

# Wzorce projektowe w .NET

Projektowanie zorientowane obiektowo  
z wykorzystaniem C# i F#

—

Dmitri Nesteruk

Helion 

Apress®

Tytuł oryginału: Design Patterns in .NET: Reusable Approaches in C# and F# for Object-Oriented Software Design

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-6270-3

First published in English under the title Design Patterns in .NET: Reusable Approaches in C# and F# for Object-Oriented Software Design by Dmitri Nesteruk, edition: 1

Copyright © Dmitri Nesteruk, 2019

This edition has been translated and published under licence from APress Media, LLC, part of Springer Nature.

APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the accuracy of the translation.

Polish edition copyright © 2020 by Helion SA  
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA  
ul. Kościuszki 1c, 44-100 Gliwice  
tel. 32 231 22 19, 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:  
<ftp://ftp.helion.pl/przyklady/wzprne.zip>

Drogi Czytelniku!

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

# Spis treści

	<b>O autorze</b> .....	<b>9</b>
	<b>Wprowadzenie</b> .....	<b>11</b>
<b>Część I</b>	<b>Wprowadzenie</b> .....	<b>13</b>
<b>Rozdział 1</b>	<b>Zasady projektowania SOLID</b> .....	<b>15</b>
	Zasada pojedynczej odpowiedzialności .....	15
	Zasada otwarty-zamknięty .....	17
	Zasada podstawiania Liskov .....	22
	Zasada segregacji interfejsów .....	23
	Zasada odwracania zależności .....	26
<b>Rozdział 2</b>	<b>Perspektywa funkcyjna</b> .....	<b>29</b>
	Podstawy funkcji .....	29
	Literały funkcyjne w języku C# .....	30
	Funkcje przechowywania w C# .....	31
	Literały funkcyjne w języku F# .....	33
	Kompozycja .....	34
	Cechy języka związane z paradygmatem funkcyjnym .....	35
<b>Część II</b>	<b>Wzorce kreacyjne</b> .....	<b>37</b>
<b>Rozdział 3</b>	<b>Budowniczy</b> .....	<b>39</b>
	Scenariusz .....	39
	Prosty budowniczy .....	40
	Płynny budowniczy .....	41
	Komunikowanie zamiaru .....	42
	Złożony budowniczy .....	43
	Parametry budowniczego .....	46
	Dziedziczenie płynnego interfejsu .....	47
	Konstrukcja DSL w F# .....	50
	Podsumowanie .....	51

<b>Rozdział 4</b>	<b>Fabryki</b> .....	<b>53</b>
	Scenariusz .....	53
	Metoda fabrykująca .....	54
	Fabryka .....	55
	Fabryka wewnętrzna .....	56
	Separacja logiczna .....	56
	Fabryka abstrakcyjna .....	57
	Fabryka funkcyjna .....	58
	Podsumowanie .....	59
<b>Rozdział 5</b>	<b>Prototyp</b> .....	<b>61</b>
	Kopiowanie głębokie i płytkie .....	61
	ICloneable to zły pomysł .....	62
	Głębokie kopiowanie z wykorzystaniem specjalnego interfejsu .....	63
	Głębokie kopiowanie obiektów .....	63
	Duplikacja za pomocą konstruktora kopującego .....	64
	Serializacja .....	65
	Fabryka prototypów .....	66
	Podsumowanie .....	67
<b>Rozdział 6</b>	<b>Singleton</b> .....	<b>69</b>
	Singleton według konwencji .....	69
	Klasyczna implementacja .....	70
	Leniwe ładowanie .....	71
	Kłopoty z singletonami .....	71
	Singletony a IoC .....	74
	Monostat .....	75
	Podsumowanie .....	76
<b>Część III</b>	<b>Wzorce strukturalne</b> .....	<b>77</b>
<b>Rozdział 7</b>	<b>Adapter</b> .....	<b>79</b>
	Scenariusz .....	79
	Adapter .....	80
	Tymczasowe stany adaptera .....	81
	Problem z generowaniem skrótów .....	83
	Adapter właściwości (surogat) .....	85
	Adaptory w .NET Framework .....	86
	Podsumowanie .....	87
<b>Rozdział 8</b>	<b>Most</b> .....	<b>89</b>
	Konwencjonalny most .....	89
	Most do dynamicznego prototypowania .....	92
	Podsumowanie .....	93
<b>Rozdział 9</b>	<b>Kompozyt</b> .....	<b>95</b>
	Grupowanie obiektów graficznych .....	95
	Sieci neuronowe .....	97
	Opakowanie kompozytu .....	99
	Podsumowanie .....	100

<b>Rozdział 10 Dekorator</b> .....	<b>101</b>
Niestandardowy StringBuilder .....	101
Adapter-dekorator .....	103
Wielokrotne dziedziczenie .....	103
Dynamiczna kompozycja dekoratora .....	106
Dekorator statyczny .....	108
Dekorator funkcyjny .....	109
Podsumowanie .....	110
<b>Rozdział 11 Fasada</b> .....	<b>111</b>
Budowa terminalu handlowego .....	112
Zaawansowany terminal .....	113
Gdzie jest fasada? .....	114
Podsumowanie .....	116
<b>Rozdział 12 Pyłek</b> .....	<b>117</b>
Nazwy użytkowników .....	117
Formatowanie tekstu .....	119
Podsumowanie .....	121
<b>Rozdział 13 Pełnomocnik</b> .....	<b>123</b>
Pełnomocnik zabezpieczający .....	123
Pełnomocnik właściwości .....	125
Pełnomocnik wirtualny .....	126
Pełnomocnik komunikacji .....	128
Podsumowanie .....	130
<b>Część IV Wzorce zachowań</b> .....	<b>131</b>
<b>Rozdział 14 Łańcuch odpowiedzialności</b> .....	<b>133</b>
Scenariusz .....	133
Łańcuch metod .....	134
Łańcuch brokerów .....	136
Podsumowanie .....	139
<b>Rozdział 15 Polecenie</b> .....	<b>141</b>
Scenariusz .....	141
Implementacja wzorca Polecenie .....	142
Operacje cofania .....	143
Polecenia złożone .....	145
Polecenie funkcyjne .....	147
Zapytania i rozdzielanie zapytań od poleceń .....	149
Podsumowanie .....	149
<b>Rozdział 16 Interpreter</b> .....	<b>151</b>
Ewaluator wyrażeń numerycznych .....	152
Leksykalizacja .....	152
Parsowanie .....	154
Wykorzystanie leksera i parsera .....	156

