41. Observer

Observer is a design pattern that facilitates communication between objects via publication and subscription model. When using this pattern, objects can subscribe to each other, so they would get notified when specific events are triggered. They can also unsubscribe from each other at any point.

Observer design pattern can be summarized as follows:

- **Subject** or **Publisher** is an object that emits some events
- **Observer** or **Subscriber** is an object that can subscribe to the **Subject**, so it can receive notifications when a specific event occurs.
- **Subject** exposes methods that allow **Observer** objects to subscribe to or unsubscribe from it.
- **Subject** maintains internal list of all **Observer** objects that are subscribed to it.

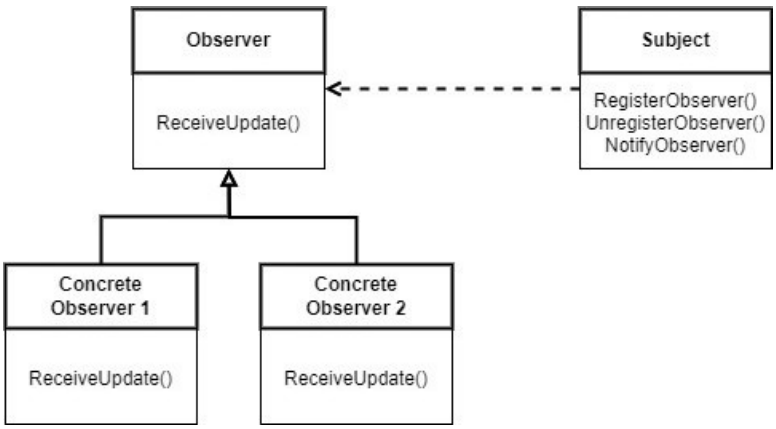

Figure 41.1 - Observer UML diagram

We will now go through an example implementation of Observer. The complete solution can be found via the link below:

https://github.com/fiodarsazanavets/design-patterns-in-csharp/tree/main/Behavioral_Patterns/Observer

Prerequisites

In order to be able to implement the code samples below, you need the following installed on your machine:

- .NET 8 SDK (or newer)
- A suitable IDE or a code editor (Visual Studio, Visual Studio Code, JetBrains Rider)

Observer implementation example

In this example, you will see how easily **Observer** objects can subscribe to events and unsubscribe from them. As in all other examples, we will use the console application template for this demo.

First, we will add an interface for our **Subject**. This interface would allow **Observer** objects to subscribe and unsubscribe. We can also use this interface to notify all the **Observer** objects that are subscribed to the **Subject**.

And don't worry about the compilation error related to `IObserver` not being present. We will add it later.

```
namespace Observer_Demo;

internal interface ISubject
{
    string Name { get; }
    void Subscribe(IObserver observer);
    void Unsubscribe(IObserver observer);
    void Notify(string message);
}
```

And this will be the interface for our **Observer** objects. It allows each of these objects to be notified by a specific **Subject**.

```
namespace Observer_Demo;

internal interface IObserver
{
    void Update(
        ISubject subject,
        string message);
}
```

Next, we will add an implementation of our **Subject**. Because a **Subject** can also be referred to as a **Publisher**, we have called our class `Publisher`.

```
namespace Observer_Demo;

internal class Publisher : ISubject
{
        private string name;
        private List<IObserver> observers;

        public Publisher(string name)
        {
                this.name = name;
                observers =
            new List<IObserver>();
        }

        public string Name => name;

        public void Subscribe(
        IObserver observer)
    {
                observers.Add(observer);
    }

        public void Unsubscribe(
        IObserver observer)
    {
                observers.Remove(observer);
    }

        public void Notify(
        string message)
    {
                foreach (var observer in observers)
        {
                        observer
            .Update(this, message);
```

```
36                  }
37      }
38  }
```

This class has a list of **Observer** objects. When something calls the `Notify` method, every object of this list gets updated. Adding a new **Observer** object to the list is done via the `Subscribe` method. Removing an **Observer** from the list is done via the `Unsubscribe` method.

Now, we will add an implementation of our **Observer**. As an **Observer** object is also referred to as a **Subscriber**, we have called our class `Subscriber`.

```
1   namespace Observer_Demo;
2
3   internal class Subscriber(
4       string name) : IObserver
5   {
6           public void Update(
7           ISubject subject,
8           string message)
9       {
10                  Console.WriteLine(
11              $"'{message
12              }' message received from {subject.Name
13              } by {name}.");
14      }
15  }
```

Now, we will bring it all together. We will replace the content of `Program.cs` file with the following code:

```
using Observer_Demo;

var publisher =
    new Publisher(
        "Message Hub");
var subscriber1 =
    new Subscriber(
        "First Subscriber");
var subscriber2 =
    new Subscriber(
        "Second Subscriber");
var subscriber3 =
    new Subscriber(
        "Third Subscriber");

Console.WriteLine(
    "Adding the first and the second subscribers to the p\
ublisher.");
publisher.Subscribe(
    subscriber1);
publisher.Subscribe(
    subscriber2);

Console.WriteLine(
    "Notifying subscribers.");
publisher.Notify(
    "Sequence initiated.");

Console.WriteLine(
    "Removing the first subscriber.");
publisher.Unsubscribe(
    subscriber1);

Console.WriteLine(
    "Adding the third subscriber.");
```

```
publisher.Subscribe(
    subscriber3);

Console.WriteLine(
    "Notifying subscribers.");
publisher.Notify(
    "Update received from the server.");

Console.ReadKey();
```

In here, we have three **Subscriber** objects. First, we will only add the first and the second **Subscribers** to the **Subject**. Then, we notify all of the subscribers that are listed by our **Subject**. Then, we remove the first subscriber and add the third one, after which we notify the subscribers again.

And, as the following screenshot shows, all the correct **Observer** objects get notified:

Figure 41.2 - Subscribing and unsubscribing by using Observer

This concluded the overview of the Observer design pattern. Let's now summarize its main benefits.

Benefits of using Observer

The key benefits of using the Observer design pattern are as follows:

- The relationships between objects can be established and removed at runtime.

- Each object has its own pre-defined role, so the single responsibility principle is well maintained.

And now let's have a look at the caveats of using the Observer design pattern that you should be familiar with.

Caveats of using Observer

Perhaps the only caveat of using the Observer design pattern is that the subscribed objects don't get updated at the same time and there is no guarantee in what order they will be updated. In most cases, it's not a problem. But it might be an issue in some scenarios.

Also, if you are building a real-life distributed application, having a proper publisher/subscription (pub/sub) system might be more appropriate than using the Observer design pattern. But still, even if you do use such a system, being familiar with how the Observer design pattern works will automatically make you understand the key principles behind any pub/sub system.