Zastosowanie wzorców projektowych do zamówień zestawów obiadowych w restauracjach

Karol Bronowski

Wyższa Szkoła Ekonomii i Informatyki,
Kraków, 2018. *karol.bronowski@me.com*

Streszczenie: Dokumentacja do projektu zaliczeniowego
z przedmiotu „Programowanie obiektowe w języku C#”
WSEI, Kraków, rok akademicki 2017/2018

1. Czym są wzorce projektowe?

To wielokrotnie powtórzone (powtarzające się) i pozytywnie zweryfikowane schematy rozwiązań często spotykanych problemów projektowych. Dotyczą one architektury całej aplikacji, a nie pojedynczej klasy.

Wzorce projektowe nie są wynajdywane, ale odkrywane. Jakiś subtelny, świetny sposób rozwiązania jakiegoś problemu nie jest wzorcem projektowym. Wzorzec musi być zastosowany wielokrotnie, niejako pojawić się w codziennej praktyce projektowej, potwierdzić swoje znaczenie. W tej praktyce jest dostrzegany i właśnie odkrywany.

Wzorców nie należy mylić z tzw. *applications framework* (przykładowo JCF nie jest wzorcem projektowym). Te ostatnie dotyczą raczej bardziej technicznych szczegółów, w tym implementacyjnych i znajdują się na niższym poziomie abstrakcji niż wzorce.

2. Po co używać wzorców projektowych?

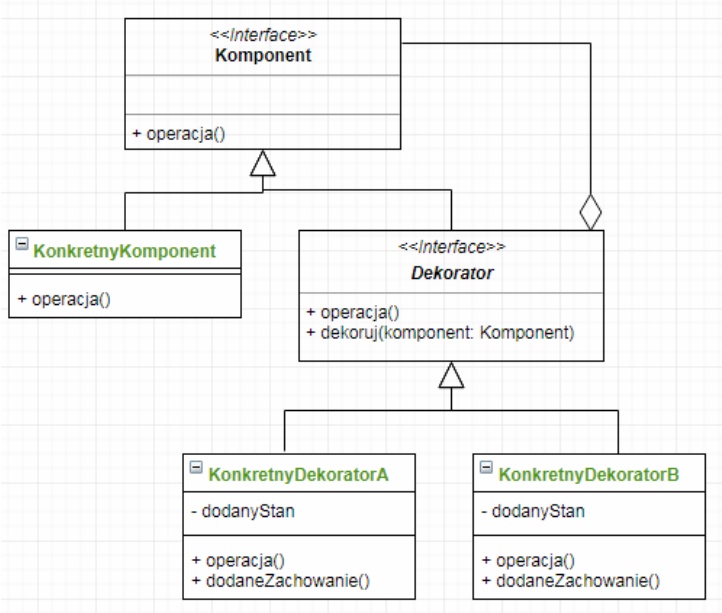
Stosowanie wzorców projektowych pozwala na pisanie lepszych, bardziej efektywnych, skalowalnych, łatwiej modyfikowalnych, mniej narażonych na błędy programów. Przede wszystkim dlatego, że w codziennej praktyce, doświadczeniu całych rzesz projektantów i programistów te właśnie problemy są w centrum uwagi i te właśnie problemy są rozwiązywane tak czy inaczej.

Dobre rozwiązania sprawdzają się, powtarzają, w końcu trafiają do "spisu" wzorców projektowych. Wzorce są dobrze opisane, wyjaśnione na konkretnych przykładach - dają więc szansę na poznanie dobrych metod programowania i zastosowanie ich po to, by ułatwić proces tworzenia i wdrażania aplikacji czy systemów.

3. Wykorzystanie przykładowego wzorca w projekcie

W programie został wykorzystany wzorzec strukturalny . **Dekorator** (ang. *decorator*) charakteryzuje się dynamicznym dodawaniem funkcjonalności do obiektów. Jest to alternatywa do dziedziczenia które rozszerza zachowanie w trakcie kompilacji, w przeciwieństwie do dekoratora który rozszerza klasy w czasie wykonywania programu.

Dekorator pozwala umieszczać obiekt w innym obiekcie który doda wspomnianą ramkę lub pasek przewijania (również oba naraz). Interfejs dekoratora musi być zgodny z interfejsem ozdabianego obiektu, dzięki temu jest przeźroczysty i umożliwia rekurencyjne zagnieżdżanie dekoratorów.



Rysunek 1. *Struktura wzorca*

- **Plusy wykorzystania wzorca:**

  - zapewnia większą elastyczność niż statyczne dziedziczenie,

  - pozwala uniknąć tworzenia przeładowanych funkcjami klas na wysokich poziomach hierarchii.

- **Minusy wykorzystania wzorca:**

  - dekorator i powiązany z nim komponent nie są identyczne,
  - powstawanie wielu małych obiektów.

