Observer Pattern

LCSCI5202: Object Oriented Design Week 9

Design Patterns

- Reusable solutions to common software design problems.
- Improve code reusability, scalability, and maintainability
- Creational, Structural, and Behavioral patterns.

Observer Pattern

- One-to-many relationship: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified automatically.

- "State" refers to the current condition or data values within an object.

- Change Detection could be:

  Explicit notification: Programmer manually calls notify after state changes
  Implicit notification: Using property setters or other mechanisms to automatically trigger notifications
  Polling vs. Push: This can be a push-based system (Subject pushes notifications) or pull-based (Observers repeatedly check for changes)

- Enables loose coupling - the subject can inform others about changes without being tightly bound to specific observer implementations.

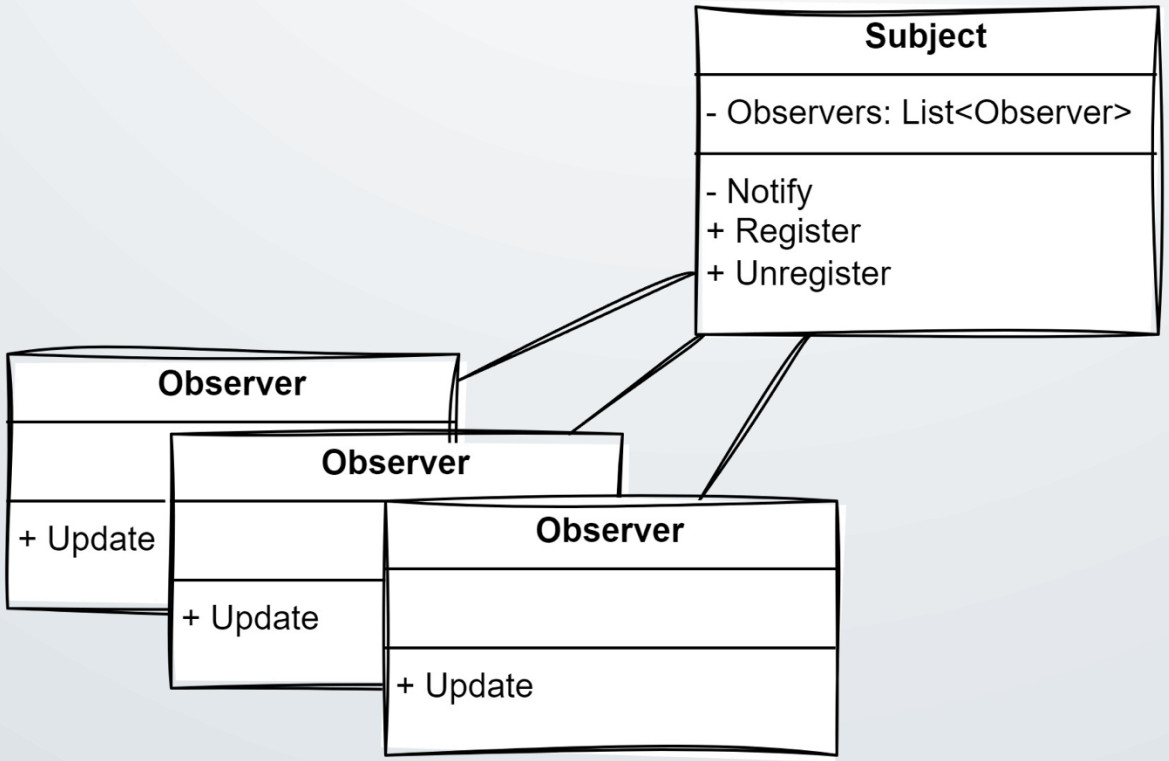
When to Use

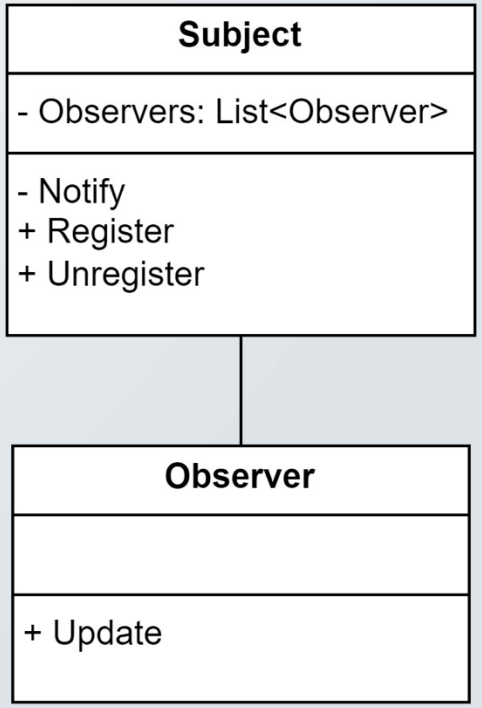
- Decoupling: Keeps the Subject independent from its Observers.
- Dynamic Subscription: Observers can subscribe/unsubscribe as needed.
- Also Known As:
    - Publish-Subscribe Pattern
    - Event-Listener Pattern
    - Model-View Pattern (part of MVC)
- Real-World Analogies
    - Magazine (Subject) publishes content
    - Weather service (Subject) detects severe weather
    - Stock (Subject) price changes on exchange.
    - Social media subscription model.

