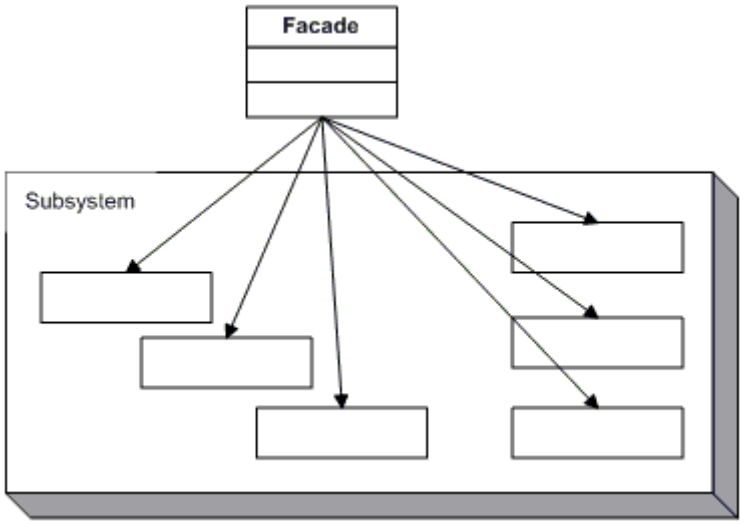
Projektowanie obiektowe oprogramowania
Wykład 5 – wzorce strukturalne
Wiktor Zychla 2025

Spis treści

1 Facade

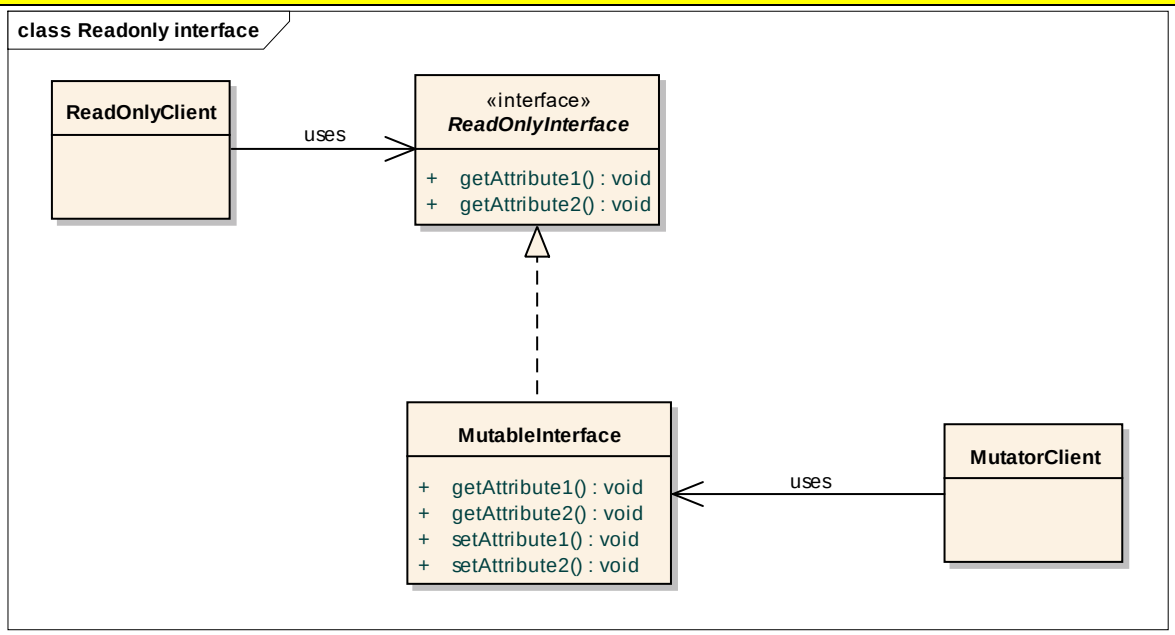
Motto: uproszczony interfejs dla podsystemu z wieloma interfejsami




```
class SmtpFacade {
      public void Send( string From, string To,
                        string Subject, string Body,
                        Stream Attachment, string AttachmentMimeType );
}
```

