Observer Pattern

- A *design pattern* from the GoF book.
- **Tight coupling**: The subject (publisher) **knows** its observers directly.
- Communication is usually **in-process**, inside the same application.
- Observers subscribe directly to the subject object.
- Example in C#: INotifyPropertyChanged, event handlers (event Action).

Flow:

Subject → notifies → Observers (direct references)

Publish–Subscribe Pattern (Pub-Sub)

- An **architectural pattern**, typically used across distributed systems.
- **Loose coupling**: Publisher and subscriber **do NOT know about each other**.
- A **message broker** (like SNS, Kafka, RabbitMQ, EventBridge) sits in the middle.
- Subscribers receive messages via channels/topics, not direct references.

Flow:

Publisher → **Broker/Topic** → Subscribers (via channels)

Key Differences Table

Feature	**Observer Pattern**	**Pub-Sub Pattern**
Coupling	Tight (subject knows observers)	Loose (publisher doesn't know subscribers)
Scope	In-process (same app)	Cross-system (distributed)
Broker?	No	Yes
Usage	GUIs, in-app events	Microservices, messaging systems
Relation	Design pattern	Architectural style

Summary

➞ Observer = direct notifications within the same app.
➞ Pub-Sub = notifications through a broker, typically in distributed systems.

They share the same idea of “event notification,” but **Pub-Sub is a more scalable, decoupled version** used between services rather than between objects.

1. Real-World Example: Stock Price Tracker (Classic Observer Pattern)

A *Stock* notifies multiple *Investor* objects whenever its price changes.

Subject (Publisher)

```
public interface IStock
{
   void Attach(IInvestor investor);
   void Detach(IInvestor investor);
   void Notify();
}

public class Stock : IStock
{
   private readonly List<IInvestor> _investors = new();
   private decimal _price;

   public string Symbol { get; }

   public Stock(string symbol, decimal price)
   {
      Symbol = symbol;
      _price = price;
   }

   public decimal Price
   {
      get => _price;
      set
      {
         if (_price != value)
         {
            _price = value;
```

```
            Notify();
        }
    }
}

public void Attach(IInvestor investor) => _investors.Add(investor);
public void Detach(IInvestor investor) => _investors.Remove(investor);

public void Notify()
{
    foreach (var investor in _investors)
        investor.Update(this);
}
}
```

Observer

```
public interface IInvestor
{
    void Update(Stock stock);
}

public class Investor : IInvestor
{
    public string Name { get; }

    public Investor(string name)
    {
        Name = name;
    }

    public void Update(Stock stock)
```

```
    {
        Console.WriteLine($"{Name} notified: {stock.Symbol} price changed to {stock.Price}");
    }
}
```

Usage:

```
var googleStock = new Stock("GOOGL", 1500);

var rishi = new Investor("Rishi");
var john = new Investor("John");

googleStock.Attach(rishi);
googleStock.Attach(john);

googleStock.Price = 1520; // Both observers get notified
googleStock.Price = 1540; // Both observers get notified
```

2. Real-World Example: Weather App (Observer in a UI scenario)

A WeatherStation notifies multiple displays when temperature changes.

Subject

```
public class WeatherStation
{
    private readonly List<IWeatherDisplay> _displays = new();
    private float _temperature;

    public float Temperature
    {
        get => _temperature;
        set
        {
            _temperature = value;
```

```
            Notify();
        }
    }

    public void Subscribe(IWeatherDisplay display) => _displays.Add(display);
    public void Unsubscribe(IWeatherDisplay display) => _displays.Remove(display);

    private void Notify()
    {
        foreach (var display in _displays)
            display.Update(_temperature);
    }
}
```

Observer

```
public interface IWeatherDisplay
{
    void Update(float temperature);
}

public class PhoneDisplay : IWeatherDisplay
{
    public void Update(float temperature)
    {
        Console.WriteLine($"Phone Display: Temperature updated to {temperature}°C");
    }
}

public class TabletDisplay : IWeatherDisplay
{
    public void Update(float temperature)
```

```
    {
        Console.WriteLine($"Tablet Display: Temperature updated to {temperature}°C");
    }
}
```

Usage

```
var station = new WeatherStation();

station.Subscribe(new PhoneDisplay());
station.Subscribe(new TabletDisplay());

station.Temperature = 26.5f;
station.Temperature = 27.1f;
```

3. Real-World Example: Using C# Events (Built-In Observer Pattern)

This is how the observer pattern is implemented natively in C# with event and delegate.

