Wzorce projektowe w .NET Core 3

Projektowanie zorientowane obiektowo z wykorzystaniem C# i F#

—

Dmitri Nesteruk

Apress®

Tytuł oryginału: Design Patterns in .NET Core 3: Reusable Approaches in C# and F# for Object-Oriented Software Design

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-8137-7

First published in English under the title Design Patterns in .NET Core 3: Reusable Approaches in C# and F# for Object-Oriented Software Design by Dmitri Nesteruk, edition: 2

Copyright © Dmitri Nesteruk, 2020

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 © 2021 by Helion S.A.
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 jakiejkolwiek 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 wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/wzprnc.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/wzprnc
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

O autorze 9

O recenzencie technicznym 10

Wprowadzenie 11

Część I **Wprowadzenie** 15

Rozdział 1. **Zasady projektowania SOLID** 17

Zasada pojedynczej odpowiedzialności 17
Zasada otwarty-zamknięty 19
Zasada podstawiania Liskov 24
Zasada segregacji interfejsów 26
Zasada odwracania zależności 29

Rozdział 2. **Perspektywa funkcyjna** 33

Podstawy funkcji 33
Literały funkcyjne w języku C# 34
Funkcje przechowywania w C# 35
Literały funkcyjne w języku F# 37
Kompozycja 38
Cechy języka związane z paradygmatem funkcyjnym 39

Kup ksi k   Pole ksi k

Część II Wzorce kreacyjne 41

Rozdział 3. Budowniczy 43
Scenariusz 43
Prosty budowniczy 44
Płynny budowniczy 45
Komunikowanie zamiaru 46
Złożony budowniczy 47
Parametry budowniczego 49
Rozszerzanie budowniczego z wykorzystaniem rekurencyjnych typów generycznych 51
Leniwy, funkcyjny budowniczy 54
Konstrukcja DSL w F# 56
Podsumowanie 57

Rozdział 4. Fabryki 59
Scenariusz 59
Metoda wytwórcza 60
Asynchroniczna metoda wytwórcza 61
Fabryka 62
Fabryka wewnętrzna 62
Fabryka abstrakcyjna 64
Fabryki-delegaty w IoC 66
Fabryka funkcyjna 67
Podsumowanie 68

Rozdział 5. Prototyp 69
Kopiowanie głębokie i płytkie 69
ICloneable to zły pomysł 70
Głębokie kopiowanie z wykorzystaniem specjalnego interfejsu 71
Głębokie kopiowanie obiektów 71
Duplikacja za pomocą konstruktora kopiującego 72
Serializacja 73
Fabryka prototypów 74
Podsumowanie 75

Rozdział 6. Singleton 77
Singleton według konwencji 77
Klasyczna implementacja 78
Podsumowanie 84

Kup ksi k   Pole ksi k

Część III Wzorce strukturalne 85

Rozdział 7. Adapter 87
Scenariusz 87
Adapter 88
Tymczasowe stany adaptera 90
Problem z generowaniem skrótów 92
Adapter właściwości (surogat) 93
Adapter generycznych wartości 95
Adapter a wstrzykiwanie zależności 100
Adaptery w .NET Framework 102
Podsumowanie 103

Rozdział 8. Most 105
Konwencjonalny most 105
Most do dynamicznego prototypowania 108
Podsumowanie 110

Rozdział 9. Kompozyt 111
Grupowanie obiektów graficznych 111
Sieci neuronowe 113
Opakowanie kompozytu 115
Specyfikacja kompozytu 117
Podsumowanie 117

Rozdział 10. Dekorator 119
Niestandardowy StringBuilder 119
Adapter-dekorator 121
Wielokrotne dziedziczenie z wykorzystaniem interfejsów 121
Wielokrotne dziedziczenie z domyślnymi składowymi interfejsu 124
Dynamiczna kompozycja dekoratora 125
Kompozycja dekoratora statycznego 126
Dekorator funkcyjny 128
Podsumowanie 128

