaruh di kelas abstrak atau helper class (komposisi), tergantung struktur program.

e. **Buat publisher konkret**

Saat terjadi perubahan status, publisher akan memanggil `notify()` untuk memberi tahu semua subscriber.

f. **Buat subscriber konkret**

Implementasikan metode `update()` untuk menentukan bagaimana objek merespons notifikasi dari publisher.

g. **Konfigurasi di Client**

Daftarkan subscriber ke publisher yang sesuai saat inisialisasi atau saat runtime, sesuai kebutuhan aplikasi.

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

**Jawaban:**

**Kelebihan Observer Pattern**

1. **Mendukung Prinsip Open/Closed:** Kamu bisa menambahkan subscriber (pengamat) baru tanpa mengubah kode publisher (subjek), sehingga kode lebih mudah dikembangkan dan dipelihara.
2. **Relasi yang Longgar (Loose Coupling):** Publisher tidak tahu detail tentang subscriber. Mereka hanya terhubung melalui antarmuka, membuat sistem lebih fleksibel dan modular.
3. **Notifikasi Otomatis:** Subscriber akan otomatis diberi tahu jika ada perubahan pada publisher, tanpa perlu melakukan polling secara manual.
4. **Fleksibel di Waktu Runtime:** Subscriber bisa bergabung atau keluar

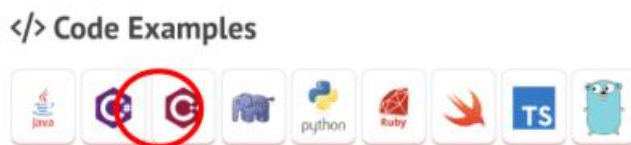
dari daftar kapan saja saat program berjalan.

## Kekurangan Observer Pattern

1. **Urutan Notifikasi Tidak Terjamin:** Subscriber menerima notifikasi dalam urutan acak, sehingga sulit jika urutan eksekusi penting.
2. **Risiko Memori Bocor:** Jika subscriber tidak dihapus dengan benar dari daftar, bisa terjadi memory leak atau notifikasi yang tidak diinginkan.
3. **Sulit untuk Menelusuri Alur Program:** Dengan banyak subscriber, alur notifikasi bisa menjadi kompleks dan sulit ditelusuri saat debugging.
4. **Potensi Overhead:** Jika jumlah subscriber besar, setiap notifikasi bisa menyebabkan overhead performa karena semua subscriber harus dipanggil.

### 3. MEMANGGIL LIBRARY DI FUNGSI UTAMA

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. Jawaban:**

Pada contoh program kali ini memilih Abstract Factory

## Kode program

```

1 using System;
2
3 namespace RefactoringPatterns.DesignPatterns.AbstractFactory.Conceptual
4 {
5     // The Abstract Factory interface declares a set of methods that return
6     // different abstract products. These products are called a family and are
7     // related by a high-level theme or concept. Products of one family are
8     // usually able to collaborate among themselves. A family of products may
9     // have several variants, but the products of one variant are incompatible
10     // with products of another.
11     //
12     public interface IAbstractFactory
13     {
14         1 reference
15         IAbstractProductA CreateProductA();
16
17         1 reference
18         IAbstractProductB CreateProductB();
19     }
20
21     // Concrete Factories produce a family of products that belong to a single
22     // variant. The factory guarantees that resulting products are compatible.
23     // Note that signature of the Concrete Factory's methods return an abstract
24     // product, while inside the method a concrete product is instantiated.
25
26     class ConcreteFactory1 : IAbstractFactory
27     {
28         2 references
29         public IAbstractProductA CreateProductA()
30         {
31             return new ConcreteProductA1();
32         }
33
34         2 references
35         public IAbstractProductB CreateProductB()
36         {
37             return new ConcreteProductB1();
38         }
39     }
40
41     // Each Concrete Factory has a corresponding product variant.
42
43     class ConcreteFactory2 : IAbstractFactory
44     {
45         2 references
46         public IAbstractProductA CreateProductA()
47         {
48             return new ConcreteProductA2();
49         }
50
51         2 references
52         public IAbstractProductB CreateProductB()
53         {
54             return new ConcreteProductB2();
55         }
56     }
57
58     // Each distinct product of a product family should have a base interface.
59     // All variants of the product must implement this interface.
60
61     public interface IAbstractProductA
62     {
63         4 references
64         string UsefulFunctionA();
65     }
66
67     // Concrete Products are created by corresponding Concrete Factories.
68
69     1 reference
70     class ConcreteProductA1 : IAbstractProductA
71     {
72         3 references
73         public string UsefulFunctionA()
74         {
75             return "The result of the product A1.";
76         }
77     }
78
79     1 reference
80     class ConcreteProductA2 : IAbstractProductA
81     {
82         3 references
83         public string UsefulFunctionA()
84         {
85             return "The result of the product A2.";
86         }
87     }
88
89     // Here's the base interface of another product. All products can
90     // interact with each other, but proper interaction is possible only between
91     // products of the same concrete variant.
92
93     1 reference
94     public interface IAbstractProductB
95

