Projektowanie obiektowe oprogramowania
Wykład 7 – wzorce czynnościowe (2)
Wiktor Zychla 2025

Spis treści

1 Mediator

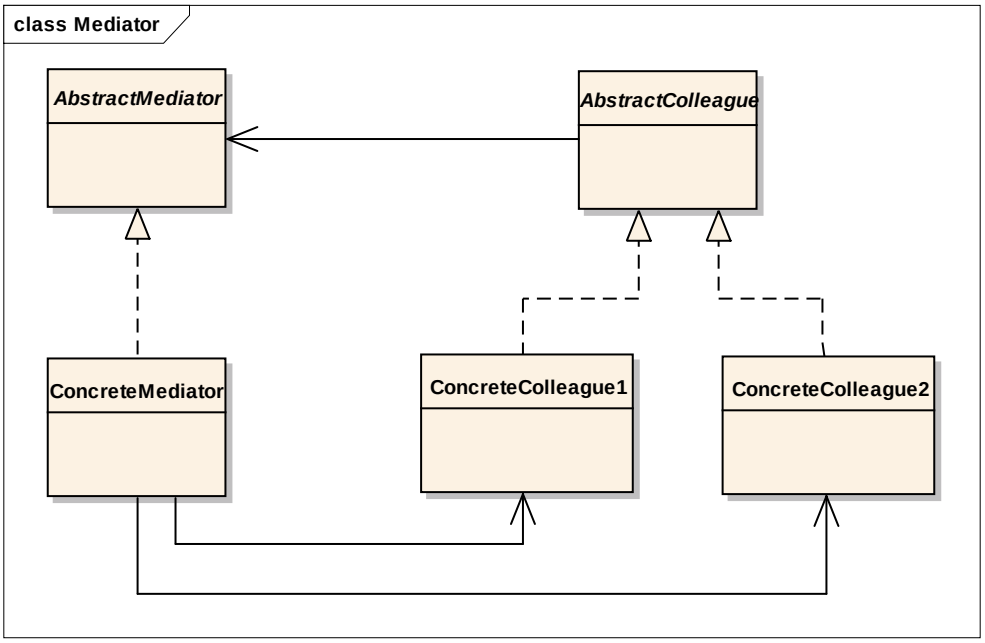
Motto: **Koordynator komunikacji ściśle określonej grupy obiektów** – dzięki niemu one nie odwołują się do siebie wprost (nie muszą nic o sobie wiedzieć), ale przesyłają sobie powiadomienia przez mediatora.

Mediator to uproszczony Observer. Też są „powiadomienia" przesyłane między zbiorem obiektów, ale zbiór współpracujących obiektów jest tu **ściśle** określony - Mediator ma jawne referencje do nich wszystkich.

Mediator może więc wykorzystać ten fakt do wyboru różnych technik przesyłania powiadomień (bezpośrednio, na styl observera itp.).

Kolejna 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 (Form) desktopowych technologii wytwarzania GUI są mediatorami między konkretnymi kontrolkami, które są zagregowane wewnątrz. Zostanie to zaprezentowane na wykładzie.




```
public abstract class AbstractColleague
{
    public Mediator mediator { get; set; }

    public void RaiseEvent( string Message )
    {
        this.mediator.NotifyColleagues( this, Message );
    }
}

public class Colleague1 : AbstractColleague
{
    public string State { get; set; }

    public void Notify1( string Message )
    {
        this.State = Message;
    }
}

public class Colleague2 : AbstractColleague
{
    public string State { get; set; }

    public void Notify2( string Message )
    {
        this.State = Message;
    }
}

public class Mediator
{
    public Colleague1 c1;
    public Colleague2 c2;
```

```
    public void NotifyColleagues( AbstractColleague sender, string Message )
    {
        if ( sender == c1 )
            c2.Notify2( Message );
        else
            c1.Notify1( Message );
    }
}

[TestClass]
public class MediatorTests
{
    [TestMethod]
    public void TestMediator()
    {
        Mediator m = new Mediator();

        Colleague1 c1 = new Colleague1() { mediator = m };
        Colleague2 c2 = new Colleague2() { mediator = m };

        m.c1 = c1; m.c2 = c2;

        c1.RaiseEvent( "foo" );
        Assert.IsTrue( c2.State == "foo" );

        c2.RaiseEvent( "bar" );
        Assert.IsTrue( c1.State == "bar" );
    }
}
```

