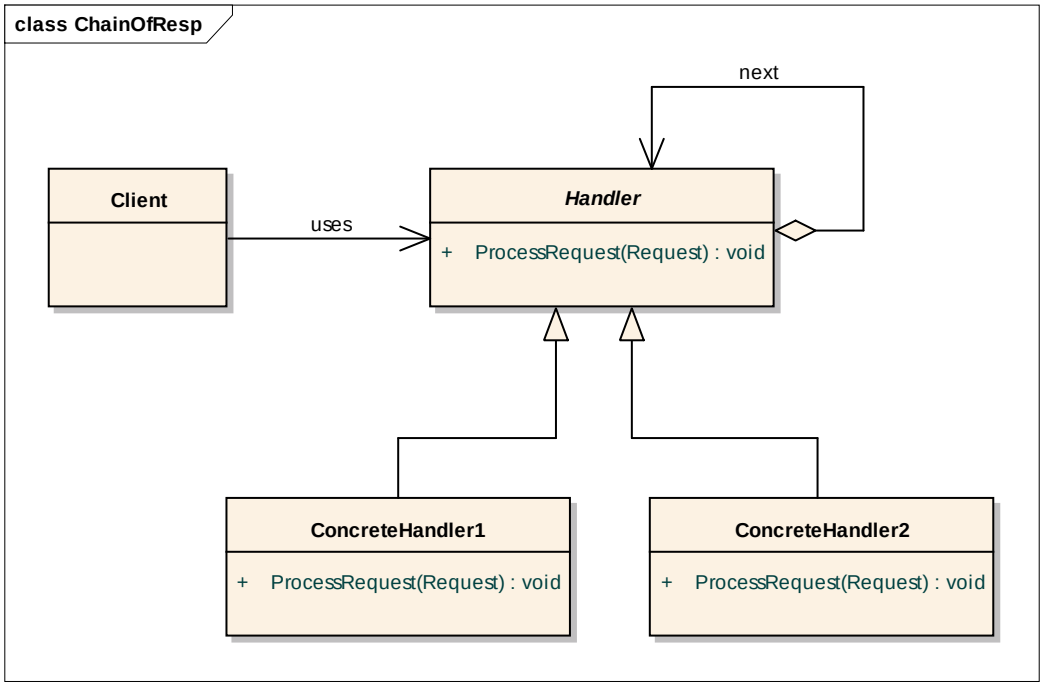
Projektowanie obiektowe oprogramowania
Wykład 8 – wzorce czynnościowe (3)
Wiktor Zychla 2024

1 Chain of Responsibility

Motto: uniknięcie związania obiektu wysyłającego żądanie z konkretnym odbiorcą w sytuacji gdy zbiór możliwych odbiorców jest dynamiczny
Kojarzyć: łańcuch niezależnych odbiorców wiadomości; przetwarzanie w skomplikowanej logice