Interpreter w paradygmacie funkcyjnym .....	156
Podsumowanie .....	159
<b>Rozdział 17 Iterator .....</b>	<b>161</b>
Właściwości wspierane przez tablice .....	162
Stwórzmy iterator .....	163
Ulepszony iterator .....	166
Podsumowanie .....	167
<b>Rozdział 18 Mediator .....</b>	<b>169</b>
Chat room .....	169
Mediator ze zdarzeniami .....	172
Podsumowanie .....	174
<b>Rozdział 19 Memento .....</b>	<b>175</b>
Rachunek bankowy .....	175
Cofnij i ponów .....	176
Podsumowanie .....	178
<b>Rozdział 20 Pusty obiekt .....</b>	<b>181</b>
Scenariusz .....	181
Podejście natrętne .....	182
Pusty obiekt .....	182
Ulepszenia projektu .....	183
Wirtualny pośrednik pustego obiektu .....	183
Dynamiczny pusty obiekt .....	184
Podsumowanie .....	185
<b>Rozdział 21 Obserwator .....</b>	<b>187</b>
Słabe zdarzenie .....	188
Obserwatory właściwości .....	190
Problemy z zależnościami .....	191
Strumienie zdarzeń .....	194
Kolekcje obserwowalne .....	197
Subskrypcje deklaratywne .....	197
Podsumowanie .....	199
<b>Rozdział 22 Stan .....</b>	<b>201</b>
Przejścia między stanami zależne od stanu .....	202
Maszyna stanów — „samoróbka” .....	204
Maszyny stanów z wykorzystaniem biblioteki Stateless .....	206
Typy, akcje i ignorowanie przejść .....	206
Ponowne wejście w ten sam stan .....	207
Stany hierarchiczne .....	208
Dodatkowe własności .....	208
Podsumowanie .....	209

<b>Rozdział 23 Strategia</b> .....	<b>211</b>
Strategia dynamiczna .....	211
Strategia statyczna .....	214
Strategia funkcyjna .....	214
Podsumowanie .....	215
<b>Rozdział 24 Metoda szablonowa</b> .....	<b>217</b>
Symulacja gry .....	217
Funkcyjna odmiana Metody szablonowej .....	219
Podsumowanie .....	220
<b>Rozdział 25 Wizytator</b> .....	<b>221</b>
Nachalny wizytator .....	222
Wyświetlacz refleksywny .....	223
Funkcyjny wizytator refleksywny .....	224
Usprawnienia .....	224
Co to jest dysponowanie? .....	225
Wizytator dynamiczny .....	227
Klasyczny wizytator .....	228
Implementacja dodatkowego wizytatora .....	229
Wizytator acykliczny .....	230
Wizytator funkcyjny .....	232
Podsumowanie .....	232





## ROZDZIAŁ 3



# Budowniczy

Wzorzec Budowniczy (ang. *builder*) dotyczy tworzenia *złożonych* obiektów, to znaczy obiektów, których nie można zbudować w jednowierszowym wywołaniu konstruktora. Takie typy obiektów mogą same w sobie składać się z innych obiektów i mogą obejmować mniej czytliwą logikę wymagającą osobnego komponentu specjalnie przeznaczonego do budowy obiektów.

Chciałbym z góry zastrzec, że chociaż powiedziałem, że wzorzec Budowniczy dotyczy tworzenia złożonych obiektów, to w tym rozdziale przyjrzymy się dość trywialnemu przykładowi. Zostanie on zaprezentowany wyłącznie w celu optymalizacji miejsca na dysku, tak aby złożoność logiki dziedzicznej nie zakłócała zdolności czytelnika do oceny faktycznej implementacji wzorca.

## Scenariusz

Wyobraźmy sobie, że budujemy komponent, który renderuje strony internetowe. Strona może się składać tylko z jednego akapitu (na razie zapomnijmy o typowych pułapkach związanych z HTML). Aby ją wygenerować, prawdopodobnie napisałbyś coś takiego:

```
var hello = "witaj";
var sb = new StringBuilder();
sb.Append("<p>");
sb.Append(hello);
sb.Append("</p>");
WriteLine(sb);
```

Jest to poważna „nadinżynieria” w stylu języka Java, ale stanowi dobrą ilustrację jednego z budowniczych, który jest dostępny za pośrednictwem .NET Framework: klasy `StringBuilder`. Jest to oczywiście oddzielny komponent używany do łączenia ciągów znaków. Zawiera metody narzędziowe, takie jak `AppendLine()`, które pozwalają dołączyć zarówno tekst, jak i znak przejścia do nowego wiersza (tak, jak w wywołaniu `Environment.NewLine`). Jednak prawdziwą korzyścią z użycia obiektu `StringBuilder` jest to, że — w przeciwieństwie do operacji konkatencji ciągów znaków, która wymaga tworzenia wielu tymczasowych ciągów — `StringBuilder` po prostu alokuje bufor i zapełnia go dodawanymi ciągami.