Key Concepts

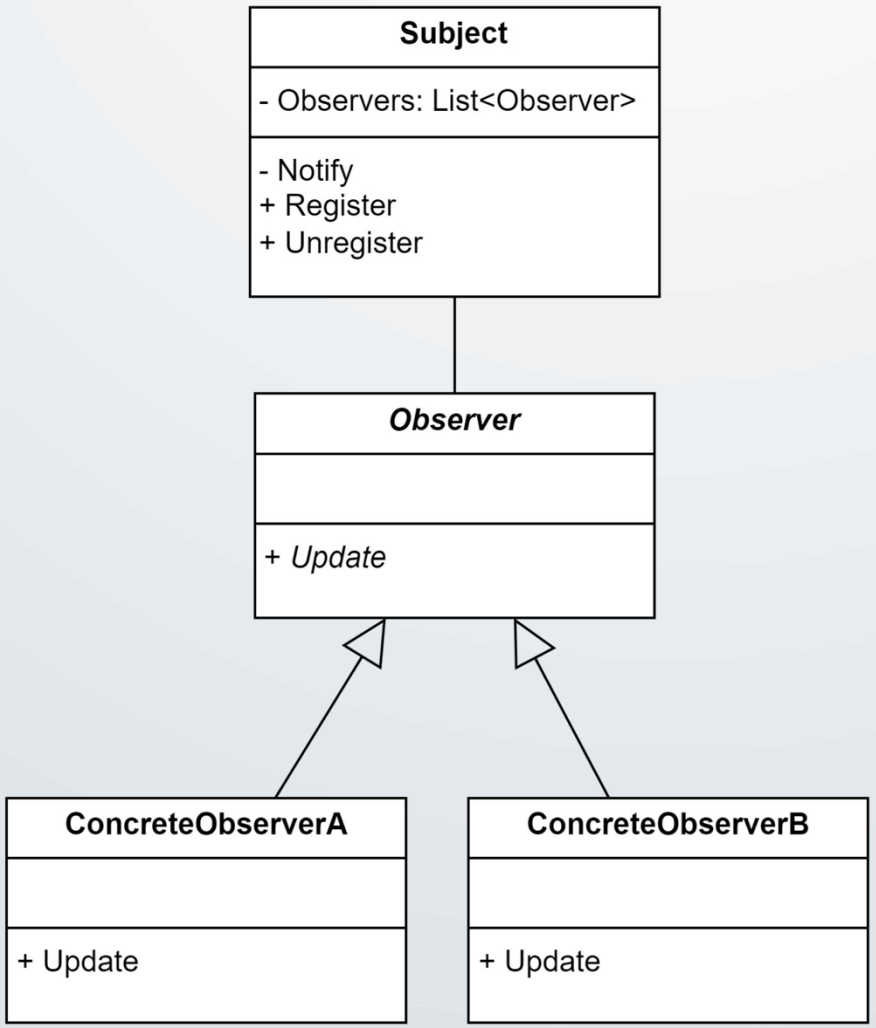
- **Subject**: Object that holds state and is being observed. It maintains a list of dependent objects (observers) and automatically notifies them when its state changes.
- **Observers**: Object that depends on the Subject's state and needs to be notified when that state changes. It defines how to respond to state change notifications..
- **Notification**: Communication mechanism or message that the Subject uses to inform all registered Observers that its state has changed..

