Wprowadzenie niektórych zagadnień OOP oraz wzorce operacyjne

1. Wprowadzenie

Wyobraźmy sobie, że jesteśmy firmą tworzącą oprogramowanie i dostaliśmy właśnie zlecenie na stworzenie symulatora kaczek.

Dostaliśmy jednak jedno wymaganie, które nasz symulator musi spełniać:

1. Wszystkie kaczki kwaczą i pływają

Tak więc przechodzimy do pracy i na chwilę obecną wystarcza nam taki twór:

```
abstract class Kaczka
{
    2 references
    public void Kwacz()
    {
        Console.WriteLine("Kwa! Kwa!");
    }

    0 references
    public void Plywaj()
    {
        Console.WriteLine("Jestem kaczką więc pływam!");
    }

    4 references
    public abstract void Wyswietl();
}
```

Jest to klasa abstrakcyjna kaczki, która zawiera metody kwakania i pływania (jako że wszystkie kaczki kwaczą i pływają tak samo – przynamniej na chwilę obecną) oraz interfejs wymuszający na konkretnej kaczce implementacje metody *Wyswietl*.

Niżej przedstawiam implementacje konkretnej kaczki:

```
class DzikaKaczka : Kaczka
{
    4 references
    public override void Wyswietl()
    {
        Console.WriteLine("Jestem dziką kaczką.");
    }
}
```

Dostaliśmy jednak dodatkowe wymaganie podczas tworzenia naszego symulatora. Otóż zarząd firmy zlecającej nam stworzenie programu stwierdził, że mamy dodać kaczkom możliwość latania. Odruchowo dodajemy do naszej klasy abstrakcyjnej metodę *Lec*, która na pierwszy rzut oka rozwiązuje problem, a my czujemy się dumnie z naszych umiejętności programistycznych ☺

Obecnie nasza klasa abstrakcyjna wygląda tak:

```
abstract class Kaczka
{
    2 references
    public void Kwacz()
    {
        Console.WriteLine("Kwa! Kwa!");
    }

    0 references
    public void Plywaj()
    {
        Console.WriteLine("Jestem kaczką więc pływam!");
    }
    0 references
    public void Lec() {
        Console.WriteLine("Ja latam!");
    }

    4 references
    public abstract void Wyswietl();

}
```

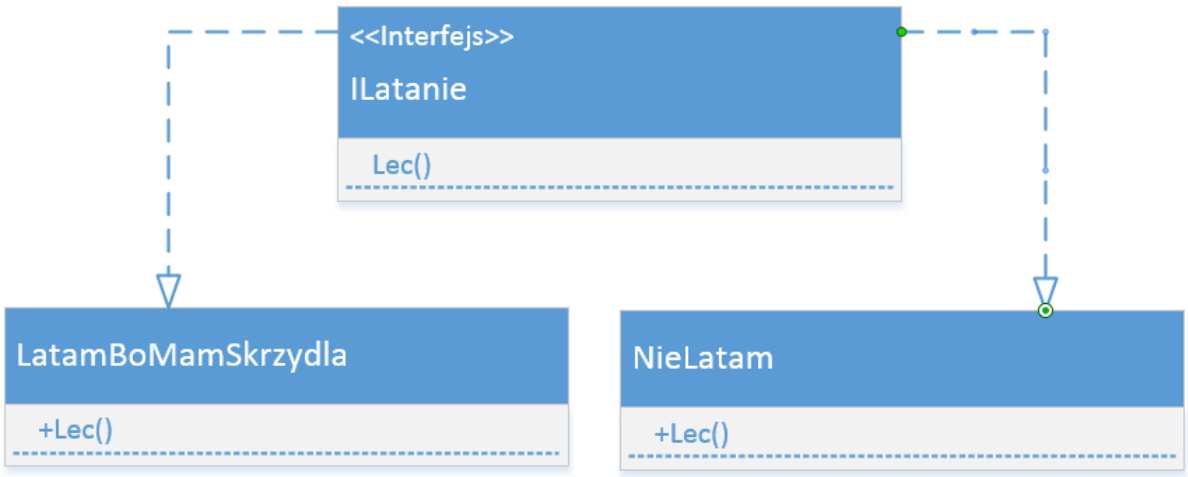
Niestety dostajemy telefon od zarządu, że dodali sobie gumową kaczkę, która niespodziewanie zaczęła latać! Nie jest to zachowanie pożądane i musimy ten problem rozwiązać.