2 Read-only interface

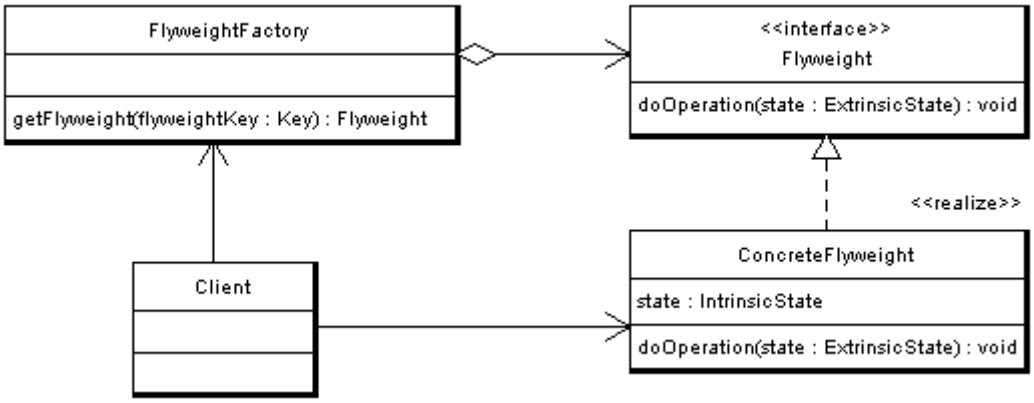
Motto: interfejs do odczytu dla wszystkich, a do zapisu tylko dla wybranych



Przykład: ReadOnlyCollection, AsReadOnly()

3 Flyweight

Motto: Efektywne zarządzanie wieloma drobnymi obiektami
Kojarzyć: Object Pool + immutable + bardzo dużo danych – zapamiętać przykład z wykładu Board vs Checker



Flyweight w implementacji przypomina Object Pool. Różnica jest taka, że Pool utrzymuje pulę **rozłącznych** obiektów, zwraca za każdym razem inną instancję, a po użyciu instancja ta wraca do puli. Motywacją dla Object Pool jest bardzo kosztowne tworzenie instancji.

W przypadku Flyweight jest inaczej – motywacją dla tego wzorca jest bowiem chęć oszczędzania pamięci w sytuacji, gdy do utworzenia jest naprawdę duża liczba potencjalnych obiektów, **zbyt** duża żeby reprezentować je w sposób klasyczny. FlyweightFactory utrzymuje więc dużo mniejszą pulę obiektów – taką, w której wyznacznikiem tożsamości obiektu jest stan, który można współdzielić między „rzekomo" różnymi instancjami. Klient dostaje wiele obiektów, a w rzeczywistości za każdym razem kiedy prosi o obiekt o tym samym „kluczu", dostaje tę samą instancję.

Takie podejście rodzi oczywiście problem identyfikacji takiej części stanu obiektu która może być współdzielona (*intrinsic*) i takiej która nie może być współdzielona (*extrinsic*) i jej przechowaniem w optymalny dla pamięci sposób zajmuje się albo **FlyweightFactory**.

Na przykład (przykład z wykładu) wyobraźmy sobie planszę do gry w warcaby o wymiarach 1mln na 1mln pól. Gdyby taka plansza była zaimplementowana tak, żeby utrzymywać instancje obiektów bierek dla każdego pola planszy, zużycie pamięci byłoby ogromne.

Zamiast tego, w klasie planszy (**Board**) reprezentacja stanu planszy jest jakaś optymalna (na przykład za pomocą tablicy bajtów). Ale klient nadal potrzebuje obiektu – bierki (**Piece)**, chociażby po to żeby wywołać na nim metody do rysowania, z odpowiednią teksturą.

Tu wkracza wzorzec Flyweight.

Z punktu widzenia silnika graficznego, bierki są tak naprawdę dwie różne – biała i czarna. FlyweightFactory (tu: **Board**) poproszony o wykonstruowanie nowej instancji (tu: **Piece**) zwróci więc co prawda za każdym razem nową instancję ale w zależności od rodzaju bierki (biała lub czarna) taka nowa instancja ma tylko tę współdzieloną (*intrinsic*) część stanu (tu: kolor czy teksturę).