To jednak było zbyt łatwe. Spróbujmy stworzyć prostą, nieuporządkowaną listę z dwoma elementami zawierającymi słowa *witaj* i *świecie*. Bardzo prosta implementacja może wyglądać następująco:

```
var words = new[] { "witaj", "świecie" };
sb.Clear();
sb.Append("<ul>");
foreach (var word in words)
{
    sb.AppendFormat("<li>{0}</li>", word);
}
sb.Append("</ul>");
WriteLine(sb);
```

To faktycznie daje nam to, czego chcemy, ale podejście nie jest zbyt elastyczne. W jaki sposób zmienić listę wypunktowaną na numerowaną? Jak dodać kolejny element po utworzeniu listy? Oczywiście, w tym sztywnym schemacie po zainicjowaniu obiektu `StringBuilder` nie jest to możliwe.

Możemy zatem skorzystać z paradygmatu OOP i zdefiniować klasę `HtmlElement` odpowiedzialną za przechowywanie informacji o każdym tagu:

```
class HtmlElement
{
    public string Name, Text;
    public List<HtmlElement> Elements = new List<HtmlElement>();
    private const int indentSize = 2;
    public HtmlElement() {}
    public HtmlElement(string name, string text)
    {
        Name = name;
        Text = text;
    }
}
```

Powyższa klasa modeluje pojedynczy tag HTML, który ma nazwę i może również zawierać tekst lub pewną liczbę potomków, które same są typu `HtmlElement`. Dzięki temu podejściu możemy teraz stworzyć naszą listę w bardziej sensowny sposób:

```
var words = new[] { "witaj", "świecie" };
var tag = new HtmlElement("ul", null);
foreach (var word in words)
    tag.Elements.Add("li", word);
WriteLine(tag); // wywołania tag.ToString()
```

Ten kod dobrze działa i daje bardziej kontrolowaną, bazującą na OOP reprezentację listy elementów. Proces budowania każdego obiektu `HtmlElement` nie jest jednak zbyt wygodny, szczególnie jeśli element ma potomków lub jakieś specjalne wymagania. W związku z tym przechodzimy do wzorca Budowniczy.

## Prosty budowniczy

Wzorec Budowniczy próbuje zlecić konstrukcję obiektu z części osobnej klasie. Pierwsza próba może wyglądać następująco:

```

class HtmlBuilder
{
    protected readonly string rootName;
    protected HtmlElement root = new HtmlElement();
    public HtmlBuilder(string rootName)
    {
        this.rootName = rootName;
        root.Name = rootName;
    }
    public void AddChild(string childName, string childText)
    {
        var e = new HtmlElement(childName, childText);
        root.Elements.Add(e);
    }
    public override string ToString() => root.ToString();
}

```

Jest to dedykowany komponent odpowiedzialny za budowanie elementu HTML. Konstruktor klasy `HtmlBuilder` przyjmuje argument `rootName`, który reprezentuje nazwę budowanego elementu głównego: może to być "ul", jeśli tworzymy listę nieuporządkowaną, "p", jeśli tworzymy akapit, i tak dalej. Wewnętrznie zapisujemy element główny jako obiekt `HtmlElement` i przypisujemy jego nazwę (właściwość `Name`) w konstruktorze, ale utrzymujemy także `rootName`, abyśmy mogli później, jeśli zajdzie taka potrzeba, zresetować budowniczego.

Metoda `AddChild()` służy do dodawania elementów potomnych do bieżącego elementu, przy czym każdy element potomny jest określony jako para nazwa-tekst. Można go używać w następujący sposób:

```

var builder = new HtmlBuilder("ul");
builder.AddChild("li", "witaj");
builder.AddChild("li", "świecie");
WriteLine(builder.ToString());

```

Jak można zauważyć, w tej chwili metoda `AddChild()` zwraca `void`. Jest wiele celów, do których moglibyśmy użyć zwracanej wartości. Jednym z najczęstszych zastosowań wartości zwracanej jest wsparcie dla budowania płynnego interfejsu.

## Płynny budowniczy

Zmieńmy definicję metody `AddChild()` na następującą:

```

public HtmlBuilder AddChild(string childName, string childText)
{
    var e = new HtmlElement(childName, childText);
    root.Elements.Add(e);
    return this;
}

```

Dzięki temu, że zwróciliśmy referencję do obiektu budowniczego, wywołania budowniczego można teraz połączyć w łańcuch. Nazywa się to *płynnym interfejsem* (ang. *fluent interface*):

```

var builder = new HtmlBuilder("ul");
builder.AddChild("li", "witaj").AddChild("li", "świecie");
WriteLine(builder.ToString());

```

„Jedna prosta sztuczka” polegająca na zwracaniu `this` pozwala budować interfejsy, w których kilka operacji można zmieścić w jednej instrukcji.