Rozdział 11. Fasada 131
Kwadraty magiczne 132
Budowa terminalu handlowego 135
Zaawansowany terminal 136
Gdzie jest fasada? 137
Podsumowanie 138

Kup ksi k Pole ksi k

Rozdział 12. Pyłek ... 139
Nazwy użytkowników ... 139
Formatowanie tekstu ... 141
Podsumowanie ... 143

Rozdział 13. Pełnomocnik ... 145
Pełnomocnik zabezpieczający ... 145
Pełnomocnik właściwości ... 147
Pełnomocnik wartości ... 148
Pełnomocnik kompozytu: SoA/AoS ... 150
Pełnomocnik kompozytu z właściwościami przechowywanymi w tablicy ... 152
Pełnomocnik wirtualny ... 153
Pełnomocnik komunikacji ... 155
Dynamiczny pełnomocnik do logowania ... 156
Podsumowanie ... 158

Część IV Wzorce zachowań ... 161

Rozdział 14. Łańcuch odpowiedzialności ... 163
Scenariusz ... 163
Łańcuch metod ... 164
Łańcuch brokerów ... 166
Podsumowanie ... 169

Rozdział 15. Polecenie ... 171
Scenariusz ... 171
Podsumowanie ... 180

Rozdział 16. Interpreter ... 181
Ewaluator wyrażeń numerycznych ... 182
Interpreter w paradygmacie funkcyjnym ... 186
Podsumowanie ... 189

Rozdział 17. Iterator ... 191
Właściwości wspierane przez tablice ... 192
Stwórzmy iterator ... 194
Ulepszony iterator ... 196
Adapter iteratora ... 197
Podsumowanie ... 198

Kup ksi k Pole ksi k

Rozdział 18. Mediator ... 199
Chat room ... 199
Mediator ze zdarzeniami ... 202
Wprowadzenie do biblioteki MediatR ... 205
Podsumowanie ... 207

Rozdział 19. Memento ... 209
Rachunek bankowy ... 209
Cofnij i ponów ... 210
Wykorzystanie wzorca Memento do interakcji z kodem niezarządzanym ... 212
Podsumowanie ... 213

Rozdział 20. Pusty obiekt ... 215
Scenariusz ... 215
Podejście natrętne ... 216
Pusty obiekt ... 217
Dynamiczny pusty obiekt ... 218
Podsumowanie ... 219

Rozdział 21. Obserwator ... 221
Słabe zdarzenie ... 222
Strumienie zdarzeń ... 224
Obserwatory właściwości ... 226
Kolekcje obserwowalne ... 235
Subskrypcje deklaratywne w Autofac ... 236
Podsumowanie ... 239

Rozdział 22. Stan ... 241
Przejścia między stanami zależne od stanu ... 242
Maszyna stanów — „samoróbka” ... 243
Maszyna stanów na bazie instrukcji switch ... 246
Kodowanie tranzycji za pomocą wyrażeń instrukcji switch ... 247
Maszyny stanów z wykorzystaniem biblioteki Stateless ... 248
Podsumowanie ... 252

Rozdział 23. Strategia ... 253
Strategia dynamiczna ... 253
Strategia statyczna ... 256
Strategie równości i porównywania ... 256
Strategia funkcyjna ... 258
Podsumowanie ... 259

Kup ksi k

Pole ksi k

Rozdział 24. Metoda szablonowa 261
Symulacja gry 261
Funkcyjna odmiana Metody szablonowej 263
Podsumowanie 264

Rozdział 25. Wizytator 265
Nachalny wizytator 266
Wyświetlacz reflektywny 267
Co to jest dysponowanie? 271
Wizytator dynamiczny 272
Klasyczny wizytator 273
Wizytator acykliczny 276
Wizytator funkcyjny 278
Podsumowanie 279

Kup ksi k
Pole ksi k

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 oczywistą 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, tak aby złożoność logiki dziedzinowej 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 konkatenacji ciągów znaków, która wymaga tworzenia wielu tymczasowych ciągów — `StringBuilder` po prostu alokuje bufor i zapełnia go dodawanymi ciągami.