Reszta stanu (*extrinsic*), czyli np. pozycja X/Y na planszy (bo przecież każda bierka ma swoje położenie), przychodzi „z zewnątrz", w parametrach wywołania konkretnej metody (tu: metody

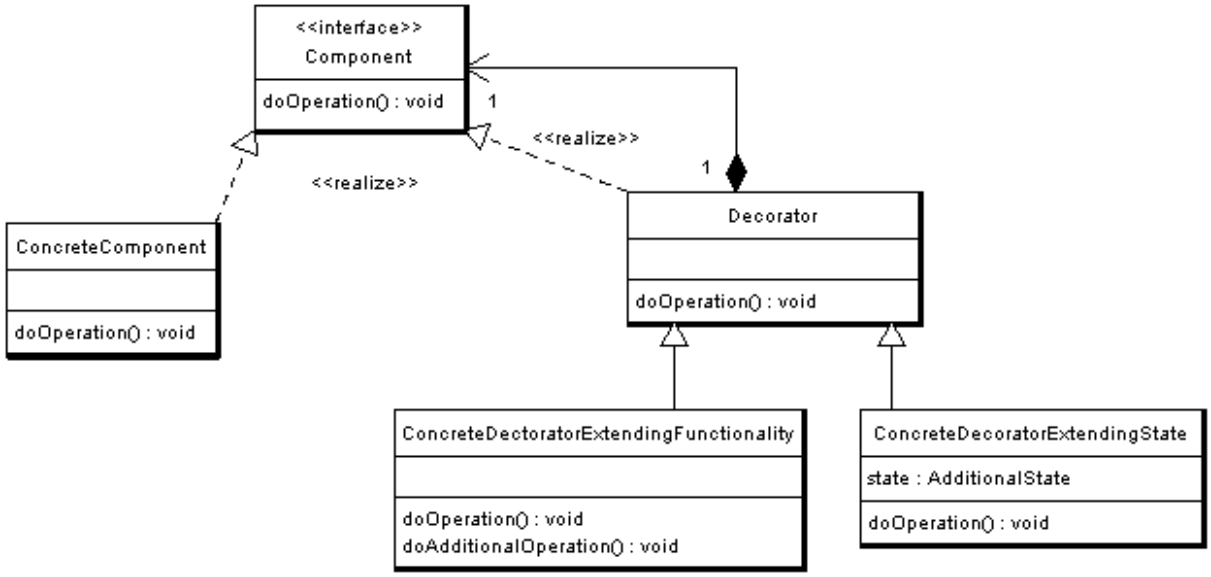
rysowania bierki). Metoda do rysowania bierki będzie więc miała parametr do przekazania tego zewnętrznego, niewspółdzielonego stanu.

```
namespace Flyweight
{
    internal class Program
    {
        static void Main( string[] args )
        {
            var board = new Board();

            for ( int x = 0; x < 5; x++ )
            {
                for ( int y = 0; y < 5; y++ )
                {
                    // niby za każdym razem nowa instancja
                    // ale tak naprawdę tylko dwie różne
                    // ale klient tego nie musi wiedzieć
                    var piece = board.GetPiece( x, y );

                    piece.Draw( 100 * x, 100 * y );
                }
            }

            Console.ReadLine();

        }
    }

    // tu: FlyweightFactory
    public class Board
    {
        private Dictionary<PieceColor, Piece> _pieces =
            new Dictionary<PieceColor, Piece>();

        public Piece GetPiece( int x, int y )
        {
            // w zależności od pozycji na planszy ale tak naprawdę tylko dwie
 różne
            PieceColor key =
                (x + y) % 2 == 0
                    ? PieceColor.White
                    : PieceColor.Black;

            if ( !_pieces.ContainsKey( key ) )
            {
                _pieces.Add( key, new Piece() { Color = key } );
            }

            return _pieces[key];
        }
    }

    // tu: Flyweight
    public class Piece
    {
        // Color tu jest w intrinsic state
        public PieceColor Color { get; set; }
```

```
        // ScreenX/ScreenY są w extrinsic state
        public void Draw( int ScreenX, int ScreenY )
        {
            Console.WriteLine( $"bierka {Color} rysowana w {ScreenX} {ScreenY
}" );
        }
    }

    // tu: FlyweightKey
    public enum PieceColor
    {
        White,
        Black
    }
}
```