Publisher

```
public class FileUploader
{
    public event Action<int> ProgressChanged;

    public async Task UploadFile()
    {
        for (int i = 1; i <= 100; i++)
        {
            await Task.Delay(50);
            ProgressChanged?.Invoke(i);
        }
    }
}
```

Observer

```
public class ProgressBar
{
    public void Subscribe(FileUploader uploader)
    {
        uploader.ProgressChanged += OnProgressChanged;
    }

    private void OnProgressChanged(int progress)
    {
        Console.WriteLine($"Progress: {progress}%");
    }
}
```

Usage

```
var uploader = new FileUploader();
var progressBar = new ProgressBar();

progressBar.Subscribe(uploader);

await uploader.UploadFile();
```

the **Observer Pattern** has two variants:

1. PUSH Model

2. PULL Model

Let's break them down **very clearly** with **simple examples**.

1. PUSH Model (Publisher pushes all data to observers)

Concept

- The **Subject sends the full updated data directly to the Observer**.
- Observer doesn't need to request anything.

- More overhead if the data is large or observers don't need everything.

Real-life analogy

You subscribe to news notifications →
The news app sends **the entire article** as a notification.
Even if you only needed the headline.

Pros

- Fast notifications
- Observer gets everything immediately

Cons

- Observers may get data they don't need
- Higher network/CPU load

PUSH Model Example (Stock Price)

Subject

```
public interface IObserver
{
   void Update(decimal newPrice);  // PUSH full data
}


public interface ISubject
{
   void Attach(IObserver observer);
   void Detach(IObserver observer);
   void Notify();
}


public class Stock : ISubject
{
   private readonly List<IObserver> _observers = new();
   private decimal _price;


   public decimal Price
```

```
    {
        get => _price;
        set
        {
            _price = value;
            Notify();
        }
    }

    public void Attach(IObserver observer) => _observers.Add(observer);
    public void Detach(IObserver observer) => _observers.Remove(observer);

    public void Notify()
    {
        foreach (var observer in _observers)
            observer.Update(_price);  // PUSH the latest price
    }
}

Observer
public class Investor : IObserver
{
    public void Update(decimal newPrice)
    {
        Console.WriteLine($"Investor received new price: {newPrice}");
    }
}
```

2. PULL Model (Publisher only notifies; observer pulls what it wants)

Concept

- The **Subject sends a simple notification** like:
  "Something changed."
- Observer then **pulls only the data it needs**.

Real-life analogy

You get a notification:
"New article available."
You open the app and fetch only the part you want.

Pros

- More efficient for large data
- Observers choose what to fetch
- Very flexible (different observers can pull different parts)

Cons

- Observers must query subject
- More calls between subject & observer

PULL Model Example (Stock Price)

Observer interface only gets a notification

```
public interface IObserver
{
   void Update(ISubject subject);  // PULL — only reference to subject
}


public interface ISubject
{
   void Attach(IObserver observer);
   void Detach(IObserver observer);
   void Notify();
}


Subject
public class Stock : ISubject
{
```

```
    private readonly List<IObserver> _observers = new();
    private decimal _price;

    public decimal Price => _price;  // Exposed for pulling

    public decimal SetPrice(decimal price)
    {
        _price = price;
        Notify();
        return _price;
    }

    public void Attach(IObserver observer) => _observers.Add(observer);
    public void Detach(IObserver observer) => _observers.Remove(observer);

    public void Notify()
    {
        foreach (var observer in _observers)
            observer.Update(this);  // Only sends reference
    }
}

Observer:
public class Investor : IObserver
{
    public void Update(ISubject subject)
    {
        if (subject is Stock stock)
        {
            Console.WriteLine($"Investor pulled new price: {stock.Price}");
        }
```

```
    }
}
```