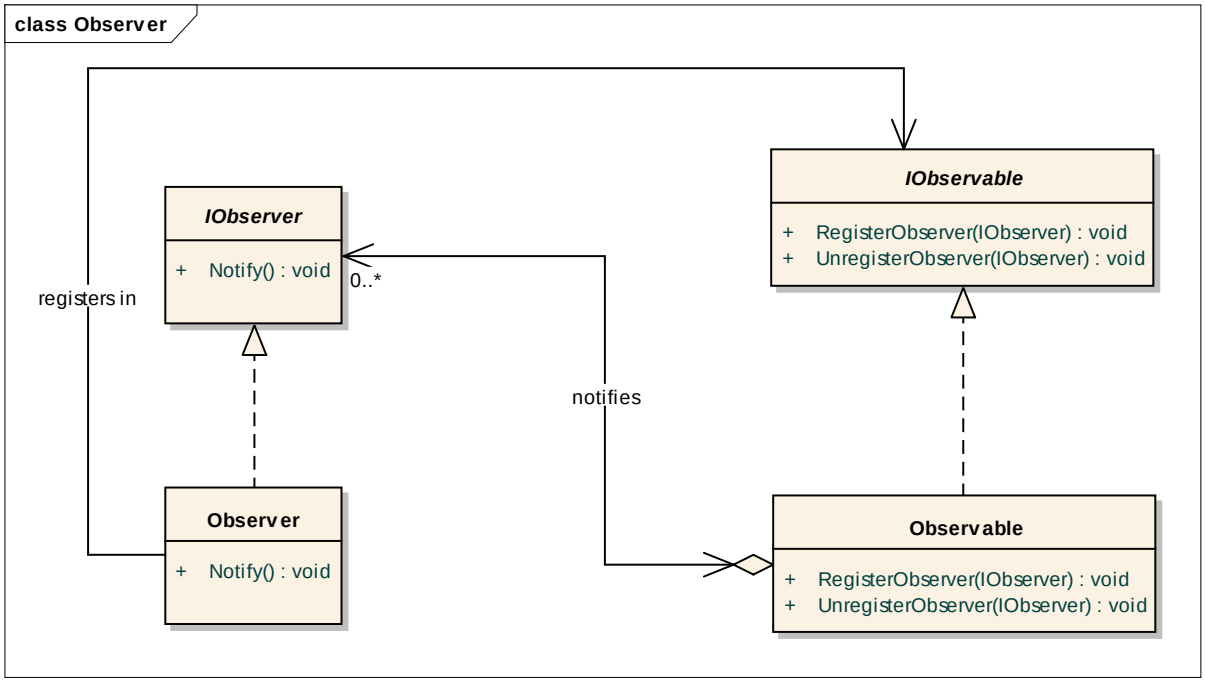
2 Observer

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

Skojarzenie:: zdarzenia w C#

Przykład z życia: architektura aplikacji oparta o powiadomienia między różnymi widokami (w środku okienka – powiadomienia są implementowane przez Mediatora, pomiędzy różnymi okienkami – przez wzorzec Observer)

Jeszcze inaczej – Observer **ujednolica** interfejs „Colleagues" Mediatora, dzięki czemu obsługuje dowolną liczbę „Colleagues" (zamiast referencji wprost tu jest lista)



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

```
public interface IObserver
{
    void Notify(int state);
}

public interface IObservable
{
    void RegisterObserver( IObserver observer );
    void UnregisterObserver( IObserver observer );
}

public class Observer : IObserver
{
    public int State;
    public void Notify( int state )
    {
        this.State = state;
    }
}

public class Observable : IObservable
{
    private List<IObserver> _observers = new List<IObserver>();

    public void RegisterObserver( IObserver observer )
    {
        this._observers.Add( observer );
    }

    public void UnregisterObserver( IObserver observer )
    {
        this._observers.Remove( observer );
```

```
    }

    public void NotifyObservers( int state )
    {
        foreach ( var observer in this._observers )
        {
            observer.Notify( state );
        }
    }
}

[TestClass]
public class ObserverTests
{
    [TestMethod]
    public void TestObserver()
    {
        Observable observable = new Observable();

        Observer o1 = new Observer();
        Observer o2 = new Observer();

        // o1 i o2 obserwują
        observable.RegisterObserver( o1 );
        observable.RegisterObserver( o2 );

        observable.NotifyObservers( 17 );

        Assert.IsTrue( o1.State == 17 );
        Assert.IsTrue( o2.State == 17 );

        // o2 już nie obserwuje
        observable.UnregisterObserver( o2 );

        observable.NotifyObservers( 19 );

        Assert.IsTrue( o1.State == 19 );
        Assert.IsTrue( o2.State == 17 );

    }
}
```

