Politechnika Wrocławska
Wydział Mechaniczny
Katedra Konstrukcji i Badań Maszyn i Pojazdów

Programowanie obiektowe Lab 13

Dr inż. Paweł Maślak

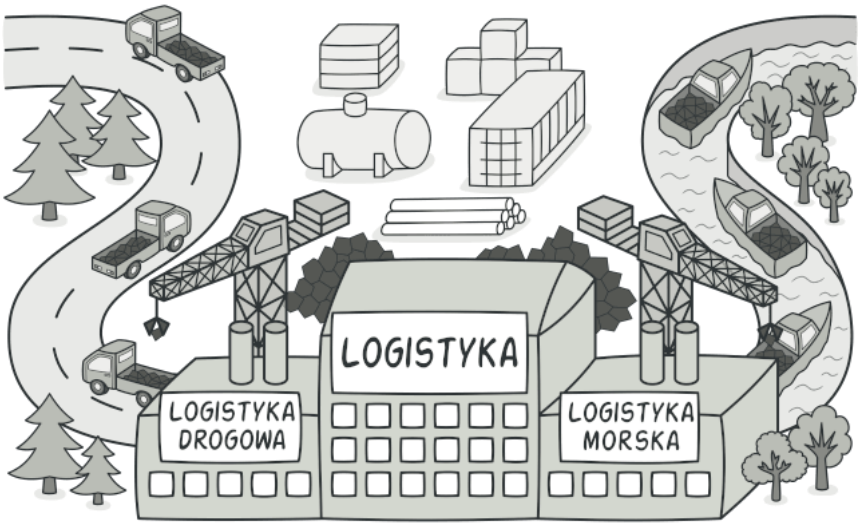
pawel.maslak@pwr.edu.pl

Laboratorium 13: Wybrane wzorce projektowe w C#

1. W pierwszej części zajęć zaprezentowane są przykłady koncepcyjne kodu w C# dla wzorców przedstawionych na wykładzie. W kolejnej części jest szeroko omówiony kolejny przykład wzorca koncepcyjnego : Fabryki abstrakcyjnej.

Wzorzec kreacyjny - **Metoda wytwórcza**

Metoda wytwórcza jest kreacyjnym wzorcem projektowym rozwiązującym problem tworzenia obiektów-produktów bez określania ich konkretnych klas.



Metoda wytwórcza definiuje metodę która ma służyć tworzeniu obiektów bez bezpośredniego wywoływania konstruktora (poprzez operator new). Podklasy mogą nadpisać tę metodę w celu zmiany klasy tworzonych obiektów.

Przykłady użycia: Wzorzec Metoda wytwórcza jest często stosowany w kodzie C#. Przydaje się gdy chcesz nadać swojemu kodowi wysoki poziom elastyczności.

Identyfikacja: Metody wytwórcze można poznać po metodach kreacyjnych tworzących obiekty na podstawie konkretnych klas, ale zwracających typ abstrakcyjny lub interfejs.

Przykład koncepcyjny

```
using System;

namespace RefactoringGuru.DesignPatterns.FactoryMethod.Conceptual
{
    // The Creator class declares the factory method that is supposed to return
    // an object of a Product class. The Creator's subclasses usually provide
    // the implementation of this method.
    abstract class Creator
    {
        // Note that the Creator may also provide some default implementation of
        // the factory method.
        public abstract IProduct FactoryMethod();

        // Also note that, despite its name, the Creator's primary
        // responsibility is not creating products. Usually, it contains some
        // core business logic that relies on Product objects, returned by the
        // factory method. Subclasses can indirectly change that business logic
        // by overriding the factory method and returning a different type of
        // product from it.
        public string SomeOperation()
        {
            // Call the factory method to create a Product object.
            var product = FactoryMethod();
            // Now, use the product.
            var result = "Creator: The same creator's code has just worked with "
                + product.Operation();

            return result;
        }
    }

    // Concrete Creators override the factory method in order to change the
    // resulting product's type.
    class ConcreteCreator1 : Creator
    {
        // Note that the signature of the method still uses the abstract product
        // type, even though the concrete product is actually returned from the
        // method. This way the Creator can stay independent of concrete product
        // classes.
        public override IProduct FactoryMethod()
        {
            return new ConcreteProduct1();
        }
    }

    class ConcreteCreator2 : Creator
    {
        public override IProduct FactoryMethod()
        {
            return new ConcreteProduct2();
        }
    }

    // The Product interface declares the operations that all concrete products
    // must implement.
    public interface IProduct
    {
        string Operation();
    }

    // Concrete Products provide various implementations of the Product
    // interface.
```