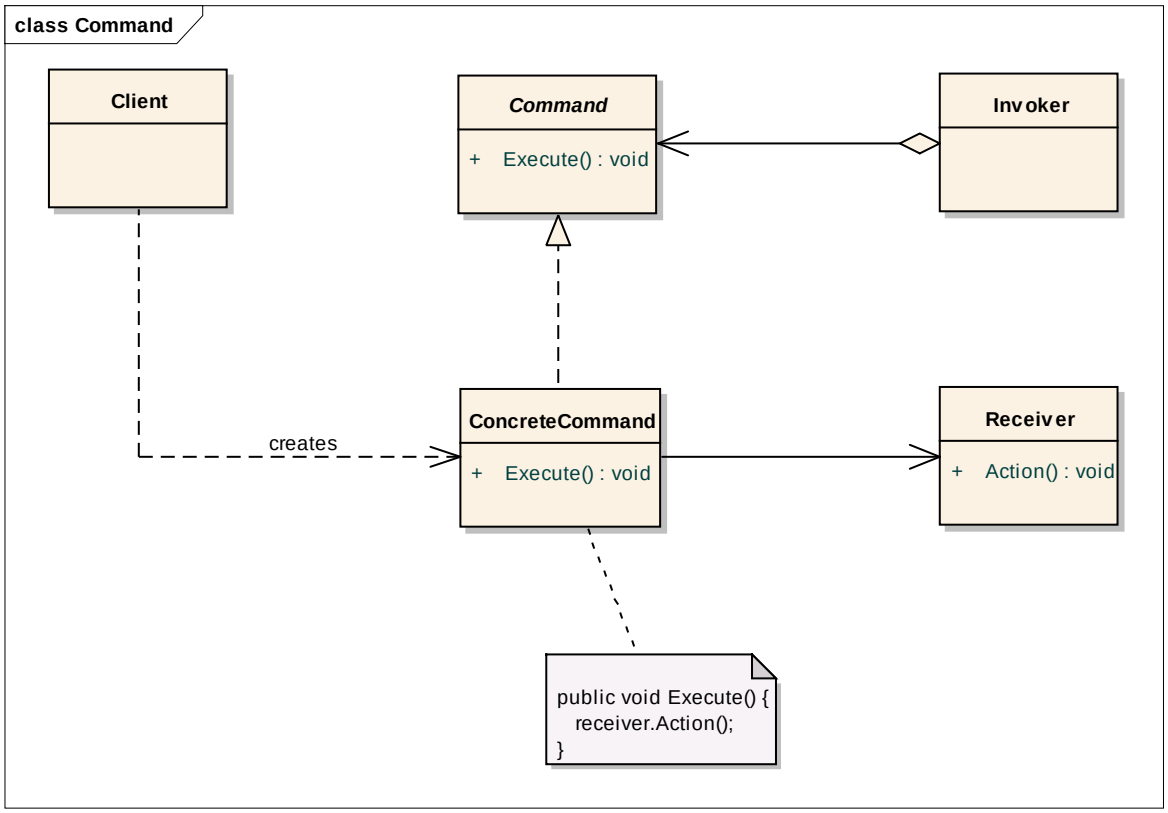

```
public class Request
{
    public int Temperature { get; set; }
}

public abstract class AbstractHandler
{
    protected AbstractHandler _nextHandler;

    public abstract bool ProcessRequest( Request request );

    public void DispatchRequest( Request request )
    {
        bool result = this.ProcessRequest( request );

        if ( !result && _nextHandler != null )
```

```
            _nextHandler.ProcessRequest( request );
    }

    /// <summary>
    /// Dokleja na koniec łańcucha
    /// </summary>
    /// <param name="Handler"></param>
    public void AttachHandler( AbstractHandler Handler )
    {
        if ( _nextHandler != null )
            _nextHandler.AttachHandler( Handler );
        else
            _nextHandler = Handler;
    }
}

public class LessThanZeroHandler : AbstractHandler
{
    public override bool ProcessRequest( Request request )
    {
        if ( request.Temperature < 0 )
        {
            HandlerMonitor.LTZ += 1;

            return true;
        }

        return false;
    }
}

public class GeqZeroHandler : AbstractHandler
{
    public override bool ProcessRequest( Request request )
    {
        if ( request.Temperature >= 0 )
        {
            HandlerMonitor.GEQZ += 1;

            return true;
        }

        return false;
    }
}

public class Client
{
    public void TestChain()
    {
        AbstractHandler sink = new LessThanZeroHandler();
        sink.AttachHandler( new GeqZeroHandler() );

        HandlerMonitor.Reset();
        Request r1 = new Request() { Temperature = -1 };
        sink.DispatchRequest( r1 );

        //Assert.That( HandlerMonitor.LTZ == 1 );
        //Assert.That( HandlerMonitor.GEQZ == 0 );
```

```
        HandlerMonitor.Reset();
        Request r2 = new Request() { Temperature = 1 };
        sink.DispatchRequest( r2 );

        //Assert.That( HandlerMonitor.LTZ == 0 );
        //Assert.That( HandlerMonitor.GEQZ == 1 );

    }
}

public class HandlerMonitor
{
    public static int LTZ;
    public static int GEQZ;

    public static void Reset()
    {
        LTZ = 0;
        GEQZ = 0;
    }
}
```