```

```

77 // Product B is able to do its own thing...
78 //reference
79 string UsefulFunctionA();
80 // ...but it also can collaborate with the ProductA.
81 // The Abstract Factory makes sure that all products it creates are of
82 // the same variant and thus, compatible.
83 //reference
84 string AnotherUsefulFunctionB(IAbstractProductA collaborator);
85 }
86 // Concrete Products are created by corresponding Concrete Factories.
87 //reference
88 class ConcreteProductB1 : IAbstractProductB
89 {
90 //reference
91 public string UsefulFunctionA()
92 {
93 return "The result of the product B1.";
94 }
95 // The variant, Product B1, is only able to work correctly with the
96 // variant, Product A1. Nevertheless, it accepts any instance of
97 // IAbstractProductA as an argument.
98 //reference
99 public string AnotherUsefulFunctionB(IAbstractProductA collaborator)
100 {
101 var result = collaborator.UsefulFunctionA();
102 return $"The result of the B1 collaborating with the {(result)}";
103 }
104 }
105 //reference
106 class ConcreteProductB2 : IAbstractProductB
107 {
108 //reference
109 public string UsefulFunctionA()
110 {
111 return "The result of the product B2.";
112 }
113 // The variant, Product B2, is only able to work correctly with the
114 // variant, Product A2. Nevertheless, it accepts any instance of
115 // IAbstractProductA as an argument.
116 //reference
117 public string AnotherUsefulFunctionB(IAbstractProductA collaborator)
118 {
119 var result = collaborator.UsefulFunctionA();
120 return $"The result of the B2 collaborating with the {(result)}";
121 }
122 }
123 //reference
124 class Client
125 {
126 // The client code works with factories and products only through abstract
127 // types: IAbstractFactory and IAbstractProduct. This lets you pass any
128 // factory or product subclass to the client code without breaking it.
129 //reference
130 public void Main()
131 {
132 // The client code can work with any concrete factory class.
133 Console.WriteLine("Client: Testing client code with the first factory type...");
134 ClientMethod(new ConcreteFactory1());
135 Console.WriteLine();
136 Console.WriteLine("Client: Testing the same client code with the second factory type...");
137 ClientMethod(new ConcreteFactory2());
138 }
139 //reference
140 public void ClientMethod(IAbstractFactory factory)
141 {
142 var productA = factory.CreateProductA();
143 var productB = factory.CreateProductB();
144 Console.WriteLine(productB.UsefulFunctionA());
145 Console.WriteLine(productB.AnotherUsefulFunction(productA));
146 }
147 }
148 //reference
149 class Program
150 {
151 //reference
152 static void Main(string[] args)
153 {
154 new Client().Main();
155 }
156 }

```

C. Jalankan program tersebut dan pastikan tidak ada error pada saat project dijalankan

## Output

```

Microsoft Visual Studio Debu x + v
Client: Testing client code with the first factory type...
The result of the product B1.
The result of the B1 collaborating with the (The result of the product A1.)

Client: Testing the same client code with the second factory type...
The result of the product B2.
The result of the B2 collaborating with the (The result of the product A2.)

D:\Backup Folder E:\WULIAH\SEMESTER 6\PRAK MONSTRUKSI\CODING\tpmodul13_2211104013\tpmodul13_2211104013\bin\Debug\net8.0\tpmodul13_2211104013.exe (process 21656) 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 . . .

```

D. Jelaskan tiap baris kode yang terdapat di bagian method utama atau “main”

### 1. class Program

Mendeklarasikan sebuah kelas bernama Program yang merupakan entri utama dari aplikasi C#.

### 2. static void Main(string[] args)

Ini adalah **method utama (entry point)** dari program. Ketika program dijalankan, eksekusi dimulai dari sini. string[] args adalah parameter opsional yang bisa menerima argumen dari command line.

### 3. new Client().Main();

- Membuat objek baru dari class Client dengan new Client().
- Langsung memanggil method Main() milik objek tersebut, yang berisi logika client untuk menguji dua concrete factory: ConcreteFactory1 dan ConcreteFactory2.
- Dalam method Client.Main(), dua factory tersebut dipakai untuk menciptakan produk dan menjalankan metode kolaborasi antar produk, serta mencetak hasilnya ke konsol.