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

Kup ksi k	Pole ksi k

```
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

Wzorzec 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();
```


Kup ksi k Pole ksi k

```
    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. Zauważ, że klasa `StringBuilder` także udostępnia płynny interfejs. Płynne interfejsy są na ogół ładne, ale tworzenie dekoratorów, które ich używają

Kup ksi k   Pole ksi k

(np. z wykorzystaniem automatycznych narzędzi, takich jak ReSharper lub Rider), może być problemem — opowiem o tym później.

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 wytwórcza
    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 wytwórczą (jest to wzorzec 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 (oznaczone modyfikatorem `protected`). 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:

Kup ksi k   Pole ksi k

```
HtmlElement root = HtmlElement
    .Create("ul")
    .AddChildFluent("li", "witaj")
    .AddChildFluent("li", "świecie");
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.

- Pole `person` jest referencją do budowanego obiektu. Jest ono oznaczone jako **`protected`**. Zostało to zrobione celowo do wykorzystania przez pomocniczych budowniczych. Warto

Kup ksi k Pole ksi k

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.
- **Konstrukcja** **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. Dzięki temu unikamy przypadkowego tworzenia wielu 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.

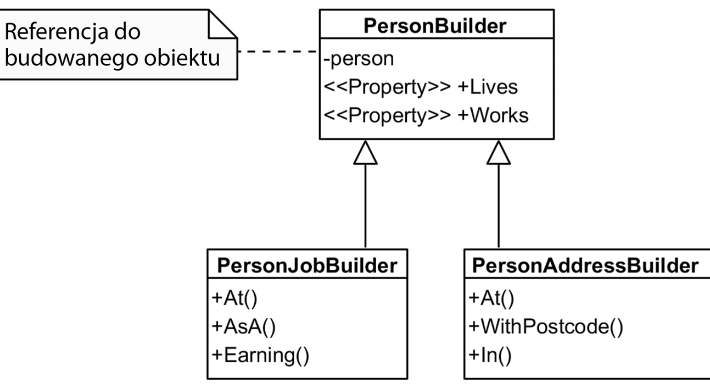
Teraz nadchodzi czas, na który czekałeś: praktyczny przykład użycia budowniczych.

```
var pb = new PersonBuilder();
Person person = pb
```


Kup ksi k Pole ksi k

```
    .Lives
        .At("Aleja Gdańska 123")
        .In("Malbork")
        .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`. Alternatywne podejście polega na wywołaniu metody `Build()`.

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, czasami nawet ukrywając przed nim faktycznie budowany obiekt.

Kup ksi k   Pole ksi k

Załóżmy 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 klienty, 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.

Kup ksi k Pole ksi k

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. Dzięki zastosowaniu sztuczki z delegatem Action klient może uzyskać dostęp do już zainicjowanego obiektu budowniczego.

Rozszerzanie budowniczego z wykorzystaniem rekurencyjnych typów generycznych

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 umożliwia 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:

Kup ksi k   Pole ksi k

```
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 generyczny argument `SELF`. Co bardziej interesujące, ten argument `SELF` jest określony jako dziedziczący po komponencie `PersonInfoBuilder<SELF>`. Innymi słowy, wymagany jest generyczny argument klasy w celu dziedziczenia po tej właśnie klasie. Może to wydawać się szaleństwem, ale w rzeczywistości jest to bardzo popularna sztuczka w celu uzyskania w C# dziedziczenia w stylu CRTP[1]. 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>
{
```

[1] CRTP — *Curiously Recurring Template Pattern* to popularna technika stosowana w języku C++ implementowana za pomocą kodu `class Foo<T> : T`. Innymi słowy jest to dziedziczenie po generycznym parametrze. W języku C# jest to niemożliwe.

Kup ksi k Pole ksi k

```
    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.

