LAPORAN PRAKTIKUM TP MODUL 13

NIM / Nama: 2311104072 – Jauhar Fajar Zuhair

Bagian I: Pendahuluan

Tugas Pendahuluan Modul 13 ini membahas salah satu Design Pattern fundamental, yaitu **Observer Pattern**. Laporan ini akan menjelaskan konsep dasar Observer Pattern berdasarkan referensi dari refactoring.guru, meliputi kondisi penggunaan, langkah implementasi, serta kelebihan dan kekurangannya. Selain itu, laporan ini juga menyajikan implementasi kode C# dari contoh konseptual Observer Pattern yang disediakan oleh refactoring.guru, seperti yang terdapat dalam file `Program.cs`.

Bagian II: Konsep Observer Pattern (Jawaban Teori - Tugas 2 TP)

A. Kondisi Penggunaan Observer Pattern:

Observer Pattern sangat berguna dalam situasi di mana:

1. **Perubahan state pada satu objek (Subject) harus secara otomatis memicu pembaruan pada objek-objek lain (Observers)** tanpa Subject perlu mengetahui secara spesifik siapa saja Observer-nya.
2. **Terdapat hubungan one-to-many dependency** antara objek, di mana satu objek (Subject) menjadi sumber informasi atau state, dan banyak objek lain (Observers) tertarik pada perubahan state tersebut.
3. **Ingin menghindari coupling yang erat** antara Subject dan Observer. Subject hanya perlu tahu tentang interface Observer, bukan implementasi konkretnya. Ini memungkinkan penambahan atau penghapusan Observer baru tanpa mengubah kode Subject.
4. **Contoh Nyata:**
    - **Antarmuka Pengguna (GUI):** Ketika data dalam model berubah (misalnya, nilai dalam spreadsheet), berbagai elemen UI (grafik, tabel, field teks) yang menampilkan data tersebut perlu diperbarui. Model adalah Subject, elemen UI adalah Observer.
    - **Sistem Notifikasi:** Ketika event penting terjadi (misalnya, stok barang menipis), berbagai pihak (manajer gudang, sistem pemesanan otomatis) perlu diberitahu. Event source adalah Subject, pihak-pihak terkait adalah Observer.
    - **Event Handling:** Komponen UI (misalnya, tombol) adalah Subject, dan kode yang bereaksi terhadap klik tombol (event listener/handler) adalah Observer.

B. Langkah-langkah Implementasi Observer Pattern:

1. **Definisikan Interface `ISubject`:** Interface ini harus mendeklarasikan metode untuk mengelola subscriber (Observer), yaitu:
    - `Attach(IObserver observer)`: Untuk menambahkan Observer baru.
    - `Detach(IObserver observer)`: Untuk menghapus Observer.
    - `Notify()`: Untuk memberitahu semua Observer yang terdaftar tentang adanya perubahan state.
2. **Definisikan Interface `IObserver`:** Interface ini harus mendeklarasikan metode yang akan dipanggil oleh Subject ketika ada pembaruan, biasanya:
    - `Update(ISubject subject)`: Metode ini dipanggil oleh Subject saat `Notify()` dieksekusi. Parameter `subject` (atau data spesifik) dikirim agar Observer bisa mendapatkan state terbaru.
3. **Buat Kelas `ConcreteSubject`:**
    - Implementasikan interface `ISubject`.
    - Tambahkan field untuk menyimpan state penting yang diminati Observer.
    - Tambahkan field (biasanya berupa `List<IObserver>`) untuk menyimpan daftar Observer yang terdaftar.
    - Implementasikan metode `Attach`, `Detach`, dan `Notify`. `Notify` akan mengiterasi daftar Observer dan memanggil metode `Update` pada masing-masing Observer.
    - Implementasikan logika bisnis inti. Ketika state penting berubah, panggil metode `Notify()`.
4. **Buat Kelas `ConcreteObserver`:**
    - Implementasikan interface `IObserver`.
    - Simpan referensi ke `ConcreteSubject` jika diperlukan untuk mengambil state detail (opsional, bisa juga state dikirim langsung via `Update`).
    - Implementasikan metode `Update`. Di dalam metode ini, Observer akan mengambil state terbaru dari Subject (jika perlu) dan melakukan aksi yang sesuai (misalnya, memperbarui tampilan, mencatat log).