4 Decorator

Motto: Dynamicznie rozszerzanie odpowiedzialności obiektów (alternatywa dla podklas)
Kojarzyć: w bibliotekach standardowych technologii przemysłowych od lat używa się wzorca Decorator do implementacji **podsystemu strumieni**




```
namespace Decorator
{
    public interface IDrink
    {
        string Name { get; }
        Decimal Cost { get; }
    }

    public class Coffee : IDrink
    {
        public string Name
        {
            get { return "coffee"; }
        }

        public decimal Cost
        {
            get { return 2m; }
        }
    }

    public class Tea : IDrink
    {

        public string Name
        {
            get { return "tea"; }
        }

        public decimal Cost
        {
            get { return 1m; }
```

```
        }
    }

    public class SugarDecorator : IDrink
    {
        private IDrink _drink;
        public SugarDecorator( IDrink drink )
        {
            this._drink = drink;
        }

        public string Name
        {
            get { return _drink.Name + " with sugar"; }
        }

        public decimal Cost
        {
            get { return _drink.Cost + 1; }
        }
    }

    [TestClass]
    public class DecoratorTests
    {
        [TestMethod]
        public void TestDecorators()
        {
            IDrink tea = new Tea();

            IDrink stea = new SugarDecorator( tea );
            IDrink s2tea = new SugarDecorator( stea );

            Assert.AreEqual( 1, tea.Cost );
            Assert.AreEqual( 3, s2tea.Cost );
        }
    }
}
```

5 Proxy

Motto: substytut (zamiennik) obiektu w celu sterowania dostępem do niego

- Proxy zdalne – reprezentant lokalny obiektu zdalnego
- Proxy wirtualne – tworzy kosztowny obiekt na żądanie
- Proxy ochraniające – kontroluje dostęp do obiektu
- Proxy logujące – loguje dostęp do obiektu

Przykład virtual proxy z biblioteki standardowej .NET - Lazy<T>.

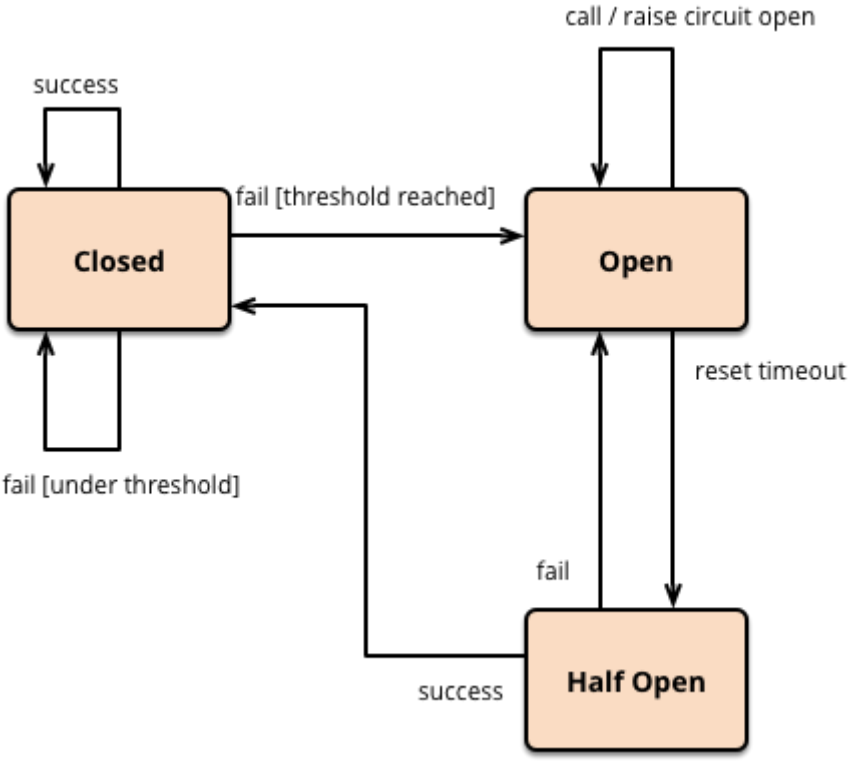
Uwaga! Struktura Proxy i Decoratora może wydawać się podobna. Różnice są następujące:

- Każdy z Decoratorów może dodawać nowe, specyficzne dla siebie operacje/informacje; Proxy nigdy nie zmienia (nie rozszerza) interfejsu obiektu
- Proxy z zasady nie jest przeznaczone do stosowania „rekursywnego”; Dekorator przeciwnie

```
namespace Proxy
{
    public interface IFoo
    {
        int Bar { get; }
    }

    public class Foo : IFoo
    {
        public Foo( int v )
        {
            this.Bar = v;
        }

        public int Bar { get; private set; }
    }

    public class FooVirtualProxy : IFoo
    {
        int v;

        public FooVirtualProxy( int v )
        {
            this.v = v;
        }

        private Foo _foo;
        public int Bar
        {
            get
            {
                if ( _foo == null )
                    _foo = new Foo( this.v );
```

```
                return _foo.Bar;
            }
        }
    }

    [TestClass]
    public class ProxyTests
    {
        [TestMethod]
        public void TestProxies()
        {
            IFoo foo = new Foo(5);
            Assert.AreEqual( 5, foo.Bar );

            IFoo foo2 = new FooVirtualProxy( 5 );
            Assert.AreEqual( 5, foo2.Bar );

            Lazy<IFoo> foo3 = new Lazy<IFoo>( () => new Foo( 5 ) );
            Assert.AreEqual( 5, foo3.Value.Bar );
        }
    }
}
```

Szczególnym Proxy, zasługującym na osobną uwagę jest tzw. **Circuit Breaker**



Rysunek 1 (za https://martinfowler.com/bliki/CircuitBreaker.html)

Circuit Beaker kontroluje wywołania zdalnego obiektu i w sytuacji w której następuje przekroczenie zdefiniowanej liczby błędów wywołania, „podnosi bezpiecznik", przechodzi do stanu w którym wywołania są obsługiwane lokalnie, z pominięciem zdalnego obiektu.

Po określonym czasie następuje próba ponowienia wywołania operacji na docelowym obiekcie i w zależności od wyniku ponowienia, bezpiecznik pozostaje podniesiony lub zamyka się.

Wzorzec Circuit Breaker jest na tyle trudny do implementacji że nie da się go szybko napisać samodzielnie. Rekomenduje się użycie istniejących bibliotek, np.

- Dla .NET: Polly Circuit Breaker
- Dla Java: resilience4j

Tego typu biblioteki posiadają zwykle kilka rodzajów proxy:

- Proxy typu **Retry** – proxy z polityką automatycznego ponawiania kolejnych nieudanych wywołań
- Proxy typu **Circuit Breaker** – czyli wykrywanie pewnej liczby nieudanych wywołań i blokowanie kolejnych nieudanych wywołań
- Proxy typu **Rate Limiter –** czyli kontrolowanie liczby wywołań w czasie
- Proxy typu **Fallback** – czyli obsługa wariantu awaryjnego w sytuacji gdy oryginalne wywołanie nie udaje się
- Proxy typu **Hedging –** czyli automatyczne przełączenie na wskazany wariant zapasowy gdy oryginalne wywołanie trwa zbyt długo

Przykład użycia biblioteki Polly i dwóch rodzajów proxy:

- Proxy typu Retry – **WaitAndRetryAsync** (w kodzie niżej)
- Proxy typu CircuitBreaker - **CircuitBreakerAsync**