## Komunikowanie zamiaru

Mamy dedykowanego budowniczego zaimplementowanego w celu tworzenia elementu HTML. Jednak skąd użytkownicy naszych klas będą wiedzieć, jak go używać? Jednym z pomysłów jest zmuszenie ich do używania budowniczego za każdym razem, gdy budują obiekt. Oto, co trzeba zrobić:

```
class HtmlElement
{
    protected string Name, Text;
    protected List<HtmlElement> Elements = new List<HtmlElement>();
    protected const int indentSize = 2;
    //ukryj konstruktory!
    protected HtmlElement() {}
    protected HtmlElement(string name, string text)
    {
        Name = name;
        Text = text;
    }
    //metoda fabryczna
    public static HtmlBuilder Create(string name) => new
    HtmlBuilder(name);
}
```

Nasze podejście jest dwutorowe. Po pierwsze, ukryliśmy wszystkie konstruktory, więc nie są już dostępne. Ukryliśmy również szczegóły implementacji samego budowniczego, czego wcześniej nie robiliśmy. Stworzyliśmy jednak metodę fabryczną (jest to wzór projektowy, który omówimy później) do tworzenia budowniczego bezpośrednio z klasy `HtmlElement`. Jest to również metoda statyczna! Oto, jak można z niej skorzystać:

```
var builder = HtmlElement.Create("ul");
builder.AddChild("li", "witaj").AddChild("li", "świecie");
WriteLine(builder);
```

W tym przykładzie zmuszamy klienta do użycia statycznej metody `Create()`, ponieważ w istocie nie ma innego sposobu na skonstruowanie obiektu `HtmlElement` — wszystkie konstruktory są chronione. Tak więc klient tworzy obiekt `HtmlBuilder`, a następnie jest zmuszony do interakcji z nim podczas konstrukcji obiektu. Ostatni wiersz listingu to wyświetlenie budowanego obiektu.

Nie zapominajmy jednak, że ostatecznym celem jest zbudowanie obiektu `HtmlElement`, a do tej pory nie mamy możliwości, aby się do niego dostać! Wisienką na torcie może być implementacja niejawnego operatora `HtmlElement` w budowniczym w celu zwrócenia ostatecznej wartości:

```
protected HtmlElement root = new HtmlElement();
public static implicit operator HtmlElement(HtmlBuilder builder)
{
    return builder.root;
}
```

Dodanie operatora pozwala nam zapisać następujący kod:

```
HtmlElement root = HtmlElement
    .Create("ul")
```

```

        .AddChildFluent("li", "witaj")
        .AddChildFluent("li", "swiecie");
WriteLine(root);

```

Niestety, nie ma sposobu na jawne powiedzenie innym użytkownikom, aby korzystali z interfejsu API w ten sposób. Mamy nadzieję, że ograniczenie konstruktorów w połączeniu z obecnością statycznej funkcji `Build()` zmusi użytkownika do korzystania z budowniczego. Oprócz operatora `+=` sensowne może być również dodanie odpowiedniej metody `Build()` do klasy `HtmlBuilder`:

```
public HtmlElement Build() => root;
```

## Złożony budowniczy

Opis wzorca Budowniczy zakończymy przykładem użycia wielu konstruktorów w celu zbudowania jednego obiektu. Powiedzmy, że zdecydowaliśmy się zapisać kilka informacji o osobie:

```

public class Person
{
    //adres
    public string StreetAddress, Postcode, City;
    //informacje o zatrudnieniu
    public string CompanyName, Position;
    public int AnnualIncome;
}

```

Obiekt `Person` składa się z dwóch części: adresu i informacji o zatrudnieniu. Co zrobić, jeśli chcemy mieć dla każdego osobne konstruktory? Jak można zapewnić najwygodniejszy interfejs API? Aby to zrobić, tworzymy złożonego budowniczego. Ta konstrukcja nie jest trywialna, więc należy na nią uważać. Mimo że potrzebujemy osobnych budowniczych dla informacji o zatrudnieniu i adresie, wydzielimy nie mniej niż trzy różne klasy.

Pierwszą z nich nazwiemy `PersonBuilder`:

```

public class PersonBuilder
{
    //obiekt, który budujemy
    protected Person person; //to jest referencja!
    public PersonBuilder() => person = new Person();
    protected PersonBuilder(Person person) => this.person = person;
    public PersonAddressBuilder Lives => new
        PersonAddressBuilder(person);
    public PersonJobBuilder Works => new PersonJobBuilder(person);
    public static implicit operator Person(PersonBuilder pb)
    {
        return pb.person;
    }
}

```

Jest to podejście o wiele bardziej skomplikowane w porównaniu z zaprezentowanym wcześniej prostym budowniczym, więc omówmy po kolei każdą część.

- `person` jest referencją do budowanego obiektu. To pole jest oznaczone jako **protected**. Zostało to zrobione celowo do wykorzystania przez konstruktory pomocnicze. Warto zauważyć, że takie podejście działa tylko w odniesieniu do typów referencyjnych — gdyby `person` był strukturą, mielibyśmy niepotrzebną duplikację.

- `Lives` i `Works` to właściwości zwracające dane pomocniczych budowniczych, które osobno inicjują adres i informacje o zatrudnieniu.
- **operator** `Person` to sztuczka, której używaliśmy wcześniej.

Bardzo ważną kwestią, na którą należy zwrócić uwagę, są konstruktory. Zamiast wszędzie inicjować referencję `person` za pomocą wywołania `new Person()`, robimy to tylko w publicznym, bezparametrowym konstruktorze. Istnieje inny konstruktor, który pobiera referencję i zapisuje ją — ten konstruktor jest przeznaczony do użytku przez obiekty dziedziczące, a nie przez klienta, dlatego jest chroniony. Kod został skonfigurowany w taki sposób, że obiekt `Person` został stworzony tylko raz na każde użycie budowniczego, nawet jeśli używamy pomocniczych budowniczych.

