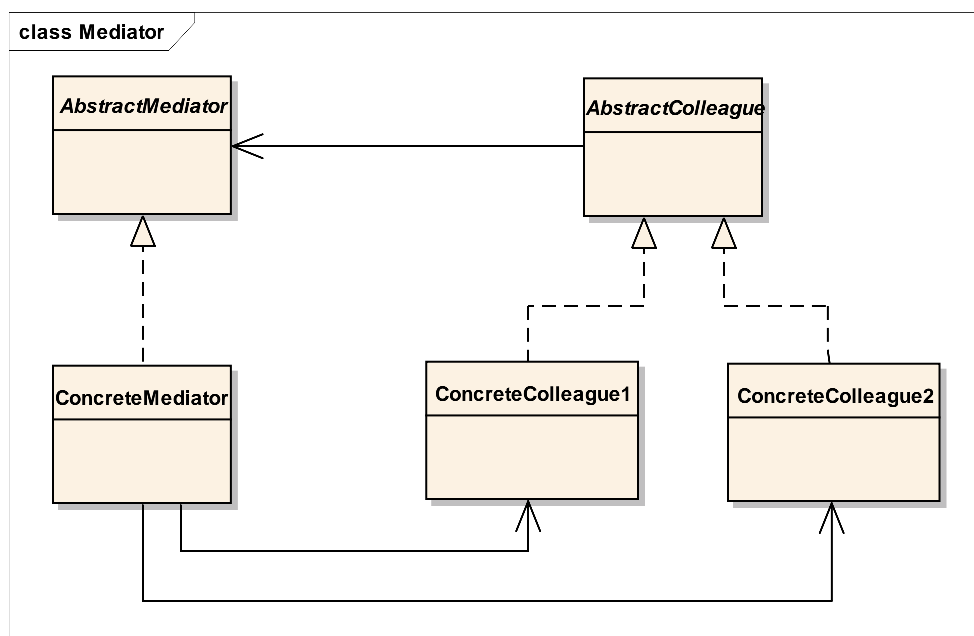
Projektowanie obiektowe oprogramowania
Wykład 7 – wzorce czynnościowe (2)
Wiktor Zychla 2019

1 Mediator

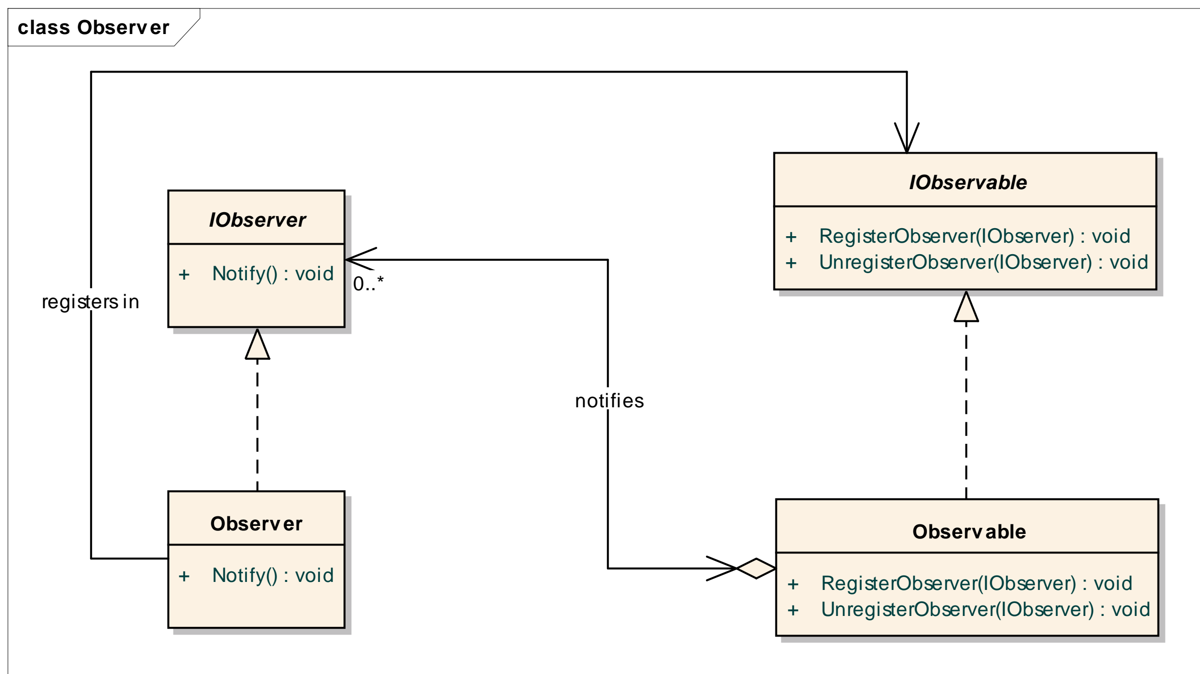
Motto: Koordynator współpracy ściśle określonej grupy obiektów – dzięki niemu one nie odwołują się do siebie wprost (nie muszą nic o sobie wiedzieć), ale przesyłają sobie powiadomienia przez mediatora.
Kojarzyć: niby Observer bo też „powiadomienia", ale zbiór współpracujących obiektów jest tu ściśle określony. Mediator może więc wykorzystać ten fakt do wyboru różnych technik przesyłania powiadomień (bezpośrednio, na styl observera itp.).
Druga różnica między Mediatorem a Observerem jest taka że to kolaborujące obiekty przesyłają sobie powiadomienia o zmianie swojego stanu, a stan Mediatora nie ma nic do tego. W Observerze wszyscy zainteresowani nasłuchują powiadomień o zmianie stanu obiektu obserwowanego. Nie ma więc zupełnie analogii między mediatorem a obserwowanym.
Przykład z życia: typowe okienka desktopowych technologii wytwarzania GUI są mediatorami między konkretnymi kontrolami, które są zagregowane wewnątrz (w środku okienka – Mediator, pomiędzy okienkami – Observer)