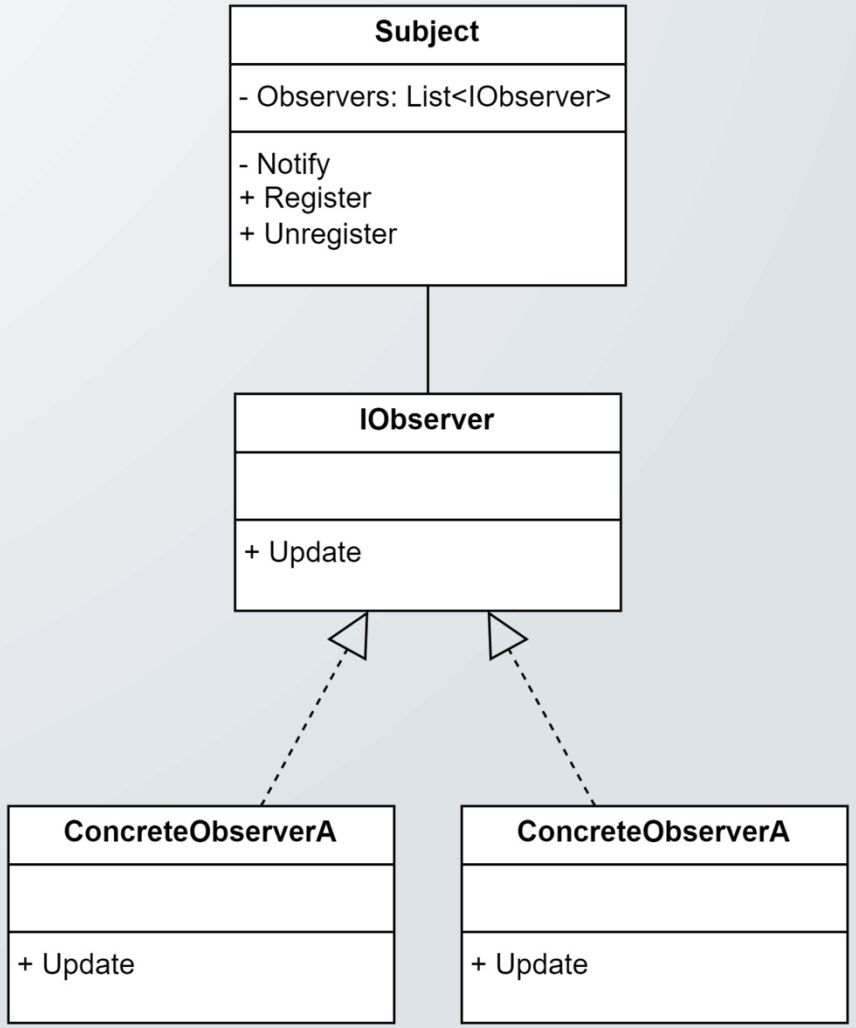
Diagram


Subject
- Observers: List<Observer>
- Notify
+ Register
+ Unregister
Observer
+ Update
Observer
+ Update
Observer
+ Update


Subject
- Observers: List<Observer>
- Notify
+ Register
+ Unregister
Observer
+ Update

Class Diagram





Example: Interfaces

```
public interface IObserver {
        void Update(string message);
}


public interface ISubject {

        void Register(IObserver observer);
        void Unregister(IObserver observer);

}
```

Classes (Subject)

```
public class NewsPublisher : ISubject {
private List<IObserver> observers = new List<IObserver>();


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

public void Unregister(IObserver observer) {
        observers.Remove(observer);
}
private void Notify(string message)
{
        foreach (var observer in observers)
                observer.Update(message);
}
public void SetState(string message)
{
        Console.WriteLine($"Subject: Changing state to '{message}'");
        Notify(message);
}
}
```

Classes (Observer)

```
public class NewsReader : IObserver {
        public void Update(string message) {
                Console.WriteLine($"Received news update: {message}");
        }

}
```

Main()

```
static void Main(string[] args)
{

        NewsPublisher np = new NewsPublisher();
        NewsReader nr1 = new NewsReader();
        NewsReader nr2 = new NewsReader();


        np.Register(nr1);
        np.Register(nr2);


        np.SetState("Hello world!");
}
```

Summary

- A powerful behavioral pattern that enables loose coupling between objects through event-driven communication.

- Subjects: Holds the state, Maintains observer list, and Notifies observers of changes.
- Observers: Implements update interface, Responds to notifications, Can register/unregister.

- Two Main Approaches: Push Model and Pull Model

- Enables dynamic relationships between objects.

Activity

- Consider the provided code (10.A.Code – Observer).
- Extend the weather station program by adding a sensor that measures temperature. This way, the weather station can use a sensor to get its data instead of generating random values directly. This will also make it easier to add more sensors in the future, like for humidity or pressure. If possible use inheritance and abstract classes.