Spójrzmy teraz na implementację klasy pomocniczego budowniczego:

```
public class PersonAddressBuilder : PersonBuilder
{
    public PersonAddressBuilder(Person person) : base(person)
    {
        this.person = person;
    }
    public PersonAddressBuilder At(string streetAddress)
    {
        person.StreetAddress = streetAddress;
        return this;
    }
    public PersonAddressBuilder WithPostcode(string postcode)
    {
        person.Postcode = postcode;
        return this;
    }
    public PersonAddressBuilder In(string city)
    {
        person.City = city;
        return this;
    }
};
```

Jak widać, klasa `PersonAddressBuilder` zapewnia płynny interfejs do budowania adresu osoby. Zauważ, że klasa ta dziedziczy po klasie `PersonBuilder` (co oznacza, że uzyskała funkcje członkowskie `Lives` i `Works`). Ma konstruktor, który pobiera i przechowuje referencję do konstruowanego obiektu, więc kiedy używamy tych pomocniczych konstruktorów, zawsze pracujemy tylko z jednym egzemplarzem klasy `Person`. Teraz przy okazji inicjuje ona wiele egzemplarzy. Kluczowe znaczenie ma to, aby był wywoływany konstruktor klasy bazowej — jeśli nie zostanie wywołany, to konstruktor pomocniczy automatycznie wywoła konstruktor bez parametrów, co spowoduje niepotrzebne utworzenie dodatkowych egzemplarzy klasy `Person`.

Jak można się domyślić, klasa `PersonJobBuilder` jest implementowana w identyczny sposób, więc tutaj ją pominiemy.

Teraz nadchodzi czas, na który czekałeś: praktyczny przykład użycia budowniczych.

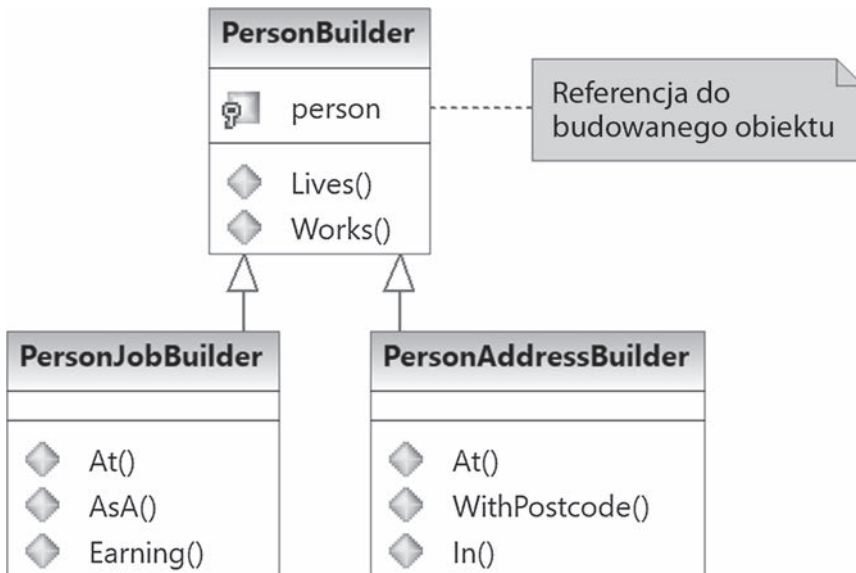
```
var pb = new PersonBuilder();
Person person = pb
    .Lives
    .At("Aleja Gdańska 123")
    .In("Gdańsk")
```

```

        .WithPostcode("82200")
    .Works
        .At("Fabryka Maszyn")
        .AsA("inżynier")
        .Earning(123000);
WriteLine(person);
// StreetAddress: Aleja Gdańska 123, Postcode: 82200, City: Malbork,
// CompanyName: Fabryka Maszyn, Position: inżynier, AnnualIncome: 123000

```

Łatwo zauważyć, co się tutaj dzieje? Tworzymy budowniczego, a następnie używamy właściwości `Lives`, aby uzyskać obiekt `PersonAddressBuilder`, ale po zakończeniu inicjalizacji informacji adresowych po prostu wywołujemy `Works` i przełączamy się na użycie obiektu `PersonJobBuilder`. Na wszelki wypadek potrzebujemy wizualnej ilustracji tego, co właśnie zrobiliśmy. Nie jest to skomplikowane, co pokazano na rysunku 3.1.



**Rysunek 3.1.** Reprezentacja w języku UML abstrakcyjnego budowniczego i dwóch budowniczych pomocniczych

Po zakończeniu procesu budowania używamy tej samej sztuczki w celu przekształcenia budowanego obiektu w obiekt klasy `Person`.

Podejście to ma jedną oczywistą wadę: nie można go rozszerzać. Mówiąc ogólnie, klasa bazowa nie powinna być świadoma własnych podklas, a tutaj dokładnie tak się dzieje: klasa `PersonBuilder` jest świadoma swoich potomków, które ujawnia za pomocą specjalnych interfejsów API. Aby stworzyć dodatkowego pomocniczego budowniczego (powiedzmy, `PersonEarningsBuilder`), musiałbyś złamać zasadę OCP i bezpośrednio edytować klasę `PersonBuilder`. Aby dodać element interfejsu, nie można po prostu stworzyć klasy potomnej.

## Parametry budowniczego

Jak pokazałem, jedynym sposobem na zmuszenie klienta do korzystania z budowniczego zamiast bezpośredniego budowania obiektu jest uczynienie konstruktorów obiektu niedostępnymi. Są jednak sytuacje, gdy chcemy jawnie zmusić użytkownika do interakcji z budowniczym.