Podsumowując, możesz teraz korzystać z budowniczego, stosując wszystkie metody w łańcuchu dziedziczenia:

Kup ksi k Pole ksi k

```
var builder = Person.New
    .Called("Natasha")
    .WorksAsA("Doctor")
    .Born(new DateTime(1981, 1, 1));
```

Leniwy, funkcyjny budowniczy

Poprzedni przykład oparty na korzystaniu z rekurencyjnych typów generycznych wymaga wiele pracy. Powstaje więc pytanie: czy należy używać dziedziczenia do rozszerzania budowniczych? W końcu moglibyśmy zamiast tego użyć metod rozszerzających.

Jeśli zastosujemy podejście funkcyjne, implementacja stanie się znacznie prostsza — nie będzie wymagać zastosowania rekurencyjnych typów generycznych. Spróbujmy po raz kolejny zbudować klasę `Person` zdefiniowaną w następujący sposób:

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

Tym razem zdefiniujemy leniwego budowniczego, który konstruuje obiekt tylko wtedy, gdy zostanie wywołana jego metoda `Build()`. Do tego czasu obiekt będzie jedynie przechowywał listę działań, które należy wykonać, gdy będzie budowany:

```
public sealed class PersonBuilder
{
    private readonly List<Func<Person, Person>> actions =
    new List<Func<Person, Person>>();
    public PersonBuilder Do(Action<Person> action)
    => AddAction(action);
    public Person Build()
    => actions.Aggregate(new Person(), (p, f) => f(p));
    private PersonBuilder AddAction(Action<Person> action)
    {
        actions.Add(p => { action(p); return p; });
        return this;
    }

}
```

Pomysł jest prosty: zamiast utrzymywania mutowalnego „obiektu w budowie”, który jest modyfikowany bezpośrednio po wywołaniu każdej metody, wystarczy przechowywać listę działań, które muszą zostać wykonane na obiekcie, gdy ktoś wywoła metodę `Build()`. W tej implementacji istnieją jednak dodatkowe komplikacje.

Pierwsza polega na tym, że działanie wykonywane na osobie, choć jest pobierane za pomocą parametru `Action<T>`, faktycznie jest przechowywane jako `Func<T, T>`. Chodzi o to, że dzięki dostarczaniu tego płynnego interfejsu umożliwiamy prawidłowe działanie wywołania `Aggregate()` wewnątrz metody `Build()`.Oczywiście moglibyśmy zamiast tego użyć dobrej, staromodnej metody `ForEach()`.

Druga komplikacja polega na tym, że aby było możliwe rozszerzanie klasy zgodnie z zasadą OCP, w gruncie rzeczy nie powinniśmy udostępniać operacji jako publicznych składowych. Takie postępowanie pozwoliłoby bowiem na wykonywanie zbyt wielu operacji na elementach listy

Kup ksi k Pole ksi k

(np. nieograniczonego usuwania), które niekoniecznie chcemy udostępniać wszystkim klientom, rozszerzającym tego budowniczego w przyszłości. Zamiast tego publicznie udostępniamy tylko jedną operację `Do()`, która pozwala określić działanie do wykonania na budowanym obiekcie. Następnie to działanie dodajemy do ogólnego zbioru działań.

W ramach tego paradygmatu możesz teraz przekazać do tego konstruktora konkretną metodę podawania nazwiska danej osoby:

```
public PersonBuilder Called(string name)
    => Do(p => p.Name = name);
```

Teraz jednak, dzięki sposobowi, w jaki jest zorganizowany budowniczy, możemy użyć metod rozszerzających zamiast dziedziczenia, aby dostarczyć budowniczemu dodatkowych funkcjonalności, takich jak zdolność do określenia stanowiska osoby:

```
public static class PersonBuilderExtensions
{
    public static PersonBuilder WorksAs
        (this PersonBuilder builder, string position)
        => builder.Do(p => p.Position = position);
}
```

W tym podejściu nie ma problemów z dziedziczeniem ani magii związanej z rekurencją. Za każdym razem, gdy chcemy wprowadzić dodatkowe zachowanie, po prostu dodajemy je w postaci metod rozszerzających, co pozwala utrzymać zgodność z zasadą OCP.