Możemy w tym momencie zauważyć, że dziedziczenie jakie tutaj zastosowaliśmy nie do końca jest dobre. Dziedziczymy metodę, którą nie w każdej klasie podrzędnej chcemy, więc możemy stworzyć interfacey, które dostarczą nam kontraktu dotyczącego latania, kwakania i pływania. Daje nam to tyle, że implementować te interfacey będą tylko te konkretne kaczki, które potrafią wykonać daną czynność. Zastanówmy się jednak przez chwile…Przecież takie kaczki jak dzika oraz płaskonosa kwaczą i pływają tak samo, także zobaczmy ile kodu będziemy musieli duplikować, co sprawi nam wiele pracy przy jakichkolwiek zmianach.

W takim razie co robimy? Wiemy, że dziedziczenie to nie jest dobre rozwiązanie, ponieważ nie wszystkie podklasy potrzebują wszystkich metod. Interfacey są dobrą ścieżką, ale nie w takiej formie jak przedstawiłem wyżej. Rozwiązaniem jest **hermetyzacja** tych zachowań.

Hermetyzacja to bardzo ważne pojęcie w programowaniu obiektowym, które oznacza ukrywanie implementacji i jej „pudełkowanie”.

Wyodrębnijmy sobie konkretne zachowania związane z lataniem, kwakaniem, itp. Stworzymy sobie odrębny obiekt, np. *LatamBoMamSkrzydla* i on będzie implementował interface *ILatanie*.



Mając taki schemacik, nasze kaczki potrzebują obiektów, które będą implementowały zachowania. Dodatkowo warto zauważyć, że dzięki hermetyzacji, stworzone przez nas zachowania mogą być wykorzystane w innych projektach, czyli to co w OOP jest piękne – **ponowne wykorzystanie stworzonego kodu**!

W tym momencie, mogę wprowadzić ciekawe pojęcie jakim jest **dependency injection**. Pojęcie to oznacza możliwość dostarczenia (wstrzyknięcia!) obiektu A do obiektu B, którego zachowanie jest zależne od obiektu A, dzięki temu możemy zmieniać zachowanie obiektu B „in runtime", czyli w czasie wykonywania się programu.

Z tym podejściem wiąże się coś w rodzaju zasady, czy też reguły:

„Skoncentruj się na tworzeniu interfejsów, a nie implementacji".

Nie jest jednak powiedziane, że MUSIMY tworzyć obiekty w taki sposób jak powyższy, ponieważ zależy to od naszych potrzeb. Jeśli potrzebujemy udostępnić interfejs, to go przeważnie implementujemy, by móc oświadczyć wszystkim, że dostarczamy dany kontrakt, ale jeśli to nasz obiekt jest tym, który wykorzysta ten interfejs, to wtedy przyjmujemy obiekty dostarczające ów kontrakt.