Usługa sieciowa:

```
using System;
using System.Threading.Tasks;
using System.Web.Http;
using System.Web.Http.SelfHost;

namespace WebApiService
{
    internal class Program
    {
        static void Main( string[] args )
        {
            var config = new HttpSelfHostConfiguration("http://localhost:8087
");

            config.Routes.MapHttpRoute(
                "API Default", "api/{controller}/{id}",
                new { id = RouteParameter.Optional } );

            using ( HttpSelfHostServer server = new HttpSelfHostServer( confi
g ) )
```

```
            {
                server.OpenAsync().Wait();
                Console.WriteLine( "Press Enter to quit." );
                Console.ReadLine();
            }
        }
    }

    public class UserController : ApiController
    {
        public async Task<IHttpActionResult> Get()
        {
            await Task.Delay( 2000 );

            return this.Ok( 1 );
        }
    }
}
```

Klient usługi sieciowej:

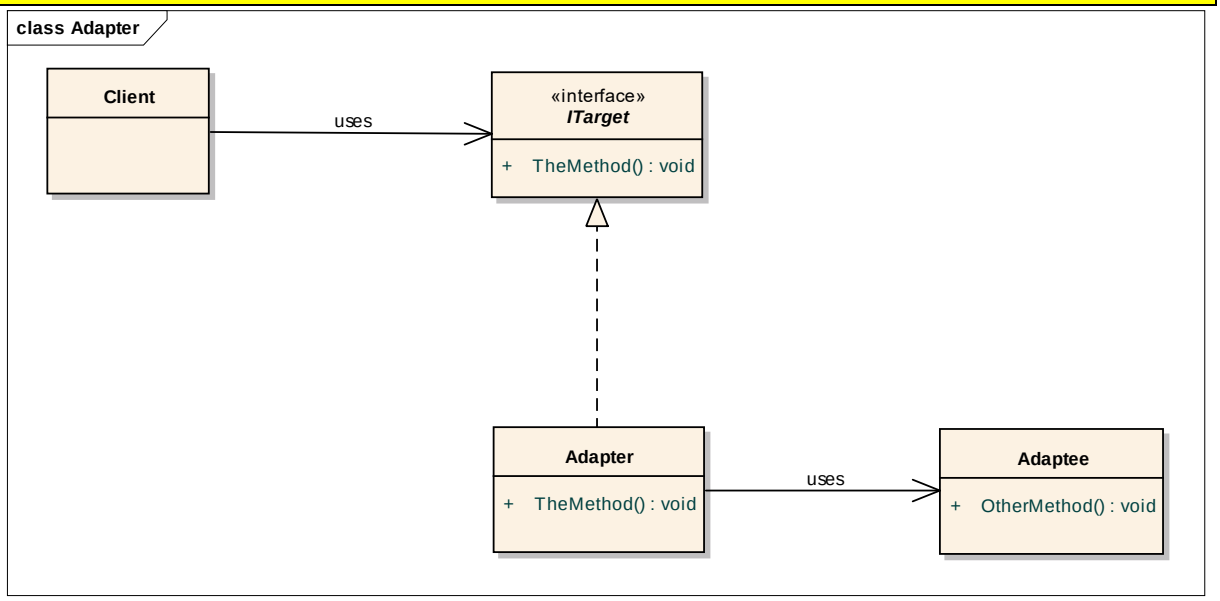
```
using Polly.CircuitBreaker;
using Polly;
using System;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
using Polly.Wrap;
using System.IO;

namespace PollyCircuitBreaker
{
    internal class Program
    {
        static void Main( string[] args )
        {
            DoWork();

            Console.ReadLine();
        }

        static void DoWork()
        {
            int n = 0;

            while ( true )
            {
                var i = n++;

                Task.Run( async () =>
                {
                    var service         = new ProductService();
                    // identyfikator korelacji
                    string operationKey = Guid.NewGuid().ToString();
```

