42. What is the Observer design pattern?

Brief summary: The Observer design pattern allows objects to notify other objects about changes in their state.

Observer design pattern allows objects to notify other objects about changes in their state.

Let's consider the following example. We have some class that is able to read the current Bitcoin price. In a real-life application it would read it from some public API, but for the example's sake let's make it return a random number from 0 to 50000 (looking at cryptocurrencies prices fluctuations, I would say it's not that far away from the truth).

```
public class BitcoinPriceReader
{
    private decimal _currentBitcoinPrice;

    3 references
    public void ReadCurrentPrice()
    {
        _currentBitcoinPrice = new Random().Next(0, 50000);
    }
}
```

Now, let's say we want to create a couple of mechanisms that will notify the application's users if the price has grown over a certain threshold. Let's say we want to be able to send users emails and/or push notifications.

```
class EmailPriceChangeNotifier
{
    private readonly decimal _notificationThreshold;

    1 reference
    public EmailPriceChangeNotifier(decimal notificationThreshold)
    {
        _notificationThreshold = notificationThreshold;
    }

    0 references
    public void Update(decimal currentBitcoinPrice)
    {
        if (currentBitcoinPrice > _notificationThreshold)
        {
            Console.WriteLine($"Sending an email saying that " +
                $"the Bitcoin price exceeded {_notificationThreshold} " +
                $"and is now {currentBitcoinPrice}");
        }
    }
}
```

The class for sending push notifications would be almost the same, except that the message would be different. Please notice that this is a simplification, and in a real project, those classes would actually send emails or push notifications. We could also implement more classes to perform other types of notifications.

All right, so here is the big picture: we have the **BitcoinPriceReader** that reads the price, and two classes that wait to be notified about the price change - **EmailPriceChangeNotifier** and **PushPriceChangeNotifier**. When the price is read from the BitcoinPriceReader, we want it to execute the **Update** method from both the classes that wait for the information about the new price:

```
public class BitcoinPriceReader
{
    private decimal _currentBitcoinPrice;

    private readonly EmailPriceChangeNotifier _emailPriceChangeNotifier;
    private readonly PushPriceChangeNotifier _pushPriceChangeNotifier;

    1 reference
    public BitcoinPriceReader(
        EmailPriceChangeNotifier emailPriceChangeNotifier,
        PushPriceChangeNotifier pushPriceChangeNotifier)
    {
        _emailPriceChangeNotifier = emailPriceChangeNotifier;
        _pushPriceChangeNotifier = pushPriceChangeNotifier;
    }

    3 references
    public void ReadCurrentPrice()
    {
        _currentBitcoinPrice = new Random().Next(0, 50000);
        _emailPriceChangeNotifier.Update(_currentBitcoinPrice);
        _pushPriceChangeNotifier.Update(_currentBitcoinPrice);
    }
}
```

Well.. this is awkward, at least. First of all, it **tightly couples** the BitcoinPriceReader with the other two classes. Secondly, this way we will only notify a single EmailPriceChangeNotifier object and a single PushPriceChangeNotifier object. What if we wanted to notify a whole group of them? Lastly, what if some of those objects will no longer be interested in listening about the price changes (for example the user of the application decides to sell all his or her crypto and move to live in the Bahamas?). We won't have any control over what objects we notify.

It's time to introduce the **Observer design pattern**. Let's do it step by step.

First of all, we want to decouple the BitcoinPriceReader (the **Observable**) from the EmailPriceChangeNotifier and PushPriceChangeNotifier (the **Observers**). We will need to define interfaces over which they can communicate. The first question we need to ask is "what **data** will be sent from the Observable to the Observers?". In our case, it will be the current Bitcoin price, so a decimal, but let's make the interfaces generic, so they can work with any payload. First, let's define the IObserver interface, which will be implemented by EmailPriceChangeNotifier and PushPriceChangeNotifier. This interface will contain a single Update method, which will be called by the Observable to send the data to the Observers:

```
3 references
public interface IObserver<TData>
{
    2 references
    void Update(TData data);
}
```

Let's use this interface before we move on to IObservable:

```
public class EmailPriceChangeNotifier : IObserver<decimal>
{
    private readonly decimal _notificationThreshold;

    1 reference
    public EmailPriceChangeNotifier(decimal notificationThreshold)
    {
        _notificationThreshold = notificationThreshold;
    }

    2 references
    public void Update(decimal currentBitcoinPrice)
    {
        if (currentBitcoinPrice > _notificationThreshold)
        {
            Console.WriteLine($"Sending an email saying that " +
                $"the Bitcoin price exceeded {_notificationThreshold} " +
                $"and is now {currentBitcoinPrice}");
        }
    }
}
```

In this case, the method was already implemented, so not much to do here. In general, the Update method is the one that receives the notification from the Observable and decides what to do about it. I also added the interface implementation to the PushPriceChangeNotifier.