W tym momencie, nasza klasa abstrakcyjna wygląda następująco:

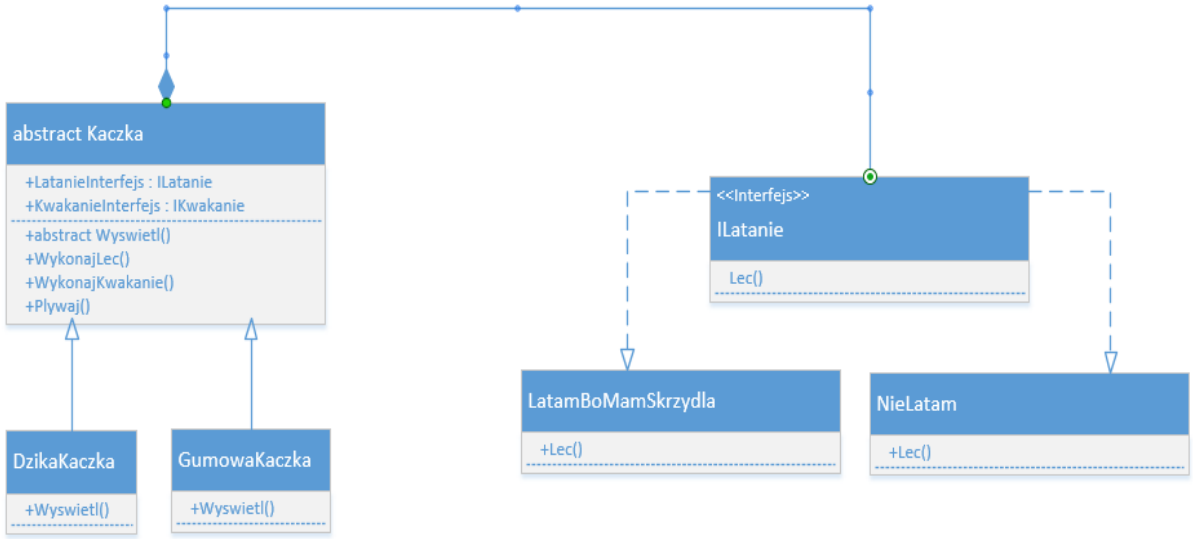
```
abstract class Kaczka
{
    5 references
    public LatanieInterfejs LatanieInterfejs { get; set; }

    4 references
    public KwakanieInterfejs kwakanieInterfejs { get; set; }

    3 references
    public abstract void Wyswietl();

    3 references
    public void WykonajLec()
    {
        LatanieInterfejs.Lec();
    }

    1 reference
    public void WykonajKwakanie()
    {
        kwakanieInterfejs.Kwacz();
    }

    0 references
    public void Plywaj()
    {
        System.Console.WriteLine("Wszystkie kaczki pływają, nawet te sztuczne!");
    }
}
```

Czy jak ktoś woli:

Stworzyliśmy sobie grupę zachowań prawda? Możemy je nazwać algorytmami, ponieważ definiują sposób w jaki obiekt będzie latał, czy też kwakał. Dodatkowo, lekko pod stołem, wślizgnęła nam się kolejna bardzo ważna zasada OOP:

„Przekładaj kompozycję nad dziedziczenie”

Dlaczego? Spójrzmy na część schematu UML naszego symulatora, związanego z lataniem:



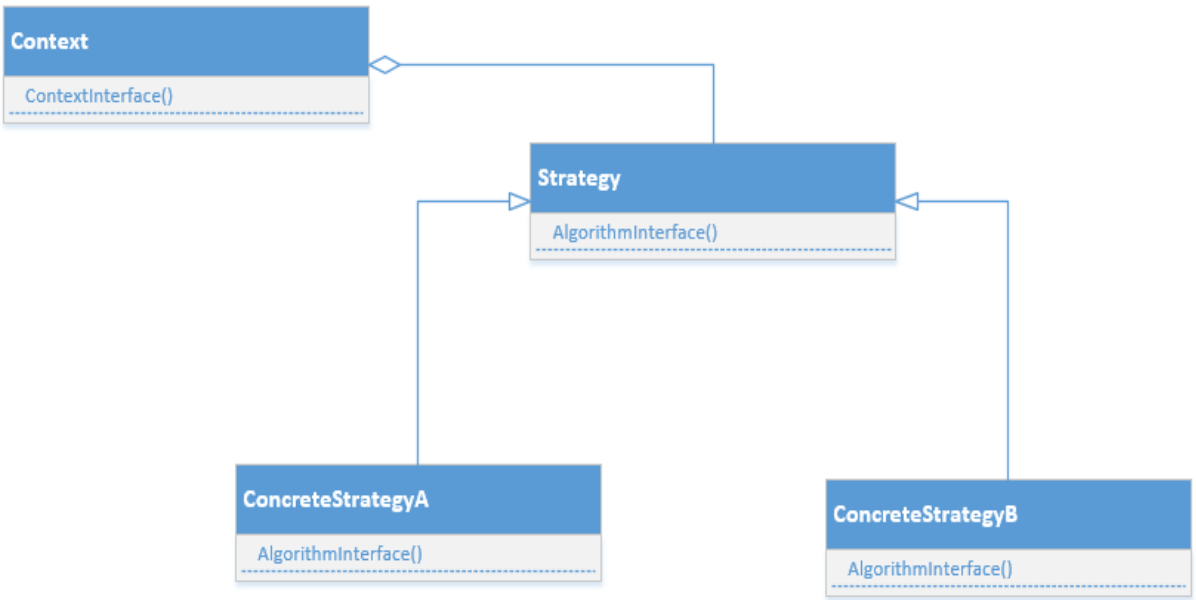
W tym momencie naszym oczom ukazał się wzorzec projektowy **Strategia!**

2. Wzorzec Strategia

Strategia definiuje rodzinę algorytmów, pakuje je jako osobne klasy i powoduje, że są one w pełni wymienne. Zastosowanie tego wzorca pozwala na to, żeby zmiany w implementacji algorytmów przetwarzania były całkowicie niezależne od strony klienta, który z nich korzysta.

Wzorzec pozwala nam na to, abyśmy skonfigurowali dana klasę konkretnym zachowaniem, którym kolejne klasy właśnie się różnią, a dodatkowo implementacja tych zachowań/algorytmów, może być zróżnicowana, ponieważ nasz obiekt chce tylko odpowiedni interfejs umożliwiający rozmawianie z dostarczonym zachowaniem.

Jego struktura wygląda następująco:



To właśnie ten wzorzec wykorzystuje, wspomnianą już wcześniej, regułę projektowania **„Przekładaj kompozycję nad dziedziczenie”.**

3. PizzaMaker

Umiejąc już wzorzec Strategia, przyjrzyjmy się kolejnemu przykładowi. Przygotujmy sobie program przygotowujący pizzę i przeanalizujmy jakie problemy i kolejne projektowe spostrzeżenia nam on przyniesie.

Na początku potrzebujemy przepis aby zrobić jakąkolwiek pizzę. Przygotowałem przykładowe przepisy dla dwóch rodzajów pizz.

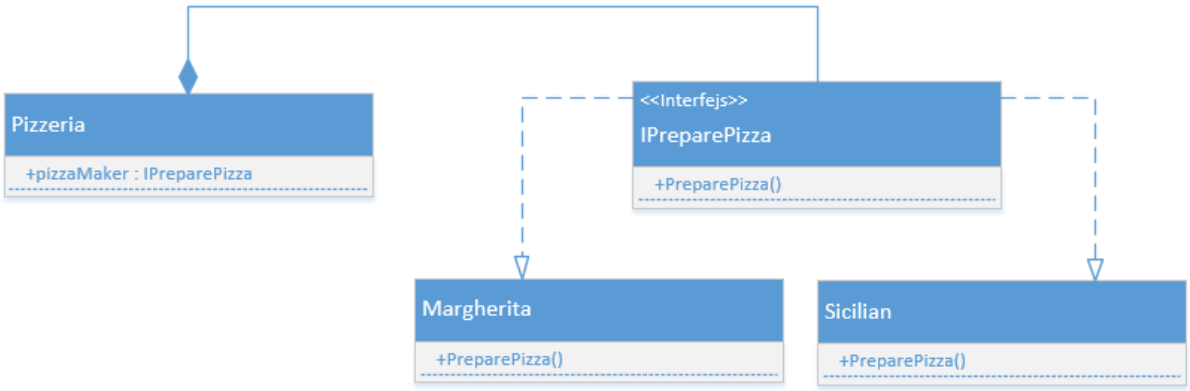
Margherita:

1. Przygotuj cienkie ciasto.
2. Dodaj sos pomidorowy.
3. Dodaj mozzarelle.
4. Dodaj bazylię i trochę oliwy.
5. Piecz przez 15 minut.

Sicilian:

1. Przygotuj grube ciasto.
2. Podaj nasz specjalny sos.
3. Dodaj oliwki i kapary.
4. Dodaj mieszankę przypraw.
5. Piecz przez 20 minut.

Widzimy dwa rodzaje przepisów, a przepis możemy nazwać algorytmem. Jeśli mamy tutaj przykład różnych algorytmów dążących do tego samego celu to możemy użyć Strategii, prawda? Spójrzmy na schemat:



Moglibyśmy zakończyć już nasze zadanie, ale przyjrzymy się bliżej naszym przepisom, przecież oba przedstawiają podobny schemat, a co za tym idzie mamy sporo skopiowanego kodu i tym samym nie stosujemy **reguły DRY** (Don’t Repeat Yourself).

Na pomoc przychodzi konkretny wzorzec projektowy, jest nim **Metoda Szablonowa**!

4. Metoda Szablonowa i PizzaMaker

Metoda Szablonowa określa szkielet algorytmu i pozostawia doprecyzowanie niektórych jego kroków podklasom. Umożliwia modyfikacje niektórych etapów algorytmu i podklasach bez zmiany jego struktury.

Jak ten wzorzec nam pomoże? Najpierw zróbmy coś z naszymi przepisami. Istnieje kolejna pomocna reguła projektowania:

„Refaktoryzacja w celu uogólnienia."

Za jej przykładem zrefaktorujmy nasze przepisy, by uzyskać ogólny przepis robienia pizzy.

Pizza:

1. Przygotuj ciasto.
2. Dodaj sos.
3. Dodaj dodatki.
4. Dodaj przyprawy.
5. Upiecz.

Jest kilka kroków, które różnią się szczegółami, ale główna idea/cel jest taki sam, jak, np. „przygotuj ciasto". Tworząc z tego przepisu klasę ogólną z punków od 1. do 5. tworzymy metody abstrakcyjne, dzięki temu, nasze klasy potomne zostaną zmuszone do ich implementacji. To są te kroki algorytmu, które są zmienne w zależności od pizzy. Dodatkowo klasa udostępnia metodę „PreparePizza", która służy do faktycznego uruchomienia algorytmu i przygotowania pizzy. W tym momencie jednak nie mamy wielkiego zysku z tego co zrobiliśmy, ponieważ wszystkie metody są abstrakcyjne, prócz tego, że tylko klasa ogólna zna tak naprawdę kroki algorytmu, a klasy potomne implementują poszczególne jego punkty, ale tak być nie musi! Niektóre zachowania mogą mieć, np. wartość domyślną jak krok dodawania sosu pomidorowego, który jest najczęściej stosowany. W tym momencie pojawia się prawdziwa moc, tzw. „**haczyki"** (hook methods). Implementują one w klasie nadrzędnej zachowanie domyślne, które możemy nadpisać w klasie podrzędnej.