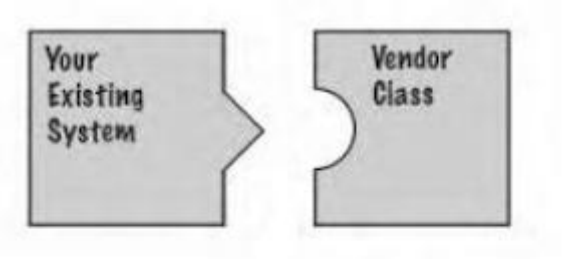
```
                    try
                    {
                        // wysłanie do sieci
                        CustomConsole.WriteLine( "DoWork::calls {0} {1}", i,
operationKey );
                        var result = await service.GetSample( i, operationKey
 );

                        CustomConsole.WriteLine( "DoWork::returns {0} = {1} {
2}", i, result, operationKey );
                    }
                    catch ( BrokenCircuitException )
                    {
                        // bezpiecznik podniesiony - brak komunikacji
                        CustomConsole.WriteLine( "DoWork::broken circuit hand
led for {0} {1}", i, operationKey );
                    }
                    catch ( Exception )
                    {
                        // bezpiecznik właśnie podniósł się -
 błąd komunikacji
                        CustomConsole.WriteLine( "DoWork::failed for {0} {1}"
, i, operationKey );
                    }
                } );

                Thread.Sleep( 500 );
            }
        }
    }

    /// <summary>
    /// Wywołanie usługi sieciowej
    /// </summary>
    public class ProductService
    {
        private static HttpClient _client = new HttpClient();

        private static readonly AsyncPolicyWrap _policy;

        static ProductService()
        {
            _client.Timeout = TimeSpan.FromSeconds( 3 );

            // polityka ponawiania
            var _retryPolicy = Policy.Handle<Exception>()
                                     .WaitAndRetryAsync(
                                         4,
                                         //n => TimeSpan.FromSeconds( Math.Pow
(2, n)),
                                         n => TimeSpan.FromSeconds(1),
                                         (ex, t, ctx) =>
                                         {
                                             CustomConsole.WriteLine( "retry {
0} in {1}", ctx.OperationKey, t );
                                         } );

            // polityka "circuit breaker"
            var _cbPolicy = Policy.Handle<Exception>()
                                          .CircuitBreakerAsync( 1, TimeSpan.F
```

```
romSeconds( 20 ),
                                                ( ex, span, context ) =>
                                                {
                                                    CustomConsole.WriteLine( "break
 {0}", context.OperationKey );

                                                    throw new BrokenCircuitExceptio
n();
                                                },
                                                (context) =>
                                                {
                                                    CustomConsole.WriteLine( "on re
set {0}", context.OperationKey );
                                                },
                                                () =>
                                                {
                                                    CustomConsole.WriteLine( "on ha
lf open" );
                                                } );

            // ostateczna polityka to "złożenie" obu
            _policy = _cbPolicy.WrapAsync( _retryPolicy );
        }

        public async Task<int> GetSample( int n, string operationKey )
        {
            // policy.ExecuteAsync to proxy (tu "ponawianie + circuit breaker
")
            return await _policy.ExecuteAsync<int>( async (c) =>
            {
                // rzeczywiste wywołanie usługi
                var response = await _client.GetAsync("http://localhost:8087/
api/user");
                var result   = await response.Content.ReadAsStringAsync();

                return n;
            }, new Context(operationKey) );
        }
    }

    public class CustomConsole
    {
        public static void WriteLine( string message, params object[] ps )
        {
            string s = string.Format( DateTime.Now + " " + message + "\r\n",
ps );
            Console.Write( s );
        }
    }
}
```