Let's now define the IObservable interface.

```
public interface IObservable<TData>
{
    0 references
    void AttachObserver(IObserver<TData> observer);
    0 references
    void DetachObserver(IObserver<TData> observer);
    0 references
    void NotifyObservers();
}
```

The first two methods are used to attach (or “subscribe”) the observer to the observable. This way we will have control over who is notified. We can detach (or “unsubscribe”) the observers at any time if they are no longer interested in receiving the notifications from the Observable.

The last method will be executed to send the notification to all subscribed observers.

Let’s implement this interface in the BitcoinPriceReader class. First, we need to define a collection of Observers:

```
public class BitcoinPriceReader : IObservable<decimal>
{
    private decimal _currentBitcoinPrice;
    private List<IObserver<decimal>> _observers =
        new List<IObserver<decimal>>();

    3 references
    public void AttachObserver(IObserver<decimal> observer)
    {
        _observers.Add(observer);
    }

    2 references
    public void DetachObserver(IObserver<decimal> observer)
    {
        _observers.Remove(observer);
    }
```

The NotifyObservers method will simply iterate the List of Observers and execute the Update method on them with the _currentBitcoinPrice:

```
public void NotifyObservers()
{
    foreach (var observer in _observers)
    {
        observer.Update(_currentBitcoinPrice);
    }
}
```

The only thing left to do is to call the NotifyObservers method after the latest Bitcoin price has been read:

```
2 references
public void ReadCurrentPrice()
{
    _currentBitcoinPrice = new Random().Next(0, 50000);

    NotifyObservers();
}
```

As you can see the NotifyObservers method could be private, but I'll leave it public as this is the most typical implementation of the Observer design pattern.

All right, let's put it all together. First, let's create the Observers and attach them to the Observable. Let's say the email should be sent if the price exceeds 25000, and push notification - when it exceeds 40000.

```
var bitcoinPriceReader = new BitcoinPriceReader();

var emailPriceChangeNotifier = new EmailPriceChangeNotifier(25000);
bitcoinPriceReader.AttachObserver(emailPriceChangeNotifier);

var pushPriceChangeNotifier = new PushPriceChangeNotifier(40000);
bitcoinPriceReader.AttachObserver(pushPriceChangeNotifier);
```

Now, let's execute the ReadCurrentPrice method couple of times:

```
bitcoinPriceReader.ReadCurrentPrice();
bitcoinPriceReader.ReadCurrentPrice();
```

And here is the result:

```
Sending an email saying that the Bitcoin price exceeded 25000 and
is now 44326

Sending a push notification saying that the Bitcoin price exceeded
 40000 and is now 44326
```

It seems like one of the calls triggered both email and push notifications, and the other did not trigger any of them (so the price must have been below 25000).

Now, let's detach the PushPriceChangeNotifier:

```
Console.WriteLine("Push notifications OFF");
bitcoinPriceReader.DetachObserver(pushPriceChangeNotifier);
```

And call the ReadCurrentPrice method again:

```
Sending an email saying that the Bitcoin price exceeded 25000
 and is now 35705

Push notifications OFF
Sending an email saying that the Bitcoin price exceeded 25000
 and is now 47023
```

As you can see, after the push notifications have been unsubscribed, they are not sent even if the price exceeded 40000.

Remember that this code bases on random numbers, so when you execute it, you will have different results. Run it a couple of times and see what happens!

All right. We implemented the basic Observer design patterns.

Please note that there is an existing Microsoft's implementation of this pattern, but it's a bit more complex. I wanted to show you custom implementation so you

see exactly what is going on. If you are curious about Microsoft's implementation, make sure to read this article:
https://docs.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern

We will revisit the topic of the Observer design pattern in the next lecture, where we will talk about events, as they have very much in common.

Bonus questions:

- "**In the Observer design pattern, what is the Observable and what is the Observer?**"
  The Observable is the object that's being observed by Observers. The Observable notifies the Observers about the change in its state.