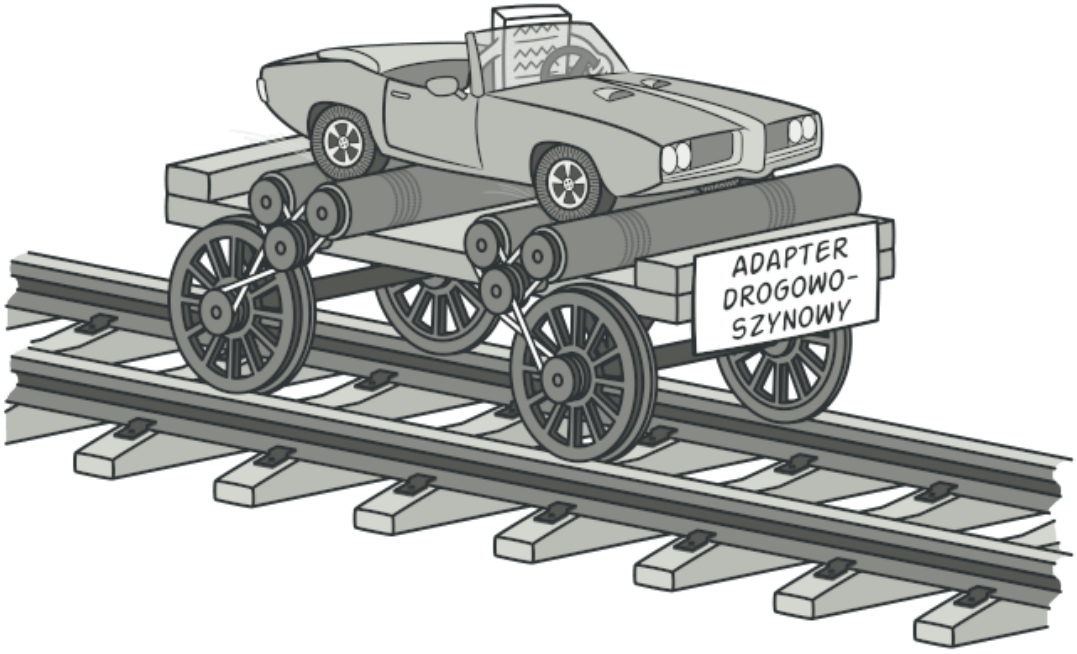
```
    class ConcreteProduct1 : IProduct
    {
        public string Operation()
        {
            return "{Result of ConcreteProduct1}";
        }
    }

    class ConcreteProduct2 : IProduct
    {
        public string Operation()
        {
            return "{Result of ConcreteProduct2}";
        }
    }

    class Client
    {
        public void Main()
        {
            Console.WriteLine("App: Launched with the ConcreteCreator1.");
            ClientCode(new ConcreteCreator1());

            Console.WriteLine("");

            Console.WriteLine("App: Launched with the ConcreteCreator2.");
            ClientCode(new ConcreteCreator2());
        }

        // The client code works with an instance of a concrete creator, albeit
        // through its base interface. As long as the client keeps working with
        // the creator via the base interface, you can pass it any creator's
        // subclass.
        public void ClientCode(Creator creator)
        {
            // ...
            Console.WriteLine("Client: I'm not aware of the creator's class," +
                "but it still works.\n" + creator.SomeOperation());
            // ...
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            new Client().Main();
        }
    }
}
```

Odpowiedź systemu:

```
App: Launched with the ConcreteCreator1.
Client: I'm not aware of the creator's class, but it still works.
Creator: The same creator's code has just worked with {Result of ConcreteProduct1}

App: Launched with the ConcreteCreator2.
Client: I'm not aware of the creator's class, but it still works.
Creator: The same creator's code has just worked with {Result of ConcreteProduct2}
```

Wzorzec strukturalny – **Adapter**

Adapter to strukturalny wzorzec projektowy pozwalający na współpracę niekompatybilnych obiektów ze sobą.



Adapter pełni rolę opakowania dwóch obiektów. Przechwytuje wywołania jednego z obiektów i przekształca je na format i interfejs zrozumiały dla drugiego obiektu.

Przykłady użycia: Wzorzec Adapter jest dość powszechny w kodzie C#. Często stosuje się go w systemach bazujących na przestarzałym kodzie, gdzie pozwala na współdziałanie takiego kodu z nowoczesnymi klasami.

Identyfikacja: Adapter można rozpoznać po konstruktorze przyjmującym instancję innego typu abstrakcji/interfejsu. Gdy adapter otrzymuje wywołanie kierowane do którejś z jego metod, tłumaczy parametry wywołania do stosownego formatu i przekazuje je do jednej lub wielu metod opakowanego obiektu.