C. Kelebihan dan Kekurangan Observer Pattern:

Kelebihan:

- **Loose Coupling:** Subject tidak terikat erat dengan Concrete Observer. Subject hanya berinteraksi melalui interface `IObserver`. Ini meningkatkan fleksibilitas dan kemudahan pemeliharaan.
- **Open/Closed Principle:** Kita dapat memperkenalkan jenis Observer baru tanpa harus memodifikasi kode Subject. Cukup buat kelas Observer baru yang mengimplementasikan `IObserver` dan daftarkan ke Subject.
- **Broadcast Communication:** Subject dapat memberitahu banyak Observer sekaligus dengan satu panggilan `Notify()`.
- **Dynamic Relationships:** Hubungan antara Subject dan Observer dapat dibangun dan diubah saat runtime.

Kekurangan:

- **Urutan Notifikasi Tidak Terjamin:** Observer akan diberitahu dalam urutan yang mungkin tidak dapat diprediksi (tergantung implementasi penyimpanan observer, misal `List`).
- **Potensi Update Beruntun (Cascading Updates):** Jika Observer juga merupakan Subject, satu notifikasi dapat memicu serangkaian notifikasi lain, yang bisa menjadi kompleks dan sulit di-debug.
- **Lapsed Listener Problem (Memory Leaks):** Jika Observer tidak di-`Detach` dengan benar (terutama dalam kasus di mana Observer memiliki siklus hidup yang lebih pendek dari Subject), referensi ke Observer akan tetap disimpan oleh Subject, mencegah garbage collection dan menyebabkan kebocoran memori.
- **Overhead Notifikasi:** Terkadang Subject melakukan perubahan kecil berkali-kali, menyebabkan banyak notifikasi yang mungkin tidak efisien.

Bagian III: Implementasi Kode (`Program.cs`)

Kode C# dalam `Program.cs` mengimplementasikan contoh konseptual dari Observer Pattern:

1. Interfaces:

- `IObserver`: Mendefinisikan metode `Update(ISubject subject)`. Observer harus bisa bereaksi terhadap pembaruan dari Subject.
- `ISubject`: Mendefinisikan metode `Attach(IObserver observer)`, `Detach(IObserver observer)`, dan `Notify()`. Subject harus bisa mengelola subscriber dan memberitahu mereka.

2. Concrete Subject (`Subject` class):

- Mengimplementasikan `ISubject`.
- Memiliki properti `State` (integer) yang merupakan data penting yang diamati.
- Menyimpan daftar Observer dalam `private List<IObserver> _observers`.
- `Attach(IObserver observer)`: Menambahkan observer ke `_observers`.
- `Detach(IObserver observer)`: Menghapus observer dari `_observers`.
- `Notify()`: Mengiterasi `_observers` dan memanggil `observer.Update(this)` untuk setiap observer, mengirimkan dirinya sendiri sebagai konteks.
- `SomeBusinessLogic()`: Metode ini mensimulasikan perubahan state (`State` diisi nilai random) dan kemudian memanggil `Notify()` untuk memberitahu semua observer yang terdaftar.

3. Concrete Observers (`ConcreteObserverA`, `ConcreteObserverB` classes):

- Mengimplementasikan `IObserver`.
- `Update(ISubject subject)`: Metode ini dipanggil oleh `Subject.Notify()`. Di dalamnya:
    - Observer memeriksa apakah `subject` adalah instance dari `Subject`.