Push vs Pull: Quick Comparison

Feature	**PUSH Model**	**PULL Model**
Notification contains data?	Yes (full data)	No (observer pulls)
Efficient for large data?	No	Yes
Observer flexibility	Low	High
Publisher–Observer coupling	Higher	Lower
When to use	Small updates	Large state objects

Real-World Example: Fitness Tracker (Heart Rate Observer)

Scenario:

A **FitnessTracker** device measures heart rate.
Observers:

- **MobileApp** → Shows your current BPM
- **HealthAlertService** → Warns if BPM too high
- **CloudSyncService** → Uploads BPM data when needed

This is a very realistic use case.

PUSH MODEL Example (FitnessTracker → Pushes Heart Rate)

Subject (Publisher)

```
public interface IObserver
{
    void Update(int newHeartRate); // PUSH full data
}

public interface ISubject
```

```
{
    void Attach(IObserver observer);
    void Detach(IObserver observer);
    void Notify();
}

public class FitnessTracker : ISubject
{
    private readonly List<IObserver> _observers = new();
    private int _heartRate;

    public int HeartRate
    {
        get => _heartRate;
        set
        {
            _heartRate = value;
            Notify(); // notify immediately
        }
    }

    public void Attach(IObserver observer) => _observers.Add(observer);
    public void Detach(IObserver observer) => _observers.Remove(observer);

    public void Notify()
    {
        foreach (var obs in _observers)
            obs.Update(_heartRate);  // PUSH MODEL
    }
}
```

Observers (PUSH)

```
public class MobileApp : IObserver
{
   public void Update(int newHeartRate)
   {
      Console.WriteLine($"Mobile App → Heart Rate Updated: {newHeartRate} BPM");
   }
}

public class CloudSyncService : IObserver
{
   public void Update(int newHeartRate)
   {
      Console.WriteLine($"Cloud Sync → Uploading heart rate: {newHeartRate}");
   }
}
```

Usage

```
var tracker = new FitnessTracker();

tracker.Attach(new MobileApp());
tracker.Attach(new CloudSyncService());

tracker.HeartRate = 85;
tracker.HeartRate = 102;
```

PULL MODEL Example (FitnessTracker → Only notifies; observers pull data)

Observer Interface (Pull-style)

```
public interface IPullObserver
{
```

```
    void Update(ISubject subject); // Only reference, no data pushed
}

Subject (Pull Ready)
public class FitnessTrackerPull : ISubject
{
    private readonly List<IPullObserver> _observers = new();
    private int _heartRate;

    public int GetHeartRate() => _heartRate;

    public int SetHeartRate(int value)
    {
        _heartRate = value;
        Notify();
        return value;
    }

    public void Attach(IPullObserver observer) => _observers.Add(observer);
    public void Detach(IPullObserver observer) => _observers.Remove(observer);

    public void Notify()
    {
        foreach (var obs in _observers)
            obs.Update(this); // PULL MODEL
    }
}

Observer (Pulls what it wants)
public class HealthAlertService : IPullObserver
{
```

```
    public void Update(ISubject subject)
    {
        if (subject is FitnessTrackerPull tracker)
        {
            int hr = tracker.GetHeartRate();
            if (hr > 120)
                Console.WriteLine($"ALERT: Dangerous Heart Rate detected: {hr} BPM");
        }
    }
}

public class MobileAppPull : IPullObserver
{
    public void Update(ISubject subject)
    {
        if (subject is FitnessTrackerPull tracker)
        {
            Console.WriteLine($"Mobile App (Pull) → BPM: {tracker.GetHeartRate()}");
        }
    }
}

Usage
var tracker = new FitnessTrackerPull();

tracker.Attach(new HealthAlertService());
tracker.Attach(new MobileAppPull());

tracker.SetHeartRate(90);
tracker.SetHeartRate(135);   // Alert triggers
```

Push vs Pull (Simple Explanation)

Model	How It Works	Example
Push Model	Subject sends data directly	Mobile App receives BPM instantly
Pull Model	Subject only sends a signal; observers pull data	Health Service pulls BPM to check if it's dangerous