Przykład koncepcyjny

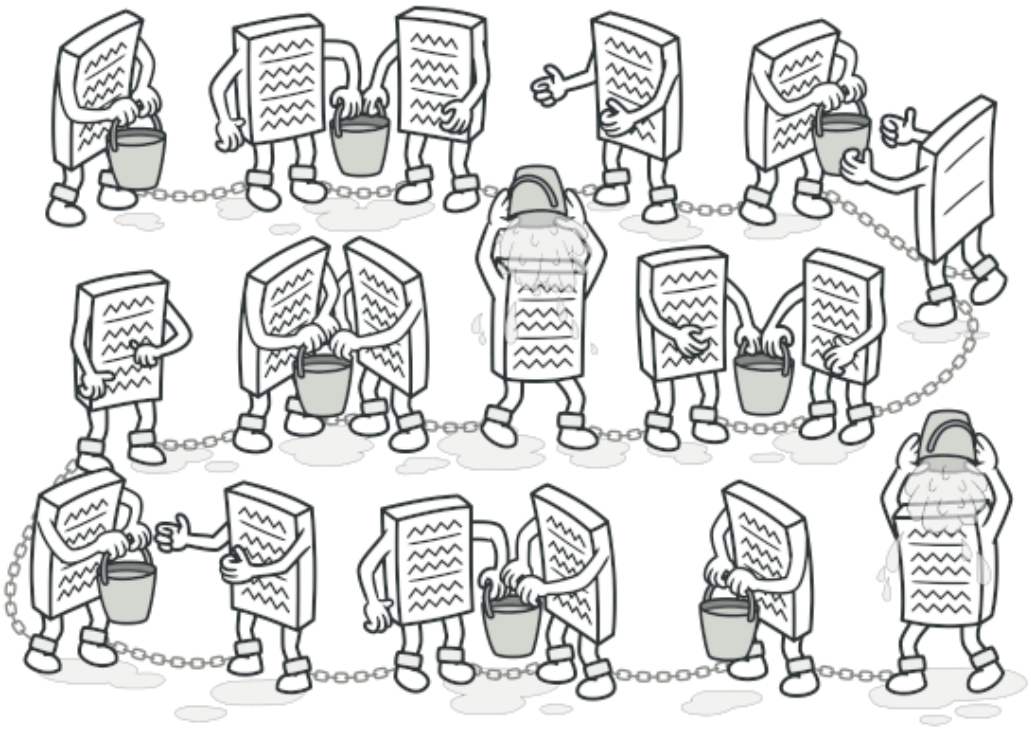
```
using System;
namespace RefactoringGuru.DesignPatterns.Adapter.Conceptual
{
    // The Target defines the domain-specific interface used by the client code.
    public interface ITarget
    {
        string GetRequest();
    }

    // The Adaptee contains some useful behavior, but its interface is
    // incompatible with the existing client code. The Adaptee needs some
    // adaptation before the client code can use it.
    class Adaptee
    {
        public string GetSpecificRequest()
        {
            return "Specific request.";
        }
    }

    // The Adapter makes the Adaptee's interface compatible with the Target's
    // interface.
    class Adapter : ITarget
    {
        private readonly Adaptee _adaptee;

        public Adapter(Adaptee adaptee)
        {
            this._adaptee = adaptee;
        }

        public string GetRequest()
        {
            return $"This is '{this._adaptee.GetSpecificRequest()}'";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Adaptee adaptee = new Adaptee();
            ITarget target = new Adapter(adaptee);

            Console.WriteLine("Adaptee interface is incompatible with the client.");
            Console.WriteLine("But with adapter client can call it's method.");

            Console.WriteLine(target.GetRequest());
        }
    }
}
```

Odpowiedź systemu:

```
Adaptee interface is incompatible with the client.
But with adapter client can call it's method.
This is 'Specific request.'
```

Wzorzec behawioralny – **Łańcuch zobowiązań**

Łańcuch zobowiązań to behawioralny wzorzec projektowy pozwalający przekazywać żądanie wzdłuż łańcucha potencjalnych obiektów obsługujących aż zostanie obsłużone.



W łańcuchu zobowiązań wiele obiektów może obsłużyć żądanie bez konieczności sprzęgania klas wysyłających je z konkretnymi klasami odbierającymi. Łańcuch można układać dynamicznie w trakcie działania programu z dowolnych obiektów obsługujących, wyposażonych w standardowy interfejs obsługi żądań.

Przykłady użycia: Łańcuch zobowiązań jest rzadkim rozwiązaniem w programach napisanych w C#, gdyż ma sens tylko w tym kodzie, w którym występują łańcuchy obiektów.

Identyfikacja: Wzorzec można rozpoznać na podstawie obecności behawioralnych metod jednej grupy obiektów pośrednio wywołujących analogiczne metody w innych obiektach za pośrednictwem wspólnego interfejsu.