2 Command

Motto: kapsułkowanie żądań w postaci obiektów o jednolitym interfejsie dostępu, oddziela wywołującego (Invoker) od odbiorcy (Receiver). Wiele różnych klas Receiver może mieć różny interfejs, ale dzięki Command dla Invokera wyglądają one tak samo. Można powiedzieć, że Command wciela w życie filozofię Adaptera na szeroką skalę – każdy ConcreteCommand jest adapterem jakiegoś Receiver do jednolitego interfejsu Command.
Uwaga: prostszą alternatywą byłoby zapamiętanie funkcji zwrotnej, ale komenda może mieć szerszy interfejs niż tylko Execute (), np. Undo() - o ile sam Receiver może nie implementować Undo wprost, o tyle ConcreteCommand może użyć innego sposobu implementacji Undo (zwykle – z pomocą innego obiektu).



Typowe niedostatki implementacyjne:

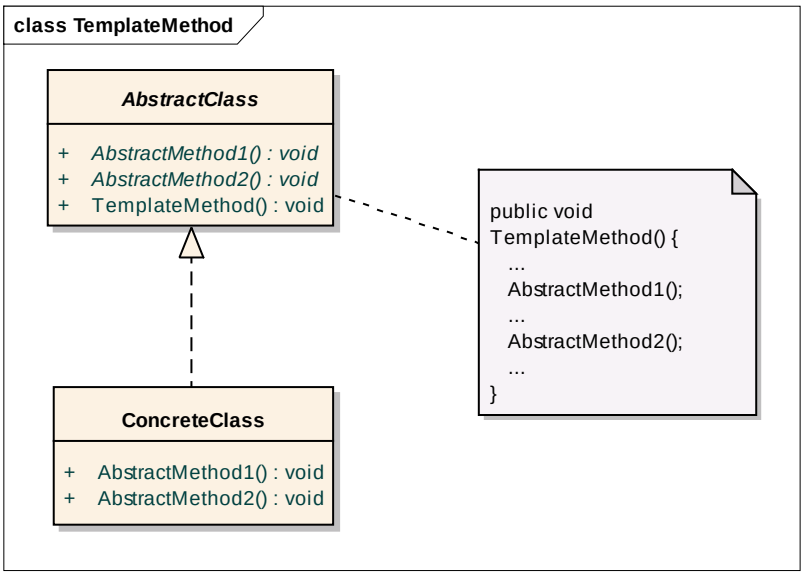
- Konkretne polecenie nie deleguje wykonania do odbiorcy, tylko całą logikę implementuje samo
- Konkretne polecenie nie zawiera żadnego stanu, jest tylko delegatorem wywołania do odbiorcy (czy to jest problem ?)

3 Template Method

Motto: określ szkielet algorytmu i zrzucając odpowiedzialność za implementację szczegółów do podklasy.

Kojarzyć z: Template Method = Strategy przez dziedziczenie

Refaktoryzacja Template Method do Strategy polega na zamianie klasy abstrakcyjnej na interfejs i wyłączeniu TemplateMethod do osobnej klasy do której wstrzykiwana jest implementacja interfejsu.