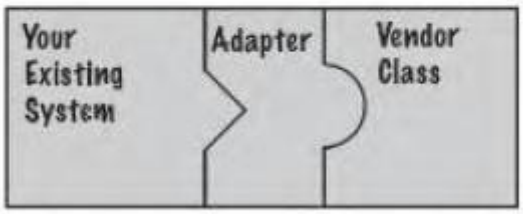
6 Adapter

Motto: uzgadnianie niezgodnych interfejsów





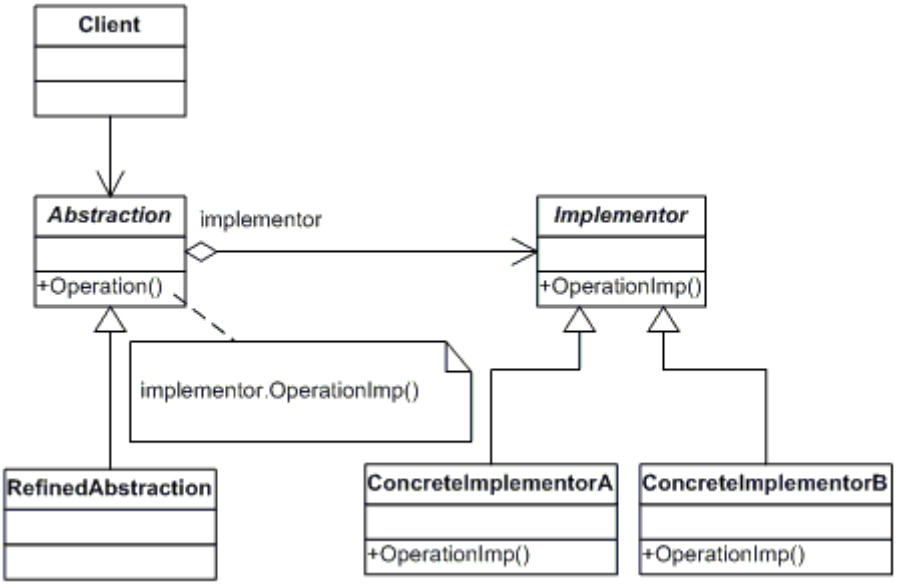
Not Compatible? We are in trouble



God Adapter Save our Life

7 Bridge

Motto: SRP + ISP + DIP dla hierarchii obiektowej o dwóch stopniach swobody; oddzielenie abstrakcji od implementacji



Poniższa klasa **PersonRegistry** ma dwa stopnie swobody:

- Źródło danych – możliwość pozyskiwania danych z różnych źródeł (np. SQL, XML)
- Przetwarzanie danych – wyprowadzenie na konsolę/do usługi http itp.

```
public class PersonRegistry
{
    /// <summary>
    /// Pierwszy stopień swobody - różne wczytywanie
    /// </summary>
    private List<Person> _persons = new List<Person>();

    /// <summary>
    /// Drugi stopień swobody - różne użycie
    /// </summary>
    public void NotifyPersons()
    {
        foreach ( Person person in _persons )
            Console.WriteLine( person );
    }
}

public class Person { }
```

Refaktoryzacja zgodnie ze wzorcem **Bridge** polega na wyłączeniu któregoś ze stopni swobody na zewnątrz, do osobnej hierarchii i utrzymaniu jednego ze stopni swobody wewnątrz klasy, z możliwością przeciążenia (dziedziczenia) które zmienia implementację.

W poniższej wersji, w klasie pozostała odpowiedzialność pozyskiwania danych a na zewnątrz wyłączono odpowiedzialność przetwarzania danych.

Analogiczną refaktoryzację wyłączającą na zewnątrz pozyskiwanie a pozostawiająca w klasie przetwarzanie pozostawia się jako ćwiczenie dla Czytelnika.

Należy zwrócić uwagę na to że Bridge nie zajmuje się szczegółami technicznymi dotyczącymi tego w jaki sposób odpowiedzialność implementacji (ta wynoszona na zewnątrz) jest dostarczana do klasy ani kiedy. O uporządkowaniu zasad komponowania zależności porozmawiamy dopiero przy okazji szerszego omawiania wzorca **Inversion of Control**.

```
public abstract class AbstractPersonRegistry
{
    public IMessenger messenger;

    private List<Person> _persons;

    /// <summary>
    /// Pierwszy stopień swobody - różne wczytywanie
    /// </summary>
    public abstract void LoadPersons();

    /// <summary>
    /// Drugi stopień swobody - różne użycie
    /// </summary>
    public void NotifyPersons()
    {
        foreach ( Person person in _persons )
            messenger.Notify( person );
    }
}

public interface IMessenger
{
    void Notify( Person person );
}

public class Person { }
```

8 Literatura

[1] Gamma, Helm, Johnson, Vlissides – Wzorce projektowe
[2] Martin, Martin – Zasady, wzorce I praktyki zwinnego wytwarzania oprogramowania w C#
[3] Grand, Merrill – Wzorce projektowe
[4] Freeman, Freeman, Sierra, Bates – Head First Design Patterns
[5] http://www.oodesign.com