Przykład koncepcyjny

```
using System;
using System.Collections.Generic;

namespace RefactoringGuru.DesignPatterns.ChainOfResponsibility.Conceptual
{
    // The Handler interface declares a method for building the chain of
    // handlers. It also declares a method for executing a request.
    public interface IHandler
    {
        IHandler SetNext(IHandler handler);
```

```
    object Handle(object request);
}

// The default chaining behavior can be implemented inside a base handler
// class.
abstract class AbstractHandler : IHandler
{
    private IHandler _nextHandler;

    public IHandler SetNext(IHandler handler)
    {
        this._nextHandler = handler;

        // Returning a handler from here will let us link handlers in a
        // convenient way like this:
        // monkey.SetNext(squirrel).SetNext(dog);
        return handler;
    }

    public virtual object Handle(object request)
    {
        if (this._nextHandler != null)
        {
            return this._nextHandler.Handle(request);
        }
        else
        {
            return null;
        }
    }
}

class MonkeyHandler : AbstractHandler
{
    public override object Handle(object request)
    {
        if ((request as string) == "Banana")
        {
            return $"Monkey: I'll eat the {request.ToString()}.\n";
        }
        else
        {
            return base.Handle(request);
        }
    }
}

class SquirrelHandler : AbstractHandler
{
    public override object Handle(object request)
    {
        if (request.ToString() == "Nut")
        {
            return $"Squirrel: I'll eat the {request.ToString()}.\n";
        }
        else
        {
            return base.Handle(request);
        }
    }
}

class DogHandler : AbstractHandler
{
    public override object Handle(object request)
    {
        if (request.ToString() == "MeatBall")
        {
            return $"Dog: I'll eat the {request.ToString()}.\n";
        }
        else
        {
            return base.Handle(request);
        }
```

```
        }
    }

    class Client
    {
        // The client code is usually suited to work with a single handler. In
        // most cases, it is not even aware that the handler is part of a chain.
        public static void ClientCode(AbstractHandler handler)
        {
            foreach (var food in new List<string> { "Nut", "Banana", "Cup of coffee" })
            {
                Console.WriteLine($"Client: Who wants a {food}?");

                var result = handler.Handle(food);

                if (result != null)
                {
                    Console.Write($"   {result}");
                }
                else
                {
                    Console.WriteLine($"   {food} was left untouched.");
                }
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // The other part of the client code constructs the actual chain.
            var monkey = new MonkeyHandler();
            var squirrel = new SquirrelHandler();
            var dog = new DogHandler();

            monkey.SetNext(squirrel).SetNext(dog);

            // The client should be able to send a request to any handler, not
            // just the first one in the chain.
            Console.WriteLine("Chain: Monkey > Squirrel > Dog\n");
            Client.ClientCode(monkey);
            Console.WriteLine();

            Console.WriteLine("Subchain: Squirrel > Dog\n");
            Client.ClientCode(squirrel);
        }
    }
}
```

Odpowiedź systemu:

```
Chain: Monkey > Squirrel > Dog

Client: Who wants a Nut?
   Squirrel: I'll eat the Nut.
Client: Who wants a Banana?
   Monkey: I'll eat the Banana.
Client: Who wants a Cup of coffee?
   Cup of coffee was left untouched.

Subchain: Squirrel > Dog

Client: Who wants a Nut?
   Squirrel: I'll eat the Nut.
Client: Who wants a Banana?
   Banana was left untouched.
Client: Who wants a Cup of coffee?
   Cup of coffee was left untouched.
```

Wzorzec kreacyjny – **Fabryka abstrakcyjna**

Jest praktycznie niemożliwe, aby utworzyć aplikację składającą się tylko z dwóch klas. Zazwyczaj aplikacja składa się z wielu klas, każda klasa jest odpowiedzialna za żądaną funkcjonalność. Oznacza to, że jest praktycznie niemożliwe, żeby klasy nie komunikowały się ze sobą. Może to być osiągnięte w bardzo prosty sposób, jeżeli pozwolimy klasie na utworzenie instancji klasy której potrzebujemy, będziemy w stanie wywoływać potrzebne metody.

Załóżmy, że mamy klasę **A**, która ma wywołać metodę z klasy **B**. Wystarczy, że obiekt klasy **B** będzie umieszczony wewnątrz **A**. Będziemy wówczas mogli wywołać metodę, której potrzebujemy. Dla łatwiejszej interpretacji problemu proszę spojrzeć na poniższy kod:

```
// Definicja klasy A
public class A
{
    // wewnątrz klasy A mamy obiekt B
    private B b;
    // w kontstruktorze A tworzymy obiekt klasy B
    public A()
    {
        b = new B();
    }
    // Metoda klasy korzysta z obiektu klasy B i wywołuje metodę z tej klasy
    public void EndTheIssue()
    {
        b.DoTaskOne();
    }
}
// Definicja klasy B
public class B
{
    // Wykonanie zadanie
    public void DoTaskOne()
    {
        Console.WriteLine("Zakończ zadanie");
    }
}
```

Podejście zapreznetowane wyżej będzie działać ale ma pewne wady. Pierwszym z nich jest fakt, że klasa taka musi wiedzieć o każdej klasie, którą chce wykorzystać. Spowoduje to, że zarządzanie taką aplikacją będzie naprawdę trudne. Ponadto takie podejście powoduje znaczy wzrost połączeń pomiędzy klasami.

Z punktu widzenia najlepszych praktyk, za każdym razem kiedy projektujemy nasze klasy powinniśmy mieć na uwadze zasadę odwracania zależności (**Dependency Inversion Principle**) jeżeli chodzi o zależności między klasami. Zasada ta mówi, że moduły wyższego poziomu powinny zależeć od abstrakcji a nie od modułów niskiego poziomu. Należy więc projektować nasze klasy w taki sposób, aby były one zależne od interfejsów i klas abstakcyjych a nie od implementacji konkretnej klasy.