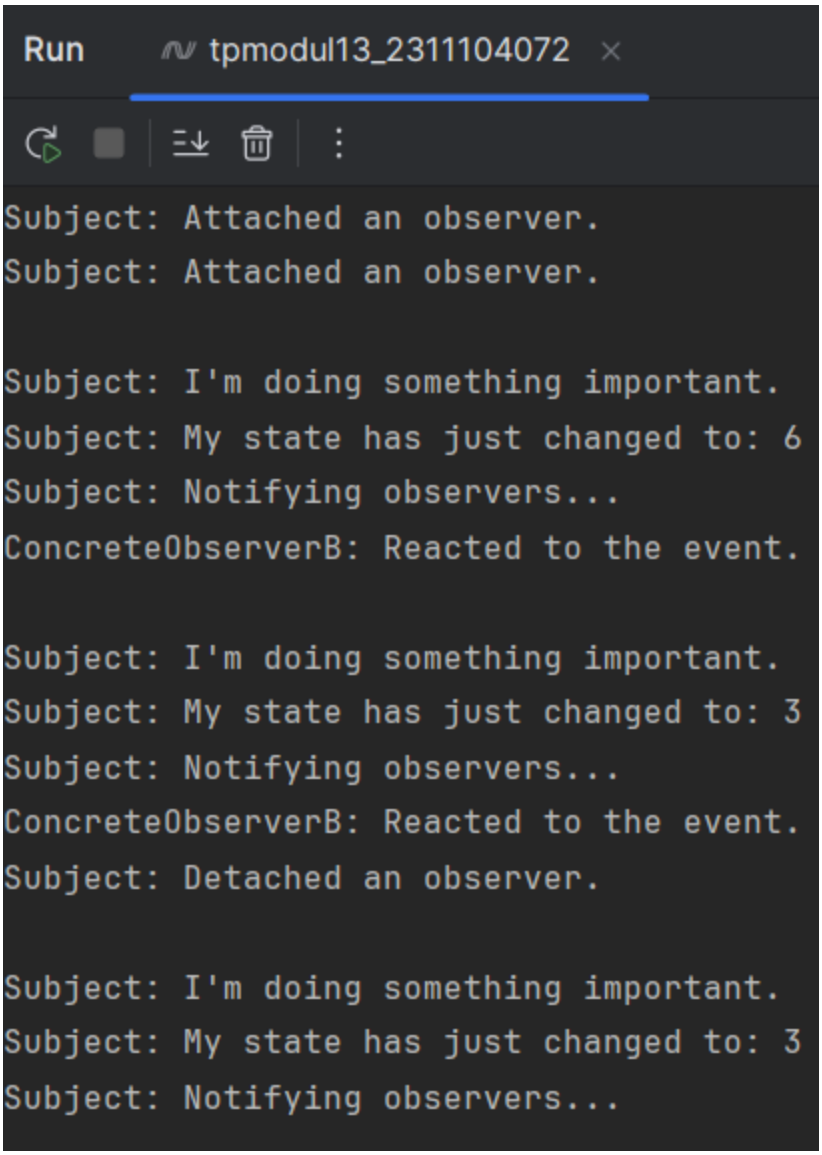
    - Observer memeriksa `subject.State`.
    - `ConcreteObserverA` hanya bereaksi (mencetak pesan) jika `subject.State < 3`.
    - `ConcreteObserverB` hanya bereaksi (mencetak pesan) jika `subject.State >= 3`.
    - Ini menunjukkan bagaimana observer yang berbeda dapat bereaksi secara berbeda atau berdasarkan kondisi tertentu terhadap perubahan state yang sama.

4. Demonstrasi (`Program.Main` method):

- Membuat satu instance `Subject`.
- Membuat satu instance `ConcreteObserverA` dan satu instance `ConcreteObserverB`.
- Mendaftarkan kedua observer ke subject menggunakan `subject.Attach()`.
- Memanggil `subject.SomeBusinessLogic()` dua kali. Setiap panggilan akan mengubah `subject.State` secara acak dan memanggil `Notify()`, yang kemudian akan memanggil `Update` pada Observer A dan B. Observer A atau B (atau tidak keduanya, tergantung nilai random `State`) akan mencetak pesan reaksi.
- Melepaskan `observerB` menggunakan `subject.Detach()`.
- Memanggil `subject.SomeBusinessLogic()` lagi. Kali ini, `Notify()` hanya akan memanggil `Update` pada `observerA` karena `observerB` sudah dilepas. Hanya `observerA` yang berpotensi mencetak pesan reaksi.

Bagian IV: Hasil Eksekusi (Contoh Tampilan Konsol)

Karena metode `SomeBusinessLogic` menggunakan `Random`, output `State` dan reaksi Observer akan bervariasi setiap kali program dijalankan. Berikut adalah *salah satu contoh kemungkinan* output:



Penjelasan Contoh Output:

1. Dua observer (A dan B) di-attach.
2. Logika bisnis pertama dijalankan, state menjadi 5. Notifikasi dikirim. Observer B bereaksi karena 5 >= 3.
3. Logika bisnis kedua dijalankan, state menjadi 1. Notifikasi dikirim. Observer A bereaksi karena 1 < 3.
4. Observer B di-detach.
5. Logika bisnis ketiga dijalankan, state menjadi 8. Notifikasi dikirim *hanya* ke Observer A. Observer A tidak bereaksi karena 8 tidak < 3. Observer B tidak menerima notifikasi sama sekali.