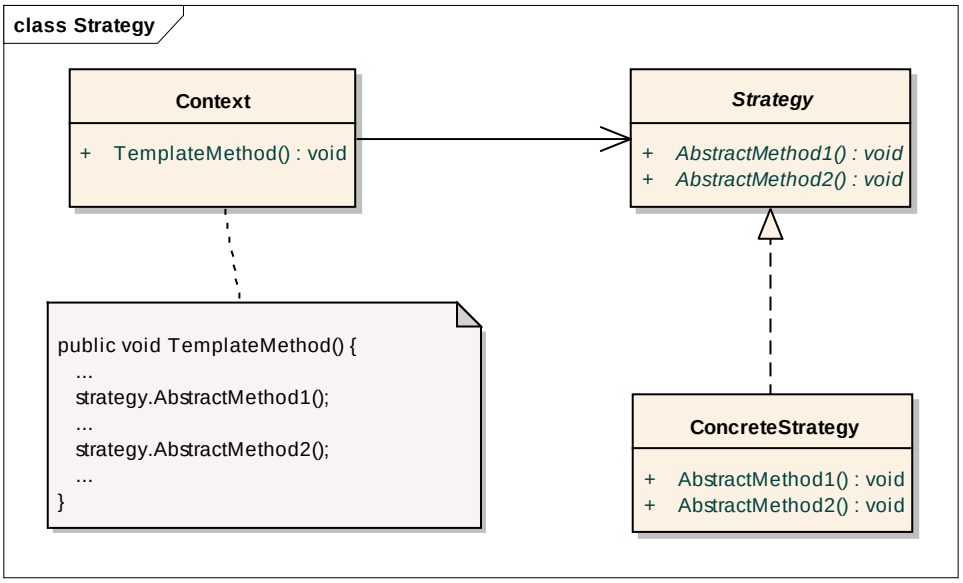


4 Strategy

Motto: określ szkielet algorytmu i zrzucając odpowiedzialność za implementację szczegółów do klasy, do której delegujesz żądania.

Kojarzyć z: Strategy = Template Method przez delegację

Refaktoryzacja Strategy do Template Method polega na likwidacji wstrzykiwania – metoda szablonowa staje się częścią klasy abstrakcyjnej, a konkretną funkcjonalność uzyskuje się przez dziedziczenie.



Template Method vs Strategy

- Template Method wiąże algorytm z konkretną hierarchią klas, Strategy jest implementowane przez interfejs + delegację – można próbować argumentować że Strategy jest lepszym wyborem („*prefer compostion over inheritance*"). Z drugiej strony klasa strategii nie ma zwykle większego sensu poza kontekstem w którym jest używana, a jej struktura jest narzucona przez interfejs, jeśli więc ma wykonać jakąś nietrywialną pracę to to ona *deleguje* tę pracę dalej. W praktyce więc te wzorce
- Template Method może oznaczać mniej kodu, bo klasa przeciąży tylko metody które chce uszczegółowić

5 State

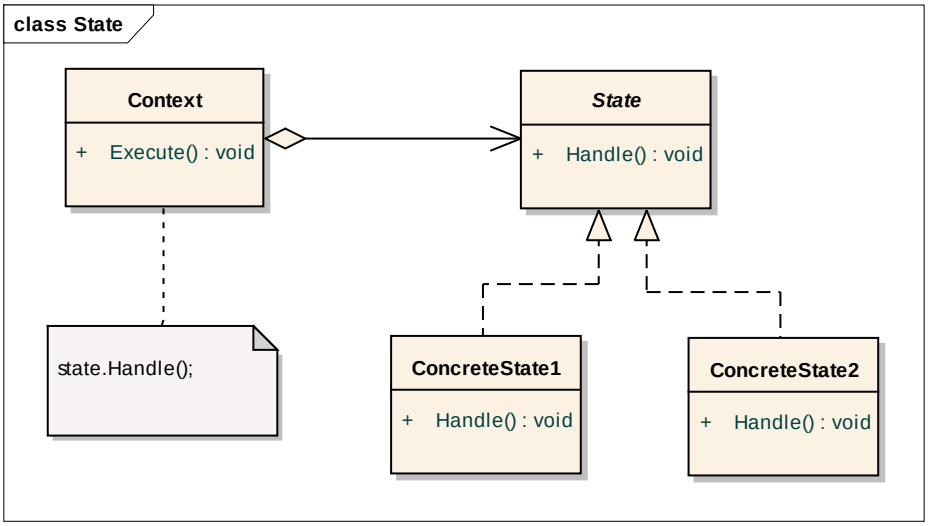
Motto: maszyna stanowa
Uwaga: zbyt rzadko stosowana (można częściej niż się wydaje)
Przykład zastosowania: zarządzanie złożonym GUI. Formularz jest kontekstem, a klasy stanów odpowiadają aktualnie wyświetlanej zawartości. W każdym ze stanów inaczej obsługuje się zestaw zdarzeń formantów.