Dlatego też klasy, które widzieliście w poprzednim przykładzie zostaną przeprojektowane. W pierwszej kolejności należy utworzyć interfejs, który **A** użyje do

wywołania metody **DoTaskOne()**. Klasa **B** powinna implementować ten interfejs. Po zmianach klasy z powyższego przykładu wyglądają tak:

```
// interfejs to tylko definicja
interface IDoTask
{
    void DoTaskOne();
}
// Definicja klasy B, która implementuje metodę interfejsu
public class B : IDoTask
{
    // Wykonanie zadanie
    public void DoTaskOne()
    {
        Console.WriteLine("Zakończ zadanie");
    }
}
public class A
{
    private IDoTask task;
    public A()
    {
        // jak utworzyć nowy obiekt w tym miejscu?
        // wywołanie task = new B();
        // wydaje się niewłaściwe
    }
    // Metoda klasy korzysta z interfejsu i wywołuje...o tym za chwilę
    public void EndTheIssue()
    {
        task.DoTaskOne();
    }
}
```

Powyższy przykład pokazuje klasy, które zostały zaprojektowane w bardzo dobry sposób. Moduły wyższego poziomu zależą od abstrakcji a moduły niższego poziomu implementacją tą abstrakcję. Ale jak zamierzamy utworzyć obiekt klasy **B**? Powinniśmy zrobić jak w poprzednim przykładzie, tj. utworzyć nowy obiekt **B** w konstruktorze klasy **A**? Czy jednak nie wpłynęłoby to na utracenie zachowania zasady odwracania zależności?

W tym właśnie miejscu przydatny będzie wyżej wspomniany wzorzec fabryki (**Factory Patern**). Wzorzec ten całkowicie ukrywa proces tworzenia obiektów oraz czyni abstrakcyjnym naszą odpowiedzialność za tworzenie klas z klas klienta. Główną zaletą takiego podejścia jest to, iż kod klienta jest całkowicie nieświadomy procesa tworzeniu klas zależnych. Aby dokonać tego w naszym powyższym przykładzie należy w wprowadzić następujące zmiany:

```
// interfejs to tylko definicja
interface IDoTask
{
    void DoTaskOne();
}
public class FactoryPatern
{
    // metoda zwracająca konkretne wykonanie
    // w naszym przypadku chodzi o obiekt klasy B
    public B GetConcreteDoable()
    {
        return new B();
```

```
    }
}
// Definicja klasy B, która implementuje metodę interfejsu
public class B : IDoTask
{
    // Wykonanie zadanie
    public void DoTaskOne()
    {
        Console.WriteLine("Zakończ zadanie");
    }
}
public class A
{
    private IDoTask task;
    public A()
    {
        // tworzymy nowy obiekt klasy FactoryPatern
        FactoryPatern fp = new FactoryPatern();
        // zwracamy konkretną implemtancję, w naszym wypadku to obiekt klasy B
        task = fp.GetConcreteDoable();
        // następnie możemy wywołać w naszej klasie metodę z klasy B
        task.DoTaskOne();
        Console.ReadKey();
    }
}
```

Takie luźne powiązanie pomiędzy klasami jest również korzystne z punktu widzenia rozwoju aplikacji. Wykorzystując ten wzorzec również klient ma możliwość używania wielu klas zależnych tak długo jak wszystkie klasy implementują przygotowany interfejs.

Użycie kodu

Przykład wykorzystany nie miał niczego wspólnego z rzeczywistością. Aby lepiej zrozumieć ten wzorzec spróbujmy przygotować prostą aplikację, która będzie rozwiązywała problem możliwy w rzeczywistości. Załóżmy, że mamy przygotować sklep internetowy, który pozwala na dwa rodzaje płatności. Pierwsza z metod będzie nazywała się **BankOne** a druga to **BankTwo**. Pierwsza z metod pobiera dodatkowo 2% z karty kredytowej jeżeli zamówienie jest mniejsze niż 50zł oraz 1% jeżeli zamówienie jest wyższe niż 1%. Z kolei metoda druga pobiera za każdym razem 1.5% prowizji. Przystąpmy do przygotowania aplikacji. W pierwszej kolejności przygotujemy definicję naszych produktów:

```
public class Product
{
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
}
```

Kolejny krok to przygotowanie interfejsu **IPaymentGateway**, który będzie w sobie zawierał deklarację metody płatności.

```
interface IPaymentGateway
{
    void MakePayment(Product product);
}
```

W świecie rzeczywistym powinniśmy jeszcze przekazywać informacje o kliencie dokonującym zakupu. Aby uprościć przykład dane takie nie będą przekazywane.