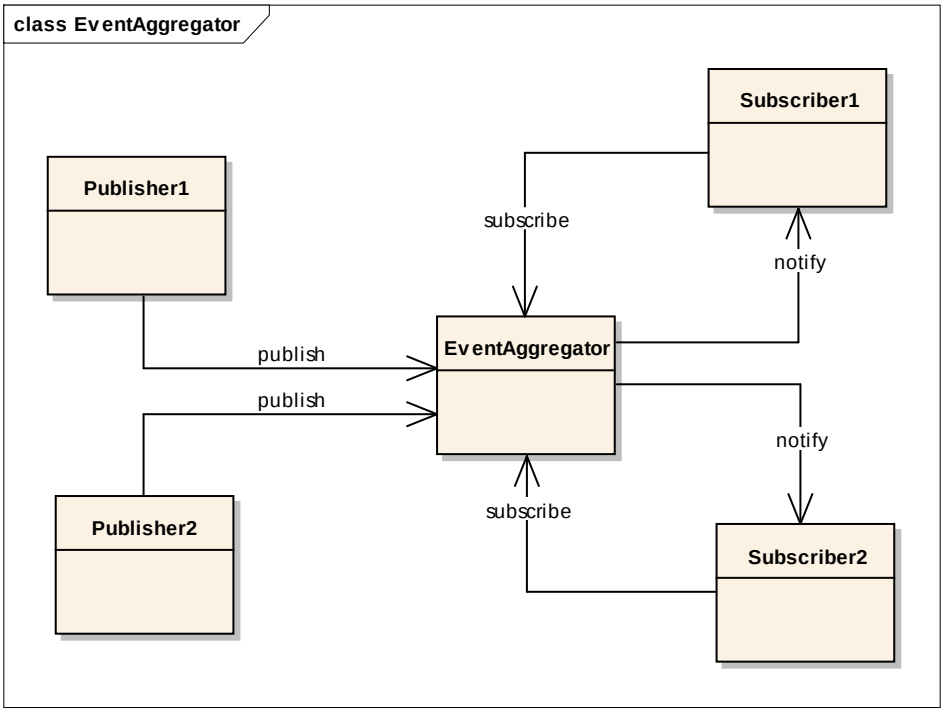
3 Event Aggregator

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 znać klasę obserwowanego. W EventAggregatorze zarówno obserwowani jak i obserwujący muszą tylko mieć referencję do EventAggregatora. W efekcie klasy obserwowane i obserwujące mogą być zdefiniowane np. w niezależnych od siebie modułach (co jest trudniejsze 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 );
```

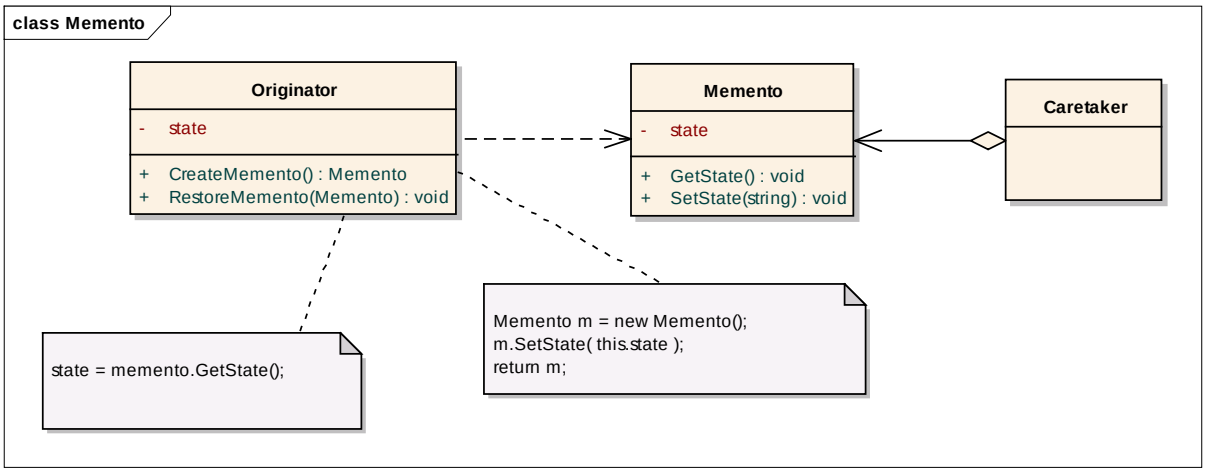
```
        }

        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 odzyskuj stan obiektu**

Uwaga: stan obiektu i stan pamiątki nie muszą być takie same. W szczególności duże obiekty mogą tworzyć małe, przyrostowe pamiątki (pamiątka pamięta wtedy *deltę* między poprzednim a następnym stanem a nie cały stan)

Samo Memento nie jest szczególnie pasjonujące, interesująco robi się dopiero wtedy kiedy próbuje się zaimplementować mechanizm Undo/Redo za pomocą list Memento.