Zalóżmy na przykład, że mamy interfejs API do wysyłania wiadomości e-mail, w którym każda wiadomość e-mail jest opisana wewnętrznie w następujący sposób:

```
public class Email
{
    public string From, To, Subject, Body;
    //tutaj inne składowe
}
```

Zauważ, że powiedziałem tutaj *wewnętrznie* — nie chcemy pozwolić użytkownikowi na interakcję z tą klasą, być może dlatego, że są w niej zapisane informacje o dodatkowych usługach. Upublicznianie ich jest jednak w porządku, pod warunkiem że nie ujawnimy użytkownikowi żadnego interfejsu API, który umożliwi klientowi bezpośrednie wysłanie wiadomości e-mail. Niektóre części wiadomości e-mail (np. Subject — dosł. temat) są opcjonalne, więc obiekt nie musi być w pełni wyspecyfikowany.

Zdecydowaliśmy o zaimplementowaniu płynnego budowniczego, który będzie wykorzystywany do konstruowania wiadomości e-mail „za kulisami”. Może on wyglądać następująco:

```
public class EmailBuilder
{
    private readonly Email email;
    public EmailBuilder(Email email) => this.email = email;
    public EmailBuilder From(string from)
    {
        email.From = from;
        return this;
    }
    //tutaj inne płynne składowe
}
```

Teraz, aby zmusić klienta do używania budowniczego tylko do wysyłania wiadomości e-mail, możesz zaimplementować usługę MailService w następujący sposób:

```
public class MailService
{
    public class EmailBuilder { ... }
    private void SendEmailInternal(Email email) {}
    public void SendEmail(Action<EmailBuilder> builder)
    {
        var email = new Email();
        builder(new EmailBuilder(email));
        SendEmailInternal(email);
    }
}
```

Jak widać, metoda SendEmail(), z której mają korzystać klienci, przyjmuje funkcję, a nie tylko zestaw parametrów lub wstępnie przygotowany obiekt. Ta funkcja pobiera obiekt EmailBuilder, a następnie oczekuje użycia budowniczego w celu zbudowania treści wiadomości. Po wykonaniu tej



czynności korzystamy z wewnętrznej mechaniki klasy `MailService` w celu przetworzenia w pełni zainicjowanej wiadomości e-mail.

Jak można zauważyć, w powyższym kodzie zastosowano sprytny podstęp: zamiast przechowywać wewnętrzne referencje do wiadomości e-mail, budowniczy otrzymuje tę referencję w argumencie konstruktora. Zaimplementowaliśmy to w taki sposób, aby obiekt `EmailBuilder` nie musiał publicznie ujawniać wiadomości e-mail w dowolnym miejscu interfejsu API.

Oto, jak wygląda użycie tego interfejsu API z perspektywy klienta:

```
var ms = new MailService();
ms.SendEmail(email => email.From("foo@bar.com")
    .To("bar@baz.com")
    .Body("Cześć, jak się masz?"));
```

Krótko mówiąc, podejście „budowniczy z parametrem” zmusza użytkowników interfejsu API do korzystania z budowniczego, czy to im się podoba, czy nie.

## Dziedziczenie płynnego interfejsu

Jednym z interesujących problemów, które wpływają nie tylko na płynnego budowniczego, ale także na dowolną klasę z płynnym interfejsem, jest problem dziedziczenia. Czy płynny budowniczy może (i czy jest to realistyczne) odziedziczyć po innym płynnym budowniczym? Tak, ale nie jest to łatwe. Oto problem. Załóżmy, że zaczynamy od następującego (bardzo trywialnego) obiektu, który chcemy zbudować:

```
public class Person
{
    public string Name;
    public string Position;
}
```

Stworzyliśmy bazową klasę budowniczego, która umożliwiła konstruowanie obiektów `Person`:

```
public abstract class PersonBuilder
{
    protected Person person = new Person();
    public Person Build()
    {
        return person;
    }
}
```

Następnie utworzyliśmy dedykowaną klasę do określenia nazwiska osoby:

```
public class PersonInfoBuilder : PersonBuilder
{
    public PersonInfoBuilder Called(string name)
    {
        person.Name = name;
        return this;
    }
}
```

To działa i nie ma z tym absolutnie żadnego problemu. Przypuśćmy teraz, że zdecydowaliśmy się stworzyć podklasę `PersonInfoBuilder`, aby podać także informacje o zatrudnieniu. Możemy napisać następujący kod:

```
public class PersonJobBuilder : PersonInfoBuilder
{
    public PersonJobBuilder WorksAsA(string position)
    {
        person.Position = position;
        return this;
    }
}
```

Niestety, zepsuliśmy teraz płynny interfejs i sprawiliśmy, że cała konfiguracja nie nadaje się do użytku:

```
var me = Person.New
    .Called("Dmitri")
    .WorksAsA("Klasyfikator") // nie skompiluje się
    .Build();
```

Dlaczego ten kod się nie skompiluje? To proste: funkcja `Called()` zwraca referencję `this`, która reprezentuje obiekt typu `PersonInfoBuilder`. Ten obiekt po prostu nie ma metody `WorksAsA()`!

Może się wydawać, że sytuacja jest beznadziejna, ale tak nie jest: możemy zaprojektować płynne interfejsy API z uwzględnieniem dziedziczenia, ale jest to dość skomplikowane. Przyjrzyjmy się, co wiąże się ze zmianą projektu klasy `PersonInfoBuilder`. Oto jej nowe wcielenie:

```
public class PersonInfoBuilder<SELF> : PersonBuilder
where SELF : PersonInfoBuilder<SELF>
{
    public SELF Called(string name)
    {
        person.Name = name;
        return (SELF) this;
    }
}
```

Co się tu stało? Ogólnie rzecz biorąc, wprowadziliśmy nowy ogólny argument `SELF`. Co bardziej interesujące, ten argument `SELF` jest określony jako dziedziczący po komponencie `PersonInfoBuilder<SELF>`. Innymi słowy, wymagany jest szablon klasy dla dziedziczenia po tej właśnie klasie. Może to wydawać się szaleństwem, ale w rzeczywistości jest to bardzo popularna sztuczka dla dziedziczenia w C# w stylu CRTP (*Curiously Recurring Template Pattern* to technika z C++). W gruncie rzeczy wymuszamy łańcuch dziedziczenia: mówimy, że `Foo<Bar>` jest jedynie akceptowalną specjalizacją, jeśli `Foo` wywodzi się z `Bar`, a dla wszystkich innych przypadków ograniczenie `where` nie będzie spełnione.