Kolejny krok to przygotowanie klas zawierających metody do wykonania płatności bankowych:

```
public class BankOne : IPaymentGateway
{
    public void MakePayment(Product product)
    {
        // Metoda to pozwala na dokonanie płatności za pomocą pierwszego sposobu
        Console.WriteLine("Pierwszy rodzaj płatności za {0}, kwota {1}", product.Name, product.Price);
    }
}
public class BankTwo : IPaymentGateway
{
    public void MakePayment(Product product)
    {
        // Metoda to pozwala na dokonanie płatności za pomocą drugiego sposobu
        Console.WriteLine("Drugi rodzaj płatności za {0}, kwota {1}", product.Name, product.Price);
    }
}
```

Teraz przyszedł czas na utworzenie klasy fabryki, która będzie zarządzała szczegółowym tworzeniem tych obiektów. Aby być w stanie zidentyfikować, który mechanizm płatności wybrał użytkownik utworzymy prosty typ wyliczeniowy **EPaymentMethod**:

```
enum EPaymentMethod
{
    BANK_ONE,
    BANK_TWO
}
```

Klasa naszej fabryki będzie używała tego typu wyliczeniowego aby zidentyfikować, który obiekt powinien zostać utworzony. Spójrzmy na poniższy przykład:

```
public class PaymentGatewayFactory
{
    public virtual IPaymentGateway CreatePaymentGateway(EPaymentMethod method, Product prod)
    {
        IPaymentGateway gateway = null;
        switch (method)
        {
            case EPaymentMethod.BANK_ONE:
                gateway = new BankOne();
                break;
            case EPaymentMethod.BANK_TWO:
                break;
                gateway = new BankTwo();
            default:
                break;
        }
        return gateway;
    }
}
```

Czym zajmuje się powyższa klasa? Powyższa klasa przyjmuje rodzaj płatności dokonany przez użytkownika a następnie na podstawie tego wyboru tworzy konkretny obiekt.

Spójrzmy jeszcze na przykład, który pokazuje w jaki sposób kod klienta pozwala na użycie wzorca projektowego fabryki:

```
public class PaymentProcessor
{
    IPaymentGateway gateway = null;
    // Dokonywanie płatności
    // Wywołanie metody CreatePaymentGateway(...) zwraca nam obiekt utworzony
    // w zależności od wyboru rodzaju płatności przez klienta
    public void MakePayment(EPaymentMethod method, Product product)
    {
        PaymentGatewayFactory factory = new PaymentGatewayFactory();
        this.gateway = factory.CreatePaymentGateway(method, product);
        this.gateway.MakePayment(product);
    }
}
```

Możecie zauważyć, że klasa klienta nie zależy od konkretnej implementacji klasy, która jest odpowiedzialna za wykonanie opłaty za zakupiony produkt, tj. **BankOne** oraz **BankTwo**. Cała logika jest wydzielona do abstrakcji a schemat taki pokazuje użycie wzorca fabryki.

Wzorzec ten (**Factory Pattern**) jest bardzo przydatny, kiedy chcemy utrzymać kod klienta oddzielnie od klas zależnych. Pozwala to na dużo łatwiejsze zarządzanie całą aplikacją oraz jednocześnie na łatwe rozszerzanie istniejących oraz tworzenie nowych klas bez wpływu na te już istniejące oraz na kod klienta.

Poniżej całkowity przykład do przetestowania. Wszystko zostało umieszczone w jednym miejscu, jedno pod drugim przez co może być nieco nieczytelne – jest to tylko przykład testowy. W normalnym projekcie należy wydzielać klasy, interfejsy czy typy wyliczeniowe. Wpłynie to w bardzo dużym stopniu na czytelność oraz będzie dużo łatwiejsze w zarządzaniu całym kodem.

```
using System;
namespace BankExample
{
    class Program
    {
        static void Main(string[] args)
        {
            // Tworzenie instancji klasy w której znajduje się metoda do dokonania płatności
            PaymentProcessor pre = new PaymentProcessor();
            // Definiujemy produkt - to jest tylko przykład
            Product prod = new Product();
            prod.Name = "Audi RS6";
            prod.Price = 560000;
            prod.Description = "Bardzo szybkie rodzinne kombi";
            // Dokonujemy płatności pierwszym sposobem.
            pre.MakePayment(EPaymentMethod.BANK_ONE, prod);
            Console.ReadKey();
            // Wynik działania programu
            // Pierwszy rodzaj platnosci za Audi RS6, kwota 560000
        }
    }
    public enum EPaymentMethod
```