4. Projekt i implementacja

Listing 1.1. *Tworzymy abstrakcyjną klasę „Zestaw”, w której tworzymy funkcje do obliczania kosztu zamówienia, jak i pobierania nazwy co klient zamawia.*

```
using System;
namespace Restauracja
{
    public abstract class Zestaw
    {
        public abstract double ObliczKoszt();
        public abstract string PobierzNazwe();
    }
}

```

Listing 1.2. *Tworzymy dekorator dziedziczący z klasy „Zestaw”.*

```
using System;
namespace Restauracja
{
    public class ZestawDecorator : Zestaw
    {
        protected Zestaw _zestaw;        //obiekt który będzie dekorowany

        public ZestawDecorator(Zestaw zestaw)
        {
            _zestaw = zestaw;
        }
        public override double ObliczKoszt()
        {
            return _zestaw.ObliczKoszt();
        }

        public override string PobierzNazwe()
        {
            return _zestaw.PobierzNazwe();
        }
    }
}

```

Listing 2.1. *Tworzymy podstawową cenę małego zestawu (bez względu na to co klient zamówi).*

```
using System;
namespace Restauracja
{
    public class MalyZestaw : Zestaw
    {
        public override double ObliczKoszt()
        {
            return 1.00;
        }

        public override string PobierzNazwe()
        {
            return "Mały zestaw zawiera: ";
        }
    }
}

```

Listing 2.2. *Adekwatnie tworzymy cenę dużego zestawu.*

```
using System;
namespace Restauracja
{
    public class DuzyZestaw : Zestaw
    {
        public override double ObliczKoszt()
        {
            return 5.00;
        }

        public override string PobierzNazwe()
        {
            return "Duża zestaw zawiera: ";
        }
    }
}

```

Listing 3.1. *Tworzymy klasę, w której znajduje się pierwszy element zestawu i przypisujemy mu cenę. Pamiętamy o dziedziczeniu z dekoratora zestawu.*

```
using System;
namespace Restauracja
{
    public class Frytki : ZestawDecorator
    {
        public Frytki(Zestaw zestaw) : base(zestaw)
        {

        }

        public override double ObliczKoszt()
        {
            return base.ObliczKoszt() + 4.90;
        }

        public override string PobierzNazwe()
        {
            return base.PobierzNazwe() + "frytki";
        }
    }
}

```

Listing 3.2. *Wykonując te same czynności tworzymy kolejne elementy zestawu obiadowego.*

```
using System;
namespace Restauracja
{
    public class Ziemniaki : ZestawDecorator
    {
        public Ziemniaki(Zestaw zestaw) : base(zestaw)
        {

        }

        public override double ObliczKoszt()
        {
            return base.ObliczKoszt() + 3.90;
        }

        public override string PobierzNazwe()
        {
            return base.PobierzNazwe() + "ziemniaki";
        }
    }
}

```

```
using System;
namespace Restauracja
{
    public class Kotlet : ZestawDecorator
    {
        public Kotlet(Zestaw zestaw) : base(zestaw)
        {

        }
        public override double ObliczKoszt()
        {
            return base.ObliczKoszt() + 8.90;
        }

        public override string PobierzNazwe()
        {
            return base.PobierzNazwe() + ", kotlet";
        }
    }
}

```

```
using System;
namespace Restauracja
{
    public class Ryba : ZestawDecorator
    {
        public Ryba(Zestaw zestaw) : base(zestaw)
        {

        }

        public override double ObliczKoszt()
        {
            return base.ObliczKoszt() + 9.90;
        }

        public override string PobierzNazwe()
        {
            return base.PobierzNazwe() + ", ryba";
        }
    }
}