Największym problemem w dziedziczeniu płynnego interfejsu jest możliwość zwrócenia referencji `this`, która ma typ klasy, w której się aktualnie znajdujesz, nawet jeśli wywołujesz metodę członkowską płynnego interfejsu klasy bazowej. Jedynym sposobem na skuteczne propagowanie tego zachowania jest istnienie generycznego parametru (`SELF`), który określa całą hierarchię dziedziczenia.

Aby to docenić, musimy spojrzeć również na klasę `PersonJobBuilder`:

```
public class PersonJobBuilder<SELF>
    : PersonInfoBuilder<PersonJobBuilder<SELF>>
where SELF : PersonJobBuilder<SELF>
{
```

```

public SELF WorksAsA(string position)
{
    person.Position = position;
    return (SELF) this;
}
}

```

Spójrz na klasę bazową! To nie jest po prostu klasa `PersonInfoBuilder`, tak jak poprzednio. Zamiast tego jest to klasa `PersonInfoBuilder<PersonJobBuilder<SELF>>`!

W związku z tym, kiedy dziedziczymy po klasie `PersonInfoBuilder`, ustawiamy argument `SELF` na obiekt `PersonJobBuilder<SELF>`. Dzięki temu wszystkie jej płynne interfejsy mogą zwracać poprawny typ, a nie typ klasy posiadającej.

Czy to ma sens? Jeśli nie, nie spiesz się i jeszcze raz przejrzyj kod źródłowy. Pozwól sprawdzić, jak to rozumiesz. Załóżmy, że wprowadziłem inny element członkowski o nazwie `DateOfBirth` i odpowiadającego mu budowniczego `PersonDateOfBirthBuilder`. Z jakiej klasy on odziedziczy?

Jeśli odpowiedziałeś

```
PersonInfoBuilder<PersonJobBuilder<PersonBirthDateBuilder<SELF>>>
```

to się mylisz, ale nie mogę Cię winić za tę próbę. Pomyśl o tym: `PersonJobBuilder` jest już typu `PersonInfoBuilder`, więc tej informacji nie trzeba jeszcze raz jawnie formułować w ramach listy typów dziedziczenia. Zamiast tego należy zdefiniować budowniczego w następujący sposób:

```

public class PersonBirthDateBuilder<SELF>
    : PersonJobBuilder<PersonBirthDateBuilder<SELF>>
    where SELF : PersonBirthDateBuilder<SELF>
{
    public SELF Born(DateTime dateOfBirth)
    {
        person.DateOfBirth = dateOfBirth;
        return (SELF) this;
    }
}

```

Ostatnie pytanie, jakie mamy, brzmi: w jaki sposób konstruujemy takiego budowniczego, biorąc pod uwagę, że on zawsze pobiera generyczny argument? Obawiam się, że teraz potrzebujemy nowego typu, a nie tylko zmiennej. Zatem na przykład implementacja `Person.New` (właściwość rozpoczynająca proces konstruowania) może zostać zaimplementowana w następujący sposób:

```

public class Person
{
    public class Builder : PersonJobBuilder<Builder>
    {
        internal Builder() {}
    }
    public static Builder New => new Builder();
    //inne składowe pominięto
}

```

Jest to prawdopodobnie najbardziej denerwujący szczegół implementacji: fakt, że aby go użyć, trzeba stworzyć niegeneryczny obiekt dziedziczący rekurencyjnego typu generycznego.

## Konstrukcja DSL w F#

Wiele języków programowania (np. Groovy, Kotlin lub F #) stara się wprowadzić funkcję języka, która uprości proces tworzenia języków dziedzinowych (ang. *domain-specific languages* — DSL) — niewielkich języków, które pomagają opisać określoną dziedzinę problemu. Wiele aplikacji takich osadzonych DSL jest używanych do implementacji wzorca Budowniczy. Na przykład, aby zbudować stronę HTML, nie trzeba bezpośrednio majstrować z klasami i metodami. Zamiast tego możemy napisać coś, co jest bardzo zbliżone do HTML, bezpośrednio w kodzie!

Jest to możliwe w języku F # przy użyciu list składanych: mechanizmu definiowania list bez żadnych wyraźnych wywołań metod budowniczego. Na przykład, żeby obsługiwać akapity i obrazy HTML, moglibyśmy zdefiniować następujące funkcje budowniczego:

```
let p args =
    let allArgs = args |> String.concat "\n"
    [" <p>"; allArgs; "</p>"] |> String.concat "\n"
let img url = "<img src=\"\" + url + \"\"/>"
```

Zwróćmy uwagę, że znacznik `img` ma tylko jeden parametr tekstowy, a znacznik `<p>` akceptuje sekwencję `args`, dzięki czemu może zawierać dowolną liczbę wewnętrznych elementów HTML, w tym zwykły tekst. Możemy zatem zbudować akapit zawierający zarówno tekst, jak i obraz:

```
let html =
    p[
        "Zobacz to zdjęcie";
        img "pokemon.com/pikachu.png"
    ]
printfn "%s" html
```

Ten kod zwraca następujący wynik:

```
<p>
Zobacz to zdjęcie

</p>
```

Takie podejście jest stosowane we frameworkach webowych, takich jak WebSharper. Istnieje wiele odmian tego podejścia. Jednym z nich jest stosowanie typów rekordowych (zezwalanie na używanie nawiasów klamrowych zamiast list), niestandardowych operatorów do wprowadzania zwykłego tekstu i nie tylko<sup>1</sup>.