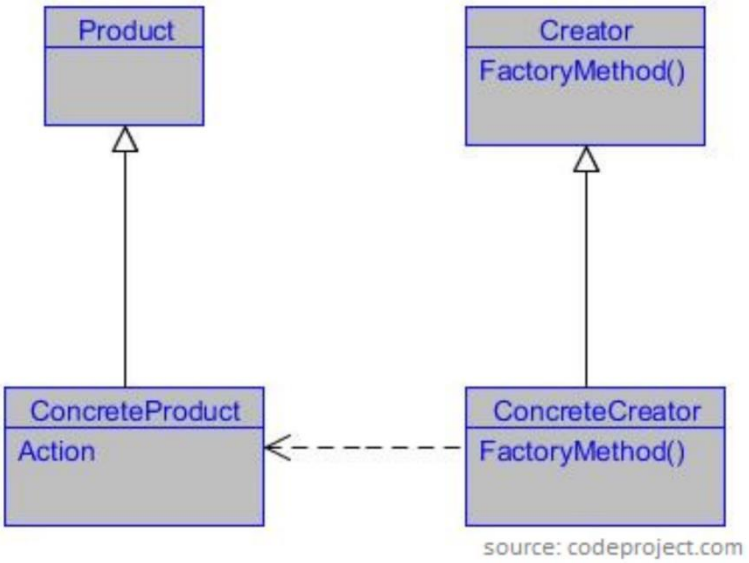
```
    {
        BANK_ONE,
        BANK_TWO
    }
    public class Product
    {
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
    }
    public interface IPaymentGateway
    {
        void MakePayment(Product product);
    }
    public class BankOne : IPaymentGateway
    {
        public void MakePayment(Product product)
        {
            // Metoda to pozwala na dokonanie płatności za pomocą pierwszego sposobu
            Console.WriteLine("Pierwszy rodzaj płatności za {0}, kwota {1}", product.Name, product.Pric
e);
        }
    }
    public class BankTwo : IPaymentGateway
    {
        public void MakePayment(Product product)
        {
            // Metoda to pozwala na dokonanie płatności za pomocą drugiego sposobu
            Console.WriteLine("Drugi rodzaj płatności za {0}, kwota {1}", product.Name, product.Price);
        }
    }
    public class PaymentGatewayFactory
    {
        public virtual IPaymentGateway CreatePaymentGateway(EPaymentMethod method, Product prod)
        {
            IPaymentGateway gateway = null;
            switch (method)
            {
                case EPaymentMethod.BANK_ONE:
                    gateway = new BankOne();
                    break;
                case EPaymentMethod.BANK_TWO:
                    break;
                    gateway = new BankTwo();
                default:
                    break;
            }
            return gateway;
        }
    }
    public class PaymentProcessor
    {
        IPaymentGateway gateway = null;
        // Dokonywanie płatności
        // Wywołanie metody CreatePaymentGateway(...) zwraca nam obiekt utworzony
        // w zależności od wyboru rodzaju płatności przez klienta
        public void MakePayment(EPaymentMethod method, Product product)
        {
            PaymentGatewayFactory factory = new PaymentGatewayFactory();
            this.gateway = factory.CreatePaymentGateway(method, product);
            this.gateway.MakePayment(product);
        }
```

```
    }
}
```

GoF (Gang of Four)

GoF definiuje metodę fabryki jako: Zdefiniuj interfejs do tworzenia obiektów, jednakże pozwól aby podklasy decydowały, którą klasę zainicjować. **Factory Method** (metoda fabryki) pozwala klasie na odłożenie inicjowania do podklasy.

Spójrzmy na poniższy diagram:



Co reprezentują powyższe klasy?

- **Product**: definiuje interfejs obiektów tworzonych metodą fabryki (**IPaymentGateway**);
- **ConcreteProduct**: implementuje interfejs Produkt (**BankOne**, **BankTwo**);
- **Creator**: deklaruje metodę fabryki, która zwraca obiektu typu **Produkt**;
- **ConcreteCreator**: przesłania metodę fabryki, aby zwrócić instancje **ConcreteProdukt**.

Jeżeli teraz porównamy naszą obecną implementację oraz implemetancję **GoF** dla metody fabryki, mamy interfejs **IPaymentGateway**, który jest interfejsem obiektu, który tworzy metoda wytwórcza. Mamy dwie klasy **BankOne** oraz **BankTwo**, które są konkretnymi produktami, tj. **ConcreteProducts**. Jak dla klas fabryki, używamy jednej klasy fabryki **PaymentGatewayFactory** zamiast tworzenia hierarchii. Jednak, gdy spojrzymy nieco bliżej zobaczymy, że nasza klasa fabryki jest w rzeczywistości klasą tworzącą wzorzec **GoF**. Jedyną różnicą jest to, że zamiast czystej abstrakcji, nasza klasa wykazuje niektóre z zachowań abstrakcji.

Jak zatem możemy włączyć oraz użyć **ConcreteCreator** do naszego projektu? Załóżmy, że chcemy utworzyć bardziej konkretne klasy pozwalające na płatności, które będą używane w innych częściach naszej aplikacji. Aby tego dokonać należy w pierwszej kolejności dodać nowe wartości do typów wyliczeniowych jak na poniższym przykładzie:

```
public enum EPaymentMethod
{
    BANK_ONE,
    BANK_TWO,
    PAYPAL,
    PRZELEWY24
}
```

Teraz możemy posiadać jedną klasę dziedziczącą po **PaymentGatewayFactory**, która będzie zawierała definicje dla nowych sposobów płatności:

```
public class PaymentGatewayFactory2 : PaymentGatewayFactory
{
    public virtual IPaymentGateway CreatePaymentGateway(EPaymentMethod method, Product prod)
    {
        IPaymentGateway gateway = null;
        switch (method)
        {
            case EPaymentMethod.PAYPAL:
                // obsługa przelewów przez system Paypal
                break;
            case EPaymentMethod.PRZELEWY24:
                // obsługa przelewów przez system Przelewy24
                break;
            default:
                // jeżeli nie realizujemy nowego sposobu płatności wywołujemy metodę bazową,
                // która obsługuje pozostałe rodzaje płatności
                base.CreatePaymentGateway(method, prod);
                break;
        }
        return gateway;
    }
}
```