```

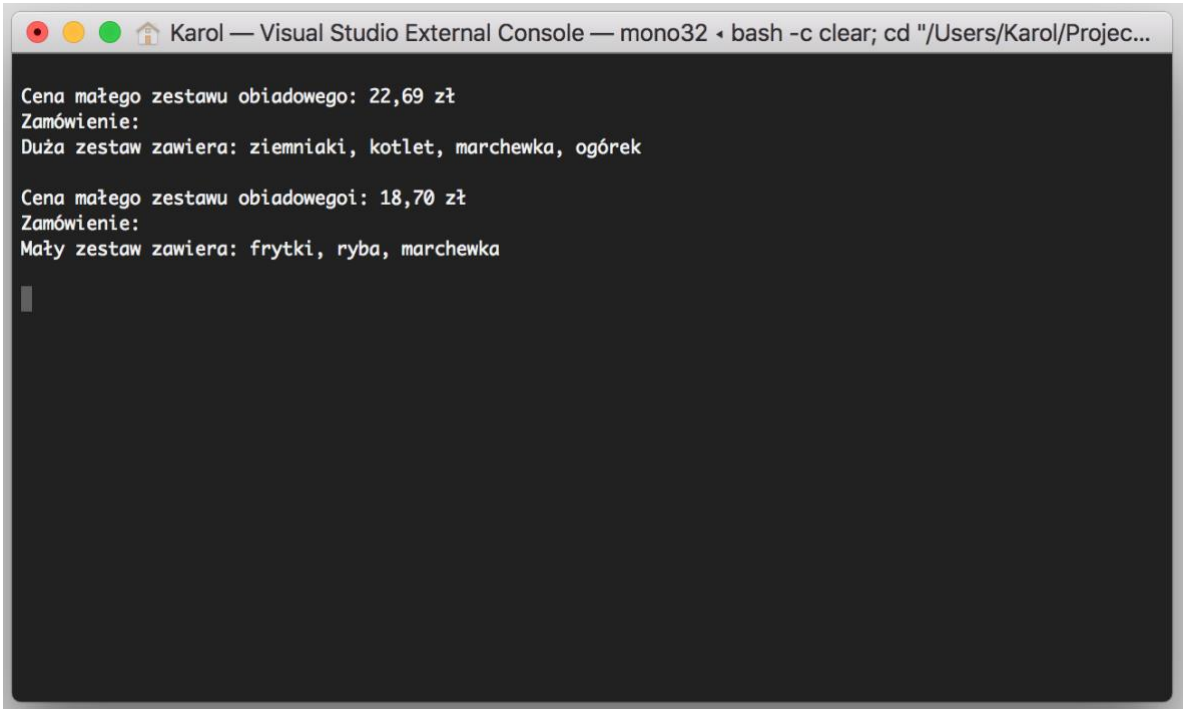
```
using System;
namespace Restauracja
{
    public class Marchweka : ZestawDecorator
    {
        public Marchweka(Zestaw zestaw) : base(zestaw)
        {

        }
        public override double ObliczKoszt()
        {
            return base.ObliczKoszt() + 2.90;
        }

        public override string PobierzNazwe()
        {
            return base.PobierzNazwe() + ", marchewka";
        }
    }
}

```

```
using System;
namespace Restauracja
{
    public class Ogórek : ZestawDecorator
    {
        public Ogórek(Zestaw zestaw) : base(zestaw)
        {

        }
        public override double ObliczKoszt()
        {
            return base.ObliczKoszt() + 1.99;
        }

        public override string PobierzNazwe()
        {
            return base.PobierzNazwe() + ", ogórek";
        }
    }
}

```

Listing 4. *Plik Program.cs, w którym tworzymy zamówienie dla klienta.*

```
using System;

namespace Restauracja
{
    class Program
    {
        static void Main(string[] args)
        {
            Zestaw duzyZestaw = new DuzyZestaw();

            duzyZestaw = new Ziemniaki(duzyZestaw);
            duzyZestaw = new Kotlet(duzyZestaw);
            duzyZestaw = new Marchweka(duzyZestaw);
            duzyZestaw = new Ogórek(duzyZestaw);

            Zestaw malyZestaw = new MalyZestaw();

            malyZestaw = new Frytki(malyZestaw);
            malyZestaw = new Ryba(malyZestaw);
            malyZestaw = new Marchweka(malyZestaw);

            Console.WriteLine("Cena małego zestawu obiadowego: " + "{0:C2}", duzyZestaw.ObliczKoszt());
            Console.WriteLine("Zamówienie: \n" + duzyZestaw.PobierzNazwe());
            Console.WriteLine();

            Console.WriteLine("Cena małego zestawu obiadowegoi: " + "{0:C2}", malyZestaw.ObliczKoszt());
            Console.WriteLine("Zamówienie: \n" + malyZestaw.PobierzNazwe());
            Console.WriteLine();

            Console.ReadKey();
        }
    }
}

```

Rysunek 2. *Uruchomiona aplikacja.*



5. Powiązane wzorce

- adapter - dekorator modyfikuje jedynie zadania obiektu, a nie jego interfejs,
- kompozyt - dekorator dodaje nowe zadania i nie jest przeznaczony do łączenia obiektów (można traktować jako uproszczony komponent),
- strategia - dekorator umożliwia zmianę "skórki" obiektu, do modyfikowania mechanizmów służy strategia 92 sposoby zmieniania obiektów).

6. Podsumowanie

W niniejszym projekcie przedstawiono zastosowanie przykładowego wzorca projektowego. Dekorator jest wzorcem strukturalnym, który doskonale nadaje się do programów dla firm realizujących zamówienia dla klientów. Przedstawiono czym jest wzorzec strukturalny jak i jego sens użycia. Wykazano również plusy jak i minusy użycia wzorca oraz opisano części kodu.

7. Literatura

- http://edu.pjwstk.edu.pl/wyklady/zap/scb/W5/W5.htm
- http://e.wsei.edu.pl/course/view.php?id=317 – Kurs wzorców projektowych
- Steven John Metsker, „Kanon Informatyki C# Wzorce projektowe”, Wyd. Helieton, Gliwice 2005.