Należy zapamiętać, że takie podejście jest wygodne tylko w przypadku pracy z niezmienną strukturą pozwalającą tylko na dołączanie. Kiedy zaczniemy korzystać z obiektów mutowalnych (np. z DSL do skonstruowania definicji dokumentu Microsoft Project), ostatecznie wrócimy do OOP. Oczywiście, końcowa składnia DSL jest nadal bardzo wygodna w użyciu, ale instalacja dodatkowa wymagana do jej działania nie jest zgrabna.

<sup>1</sup> Przykład można znaleźć w artykule Tomasa Petriceka *DSL for constructing HTML* pod adresem: <http://fssnip.net/hf>.

# Podsumowanie

Celem wzorca Budowniczy jest zdefiniowanie komponentu poświęconego całkowicie budowie części złożonego obiektu lub zestawu obiektów. Zaprezentowaliśmy następujące kluczowe cechy budowniczego:

- Obiekty budowniczych mogą mieć płynny interfejs, który można wykorzystać do skomplikowanej budowy obiektu za pomocą pojedynczego łańcucha wywołań. Aby to wesprzeć, funkcje budowniczego powinny zwracać **this**.
- Aby zmusić użytkownika interfejsu API do użycia budowniczego, możemy sprawić, że konstruktory obiektu docelowego będą niedostępne, a następnie zdefiniować statyczną funkcję `Create()` zwracającą budowniczego (wybór konkretnej nazwy należy do Ciebie; możesz nazwać ją `Make()`, `New()` lub w jakikolwiek inny sposób).
- Budowniczego można skonwertować na sam obiekt poprzez zdefiniowanie odpowiedniego niejawnego operatora konwersji.
- Możesz zmusić klienta do używania budowniczego, podając go w ramach funkcji parametru.
- Pojedynczy interfejs budowniczego może udostępniać wiele pomocniczych budowniczych. Dzięki inteligentnemu użyciu dziedziczenia i płynnych interfejsów można łatwo przejść od jednego budowniczego do innego.
- Dziedziczenie płynnych interfejsów (nie tylko na potrzeby budowniczych) jest możliwe dzięki rekurencyjnym typom generycznym.

Dla przypomnienia tego, o czym wspomniano wcześniej, użycie wzorca Budowniczy ma sens, gdy konstrukcja obiektu jest procesem *nietrywialnym*. Proste obiekty, które są jednoznacznie zbudowane z ograniczonej liczby rozsądnie nazwanych parametrów konstruktora, prawdopodobnie powinny korzystać z konstruktora (lub wstrzykiwania zależności) bez konieczności używania budowniczego jako takiego.



# PROGRAM PARTNERSKI

— GRUPY HELION —

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

GRUPA  
**Helion** 

# Wzorce projektowe w .NET: rekomendacja najlepszych architektów!

Wzorce projektowe są bardzo przydatnym narzędziem w przyborniku programisty. Pozwalają na szybkie opracowanie złożonych zagadnień, ale można je również potraktować jako wstęp do ciekawego i inspirującego dochodzenia, jak rozwiązać konkretny problem na wiele różnych sposobów, na różnych poziomach zaawansowania technicznego i z zastosowaniem różnego rodzaju kompromisów. Takie próby jednak często prowadzą do nadinżynierii lub powstawania zbyt skomplikowanych struktur i mechanizmów. Chociaż bywa to zabawne i pomaga w doskonaleniu umiejętności programistycznych, nie jest pożądanym sposobem tworzenia systemów produkcyjnych.

To książka przeznaczona dla programistów C#, którzy chcą poszerzyć wiedzę na temat sztuki programowania dzięki wykorzystaniu nowoczesnych technik projektowych do rozwiązywania konkretnych problemów programistycznych w optymalny sposób. Dogłębnie przedstawiono tu implementację klasycznych wzorców wraz ze wskazówkami dotyczącymi ich możliwości. Omówiono znaczenie poszczególnych cech języków C# i F# dla implementacji wzorców. Pokazano cały szereg przykładów i scenariuszy, możliwych implementacji wzorców, ich alternatyw i wzajemnych relacji. Co więcej, zaprezentowano sposób wykorzystania dedykowanego narzędzia do refaktoryzacji (ReSharper) do łatwej implementacji wzorców projektowych.

W tej książce między innymi:

- zasady projektowania SOLID
- cechy C# i F# związane z paradygmatem funkcyjnym
- kreatywne wzorce projektowe
- praca w środowisku Visual Studio

**Dmitri Nesteruk** jest analitykiem giełdowym i programistą. Występuje na konferencjach, tworzy kursy i pisze książki techniczne. Zawodowo interesuje się integracją rozwiązań w dziedzinie obliczeń, finansów i handlu algorytmicznego. Z upodobaniem programuje w C# i C++ i implementuje wysokowydajne przetwarzanie danych za pomocą takich technologii jak CUDA oraz FPGA. W 2009 roku za osiągnięcia w dziedzinie C# otrzymał tytuł MVP.

<b>Helion</b> ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	<i>Sprawdź nasze szkolenia!</i> SZKOLENIA  AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL	<b>KOD KORZYŚCI</b> Sięgnij po więcej! ▶  ISBN 978-83-283-6270-3  9 788328 362703
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 57,00 zł

Apress®