Spójrzmy jak wygląda stworzenie pizz:

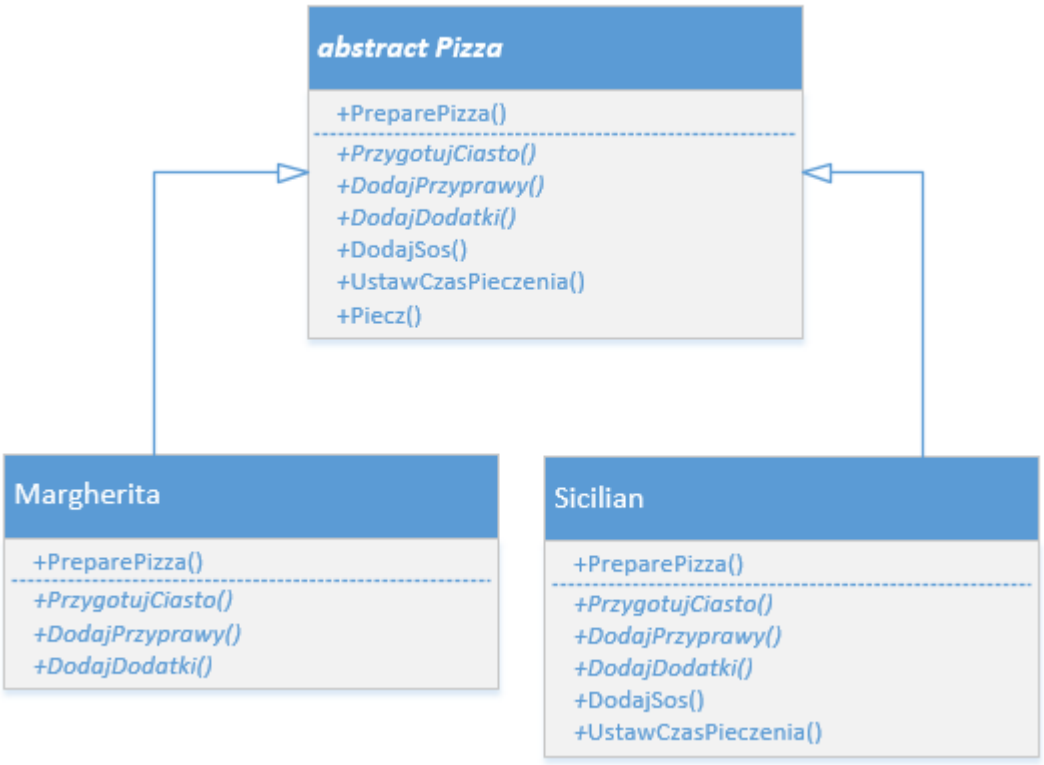
```
Pizza pizzaMargherita = new Margherita();
Pizza pizzaSicilian = new Sicilian();

pizzaMargherita.Prepare();
Console.WriteLine();
pizzaSicilian.Prepare();
```

Pisałem wcześniej, że w przypadku 5 abstrakcyjnych metod nie mamy zbytnio zysku, prócz tego, że tylko klasa nadrzędna zna kroki algorytmów. Jest to bardzo ważna własność, ponieważ jest to główna idea Metody Szablonowej, która kieruje się regułą:

„**Nie dzwoń do nas, to my zadzwonimy do Ciebie!"**

Jest to tzw. **zasada Hollywood**, znana również pod nazwą **inversion of control** (IoC). Oznacza to, że klasa podrzędna implementująca wszystkie/niektóre kroki algorytmu nie wywołuje ich ze swojego kodu, nie uruchamia algorytmu. Wywołaniem zajmuje się klasa nadrzędna, która zna przebieg algorytmu, a klasa podrzędna działa jako klient.

Przyjrzyjmy się schematowi UML naszej hierarchii:



5. Pytania na kolokwium

1. Podaj ogólną idee dependency injection i inversion of control.
2. Jak nazywają się metody umożliwiające nadpisanie w klasie podrzędnej we wzorcu “Metoda Szablonowa”?
3. Jakimi regułami kierują się wzorce Strategia i Metoda Szablonowa?