Implementacja referencyjna (zgodna z diagramem)

```
/// <summary>
/// Klasa która potrafi zachowywać/odtwarzać stan
/// </summary>
public class Originator
{
    public Originator()
    {
    }
```

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

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

    public void RestoreMemento( Memento memento )
    {
        this.State = memento.GetState();
    }

}

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

    public Memento()
    {

    }

    public string GetState()
    {
        return this.State;
    }

    public void SetState( string State )
    {
        this.State = State;
    }
}

[TestClass]
public class SimpleMememtoTests
{
    [TestMethod]
    public void TestMemento()
    {
        Originator o = new Originator();

        o.State = "foo";
        Assert.IsTrue( o.State == "foo" );

        Memento m1 = o.CreateMemento();

        o.State = "bar";
        Assert.IsTrue( o.State == "bar" );

        Memento m2 = o.CreateMemento();

        o.State = "qux";
        Assert.IsTrue( o.State == "qux" );

        o.RestoreMemento( m2 );
        Assert.IsTrue( o.State == "bar" );
```

```
        o.RestoreMemento( m1 );
        Assert.IsTrue( o.State == "foo" );
    }
}
```

W trakcie wykładu zobaczymy jak dodać dodatkową funkcjonalność Undo/Redo, nieuwzględnianą na diagramie strukturalnym. Ta funkcjonalność jest najciekawszym elementem związanym ze wzorcem **Memento** – implementacja Undo/Redo w opisany niżej sposób jest na tyle standardowa, że można jej użyć jako wzorca wszędzie tam gdzie pojawia się oczekiwanie Undo/Redo.

Tym razem implementację warto zacząć od testów – jakie warunki powinno spełniać Undo/Redo?

- Każda zmiana stanu powinna odłożyć się w historii
- Undo powinno być możliwe tak bardzo w przeszłość jak zapamiętana „historia”
- Redo powinno być możliwe tak bardzo w przyszłość jak zapamiętana „przyszłość”
- Jeżeli po Undo nastąpi zmiana stanu wymuszona z zewnątrz, to należy uciąć „przyszłość” do której można przywrócić

```
public class Originator
{
    public Originator()
    {
    }

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

            undoStates.Push( this.CreateMemento() );
            redoStates.Clear();
        }
    }

    public Memento CreateMemento()
    {
        Memento memento = new Memento();
        memento.SetState( this._state );
        return memento;
    }

    public void RestoreMemento( Memento memento )
    {
        this._state = memento.GetState();
    }

    // część implementacji odpowiedzialna za Undo/Redo

    // górny element stosu undo to bieżący stan
    // "poprzedni" stan leży pod bieżącym
    Stack<Memento> undoStates = new Stack<Memento>();
```

```
    Stack<Memento> redoStates = new Stack<Memento>();

    public void Undo()
    {
        if ( undoStates.Count > 1 )
        {
            Memento currentState = undoStates.Pop();
            redoStates.Push( currentState );

            Memento previousState = undoStates.Peek();
            this.RestoreMemento( previousState );
        }
    }

    public void Redo()
    {
        if ( redoStates.Count > 0 )
        {
            Memento futureState = redoStates.Pop();
            undoStates.Push( futureState );

            this.RestoreMemento( futureState );
        }
    }
}

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

    public Memento()
    {

    }

    public string GetState()
    {
        return this.State;
    }

    public void SetState( string State )
    {
        this.State = State;
    }
}

[TestClass]
public class Client
{
    [TestMethod]
    public void UndoRedoTestMemento()
    {
        Originator o = new Originator();

        // odkładanie historii
        o.State = "foo";
        Assert.IsTrue( o.State == "foo" );

        o.State = "bar";
        Assert.IsTrue( o.State == "bar" );
```

```
        o.State = "qux";
        Assert.IsTrue( o.State == "qux" );

        // cofanie historii (Undo)
        o.Undo();
        Assert.IsTrue( o.State == "bar" );
        o.Undo();
        Assert.IsTrue( o.State == "foo" );
        o.Undo();
        Assert.IsTrue( o.State == "foo" ); // nie można za daleko!

        // przywracanie historii
        o.Redo();
        Assert.IsTrue( o.State == "bar" );
        o.Redo();
        Assert.IsTrue( o.State == "qux" );
        o.Redo();
        Assert.IsTrue( o.State == "qux" ); // nie można za daleko!

        // undo ale potem ręczna zmiana stanu
        // powinna uciąć "przyszłość"
        o.Undo();
        Assert.IsTrue( o.State == "bar" );
        o.State = "baz";
        o.Redo();
        Assert.IsTrue( o.State == "baz" );
        o.Undo();
        Assert.IsTrue( o.State == "bar" );
    }
}
```