2 Observer

Motto: powiadamianie zainteresowanych o zmianie stanu, dzięki czemu nie odwołują się one do siebie wprost.

Kojarzyć: zdarzenia w C#
Przykład z życia: architektura aplikacji oparta o powiadomienia między różnymi widokami (w środku okienka – Mediator, pomiędzy okienkami – Observer)
Jeszcze inaczej – Observer ujednolica interfejs „Colleagues" Mediatora, dzięki czemu obsługuje dowolną liczbę „Colleagues"



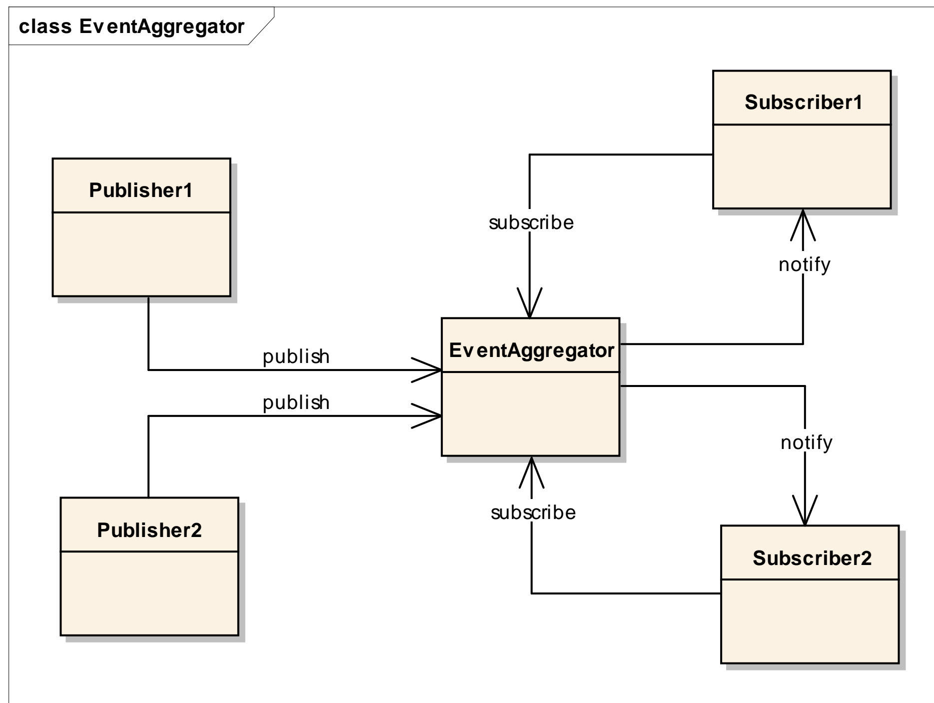
Komentarz: kolejny wzorzec który silnie wpływa na rozwój języków – C#-owe zdarzenia (events) to przykład uczynienia ze wzorca projektowego elementu języka.

3 Event Aggregator

Motto: rozwiąż problem Observera ogólniej – jeden raz dla różnych typów powiadomień
Kojarzyć: ogólniejszy Observer, „hub" komunikacyjny (Observer zaimplementowany jako „słownik list" słuchaczy indeksowany typem powiadomienia)

Event Aggregator znosi najważniejsze ograniczenie Observera – klasy obserwatorów muszą tam zna klasę obserwowanego. W EventAggregatorze zarówno obserwowani jak i obserwujący muszą tylko wiedzieć gdzie szukać EventAggregatora. W efekcie klasy obserwowane i obserwujące mogą być zdefiniowane np. w niezależnych od siebie modułach (co jest niemożliwe w przypadku Observera).