Przykład:
Zdarzenia:

- klik na drzewie,
- klik na liście,
- dwuklik na liście,
- kliknięcie prawym przyciskiem na liście

Stany

- wyświetlanie listy osób,
- wyświetlanie listy placówek
- wyświetlanie zawartości innych słowników
- itd.



Maszyna stanowa zaimplementowana „naiwnie”, stan przechowywany w zmiennej:

```
public class Taxi
{


    public const int FREE = 1;
    public const int OCCUPIED = 2;

    private int _state = FREE;

    public string Destionation;
    public bool HasArrived = false;

    public void Call()
    {
        if ( _state == FREE )
```

```
        {
            _state = OCCUPIED;
            HasArrived = false;
        }
        else
        if ( _state == OCCUPIED )
        {
            throw new Exception();
        }
    }

    public void Drive( string Destination )
    {
        if ( _state == FREE )
        {
            throw new Exception();
        }
        else
            if ( _state == OCCUPIED )
            {
                this.Destionation = Destination;
            }

    }

    public void EndTrip()
    {
        if ( _state == FREE )
        {
            throw new Exception();
        }
        else
            if ( _state == OCCUPIED )
            {
                _state = FREE;
                HasArrived = true;
            }
    }

}

[TestFixture]
public class StateTester
{
    [Test]
    public void TestState()
    {
        Taxi taxi = new Taxi();

        taxi.Call();

        Assert.That( taxi.HasArrived == false );

        taxi.Drive( "wroclaw" );
        taxi.EndTrip();

        Assert.That( taxi.Destionation == "wroclaw" );
        Assert.That( taxi.HasArrived == true );
    }
}
```

Maszyna stanowa zaimplementowana przy pomocy wzorca State:

```
public abstract class TaxiState
{
    protected Taxi Taxi { get; set; }

    public abstract void Call();
    public abstract void Drive( string Destination );
    public abstract void EndTrip();
}

public class TaxiFreeState : TaxiState
{
    public TaxiFreeState( Taxi taxi ) { this.Taxi = taxi; }

    public override void Call()
    {
        Taxi.HasArrived = false;
        Taxi.SetState( new OccupiedTaxiState( this.Taxi ) );
    }

    public override void Drive( string Destination )
    {
        throw new Exception();
    }

    public override void EndTrip()
    {
        throw new Exception();
    }
}

public class OccupiedTaxiState : TaxiState
{
    public OccupiedTaxiState( Taxi taxi ) { this.Taxi = taxi; }

    public override void Call()
    {
        throw new Exception();
    }

    public override void Drive( string Destination )
    {
        this.Taxi.Destionation = Destination;
    }

    public override void EndTrip()
    {
        this.Taxi.HasArrived = true;
        this.Taxi.SetState( new TaxiFreeState( this.Taxi ) );
    }
}
```

```
public class Taxi
{

    public Taxi()
    {
        this._state = new TaxiFreeState(this);
    }

    private TaxiState _state;

    public void SetState( TaxiState newState )
    {
        this._state = newState;
    }

    public string Destionation;
    public bool HasArrived = false;

    public void Call()
    {
        _state.Call();
    }

    public void Drive( string Destination )
    {
        _state.Drive( Destination );
    }

    public void EndTrip()
    {
        _state.EndTrip();
    }

}

[TestFixture]
public class StateTester
{
    [Test]
    public void TestState()
    {
        Taxi taxi = new Taxi();

        taxi.Call();

        Assert.That( taxi.HasArrived == false );

        taxi.Drive( "wroclaw" );
        taxi.EndTrip();

        Assert.That( taxi.Destionation == "wroclaw" );
        Assert.That( taxi.HasArrived == true );
    }
}
```