Jeżeli teraz chcemy używac nowego sposobu płatności musimy jedynie utworzyć nowy obiekt **PaymentGatewayFactory2** zamiast **PaymentGatewayFactory**. Dzięki temu nasz klient będzie miał dostęp do wszystkich 4 rodzajów płatności:

```
using System;
namespace BankExampleGoF
{
    class Program
    {
        static void Main(string[] args)
        {
            // Tworzenie instancji klasy w której znajduje się metoda do dokonania płatności
            PaymentProcessor pre = new PaymentProcessor();
            // Definiujemy produkt - to jest tylko przykład
            Product prod = new Product();
            prod.Name = "Audi RS6";
            prod.Price = 560000;
            prod.Description = "Bardzo szybkie rodzinne kombi";
            // Dokonujemy płatności pierwszym sposobem.
            pre.MakePayment(EPaymentMethod.PAYPAL, prod);
            Console.ReadKey();
            // Wynik działania programu
            // Pierwszy rodzaj platnosci za Audi RS6, kwota 560000
        }
```

```
    }
    public enum EPaymentMethod
    {
        BANK_ONE,
        BANK_TWO,
        PAYPAL,
        PRZELEWY24
    }
    public class Product
    {
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
    }
    public interface IPaymentGateway
    {
        void MakePayment(Product product);
    }
    public class BankOne : IPaymentGateway
    {
        public void MakePayment(Product product)
        {
            // Metoda to pozwala na dokonanie płatności za pomocą pierwszego sposobu
            Console.WriteLine("Pierwszy rodzaj płatności za {0}, kwota {1}", product.Name, product.Pric
e);
        }
    }
    public class BankTwo : IPaymentGateway
    {
        public void MakePayment(Product product)
        {
            // Metoda to pozwala na dokonanie płatności za pomocą drugiego sposobu
            Console.WriteLine("Drugi rodzaj płatności za {0}, kwota {1}", product.Name, product.Price);
        }
    }
    // W klasie zdefiniowana jest logika obsługi starego rodzaju płatności
    public class PaymentGatewayFactory
    {
        public virtual IPaymentGateway CreatePaymentGateway(EPaymentMethod method, Product prod)
        {
            IPaymentGateway gateway = null;
            switch (method)
            {
                case EPaymentMethod.BANK_ONE:
                    gateway = new BankOne();
                    break;
                case EPaymentMethod.BANK_TWO:
                    break;
                    gateway = new BankTwo();
                default:
                    break;
            }
            return gateway;
        }
    }
    public class PaymentGatewayFactory2 : PaymentGatewayFactory
    {
        public virtual IPaymentGateway CreatePaymentGateway(EPaymentMethod method, Product prod)
        {
            IPaymentGateway gateway = null;
            switch (method)
            {
```

```
                case EPaymentMethod.PAYPAL:
                    // obsługa przelewów przez system Paypal
                    break;
                case EPaymentMethod.PRZELEWY24:
                    // obsługa przelewów przez system Przelewy24
                    break;
                default:
                    // jeżeli nie reazlizujemy nowego sposobu płątności wywołujemy metodę bazową,
                    // która obsługuje pozostałe rodzaje płatności
                    base.CreatePaymentGateway(method, prod);
                    break;
            }
            return gateway;
        }
    }
    public class PaymentProcessor
    {
        IPaymentGateway gateway = null;
        // Dokonywanie płatności
        // Wywołanie metody CreatePaymentGateway(...) zwraca nam obiekt utworzony
        // w zależności od wyboru rodzaju płatności przez klienta
        public void MakePayment(EPaymentMethod method, Product product)
        {
            PaymentGatewayFactory2 factory = new PaymentGatewayFactory2();
            this.gateway = factory.CreatePaymentGateway(method, product);
            // w przkładzie, który został przygotowany nie została przygotowana metoda do "obsługi"
            // płatności przez PayPal - w tym miejscu wyskoczy nam błąd. Aby tego uniknać należy
            // przygotować metodę jak poniżej...
            // oraz w klasie PaymentGatewayFactory2, metodzie: CreatePaymentGateway
            // dodać kod - > gateway = new Paypal();
            this.gateway.MakePayment(product);
        }
    }
    public class Paypal : IPaymentGateway
    {
        public void MakePayment(Product product)
        {
            // Metoda to pozwala na dokonanie płatności za pomocą trzeciego sposobu
            Console.WriteLine("Trzeci rodzaj płatności (PayPal) za {0}, kwota {1}", product.Name, produ
ct.Price);
        }
    }
}
```

W powyższym przykładzie nasz **Creator**, tj. metoda wytwórcza nie jest czystą klasą abstrakcyjną, gdyż dostarcza nam pewną funkcjonalność, która jednak może być przesłonięta przez konkretną fabrykę, która z niej dziedziczy.