Uwaga: jeden z ważniejszych wzorców **dobrej architektury** aplikacji

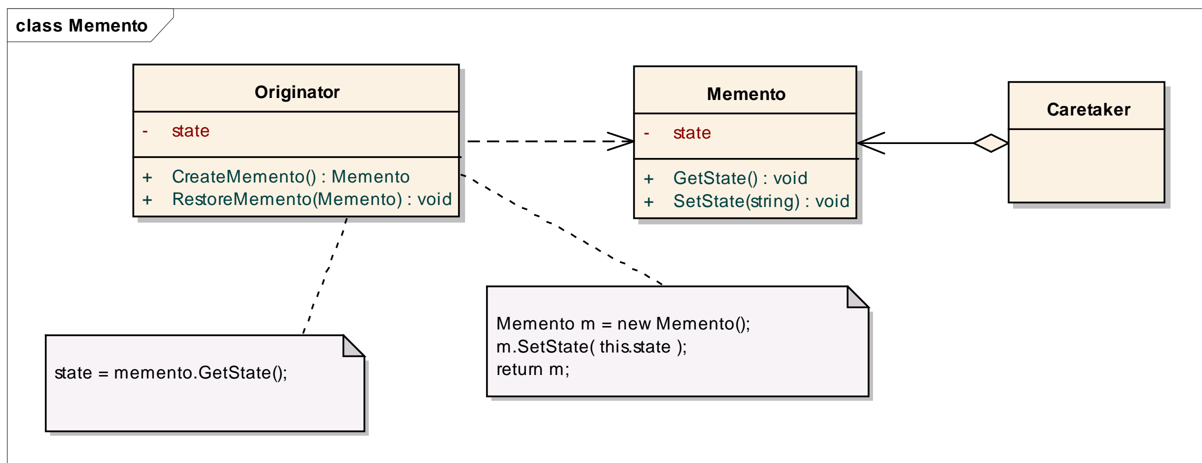



```
namespace Uwr.OOP.BehavioralPatterns.EventAggregator
{
    public interface ISubscriber<T>
    {
        void Handle( T Notification );
    }

    public interface IEventAggregator
    {
        void AddSubscriber<T>( ISubscriber<T> Subscriber );
        void RemoveSubscriber<T>( ISubscriber<T> Subscriber );
        void Publish<T>( T Event );
    }

    public class EventAggregator : IEventAggregator
    {
        Dictionary<Type, List<object>> _subscribers =
            new Dictionary<Type, List<object>>();

        #region IEventAggregator Members

        public void AddSubscriber<T>( ISubscriber<T> Subscriber )
        {
            if (!_subscribers.ContainsKey( typeof( T ) ))
                _subscribers.Add( typeof( T ), new List<object>() );

            _subscribers[typeof( T )].Add( Subscriber );
        }

        public void RemoveSubscriber<T>( ISubscriber<T> Subscriber )
        {
            if (_subscribers.ContainsKey( typeof( T ) ))
                _subscribers[typeof( T )].Remove( Subscriber );
        }
```

```
        public void Publish<T>( T Event )
        {
            if (_subscribers.ContainsKey( typeof( T ) ))
                foreach (ISubscriber<T> subscriber in
                    _subscribers[typeof( T )].OfType<ISubscriber<T>>())
                        subscriber.Handle( Event );
        }

        #endregion
    }
}
```

4 Memento

Motto: Zapamiętuj i pozwalaj odzyskać stan obiektu
Uwaga: stan obiektu i stan pamiątki nie muszą być takie same. W szczególności duże obiekty mogą tworzyć małe, przyrostowe pamiątki
Kojarzyć z: Undo (i opcjonalnym Redo).



W trakcie wykładu zobaczymy jak zbudować obiekt **Memento** i oddzielić od niego odpowiedzialność typu **Caretaker** w której umieścimy funkcjonalność Undo/Redo.

```
namespace Uwr.OOP.BehavioralPatterns.Memento
{
    public class Caretaker
    {
        Stack<Memento> undoStack = new Stack<Memento>();
        Stack<Memento> redoStack = new Stack<Memento>();

        private Originator originator;

        public Caretaker( Originator o )
        {
            this.originator = o;
            this.originator.StateChanged += OriginatorStateChanged;
        }

        public void Undo()
```

```
        {
            if (this.undoStack.Count > 1)
            {
                // bieżący stan na redo
                Memento m = undoStack.Pop();
                redoStack.Push( m );

                Memento ps = undoStack.Peek();
                this.originator.RestoreMemento( ps );
            }
        }

        public void Redo()
        {
            if (this.redoStack.Count > 0)
            {
                Memento m = redoStack.Pop();
                undoStack.Push( m );
                this.originator.RestoreMemento( m );
            }
        }

        public void OriginatorStateChanged()
        {
            redoStack.Clear();

            Memento m = this.originator.CreateMemento();
            undoStack.Push( m );
        }
    }

    public class Originator
    {
        public event Action StateChanged;

        private string _state;
        public string State
        {
            get
            {
                return _state;
            }
            set
            {
                _state = value;

                if (this.StateChanged != null)
                    this.StateChanged();
            }
        }

        public Memento CreateMemento()
        {
            Memento m = new Memento();
            m.State = this.State;
            return m;
        }

        public void RestoreMemento( Memento m )
```

```
        {
            this._state = m.State;
        }
    }

    public class Memento
    {
        public string State { get; set; }
    }
}
```