A oto sposób, w jaki można skorzystać z tej konfiguracji:

```
var person = new PersonBuilder()
    . Called("Dmitri")
    . WorksAs("Programista")
    . Build();
```

Ściśle mówiąc, powyższe podejście funkcyjne można przekształcić na generyczną klasę bazową wielokrotnego użytku, która mogłaby być używana do budowy różnych obiektów. Jedynym problemem byłaby konieczność propagowania typu potomnego do klasy bazowej, co ponownie wymaga zastosowania rekurencyjnych typów generycznych. Klasę bazową `FunctionalBuilder` można zdefiniować w następujący sposób:

```
public abstract class FunctionalBuilder<TSubject, TSelf>
where TSelf : FunctionalBuilder<TSubject, TSelf>
where TSubject : new()
{
    private readonly List<Func<TSubject, TSubject>> actions
    = new List<Func<TSubject, TSubject>>();
    public TSelf Do(Action<TSubject> action)
    => AddAction(action);
    private TSelf AddAction(Action<TSubject> action)
    {
        actions.Add(p => {
            action(p);
            return p;
        });
        return (TSelf)this;
    }
```


Kup ksi k

Pole ksi k

```
    public TSubject Build()
    => actions.Aggregate(new TSubject(), (p, f) => f(p));

}
```

Klasę `PersonBuilder` można teraz uprościć do następującej postaci:

```
public sealed class PersonBuilder
: FunctionalBuilder<Person, PersonBuilder>
{
    public PersonBuilder Called(string name)
    => Do(p => p.Name = name);

}
```

Z kolei klasę `PersonBuilderExtensions` można pozostawić bez zmian. Przy zastosowaniu tego podejścia można łatwo wielokrotnie wykorzystywać klasę `FunctionalBuilder` jako klasę bazową dla innych funkcyjnych budowniczych w aplikacji. Zauważ, że stosując paradygmat funkcyjny, wciąż trzymamy się koncepcji, że potomne klasy budowniczych są zapieczętowane i rozszerzane dzięki zastosowaniu metod rozszerzających.

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 HMTL, 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
<img src="pokemon.com/pikachu.png"/>
</p>
```


Kup ksi k   Pole ksi k

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[2].

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.

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.
- W ten sposób możesz całkowicie ukryć budowany obiekt.
- 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 wspomniałem 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.

---

[2] Przykład można znaleźć w artykule Tomasa Petriceka *DSL for constructing HTML* pod adresem: *http://fssnip.net/hf*.

Kup książkę Poleć książkę

Kup ksi k

Pole ksi k

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

Wzorce projektowe w .NET Core 3: tylko dla najlepszych architektów!

Rozpoznawanie wzorców i ich trafne stosowanie to umiejętności, które przydają się w różnych sytuacjach. Są bardzo cenne dla architektów: pozwalają na szybkie opracowanie złożonych zagadnień, ale też ułatwiają rozwiązywanie konkretnych problemów na wiele sposobów. Pomagają w podejmowaniu decyzji o konkretnej technologii i niezbędnych kompromisach. Każda kolejna wersja języka programowania czy platformy, na której uruchamia się kod, jest dobrą okazją do przejrzenia istniejących wzorców projektowych, ich zaktualizowania lub opracowania kolejnych.

To nowe, uzupełnione wydanie przewodnika po implementacjach klasycznych i zaawansowanych wzorców projektowych wdrażanych w językach C# i F#. Wzorce zaprezentowano wraz ze scenariuszami, do których mają zastosowanie. Omówiono też alternatywy i relacje zachodzące między wzorcami, pokazano również sposoby użycia narzędzi do refaktoryzacji (ReSharper) w celu ułatwienia implementacji. Nowe wydanie książki zostało uzupełnione o takie wzorce jak funkcyjny budowniczy, asynchroniczna metoda wytwórcza, adapter generycznych wartości i pełnomocnik kompozytu. Poszczególne zagadnienia przedstawiono w formie kompletnych, samodzielnych przykładów, z których wiele zawiera także scenariusze zaawansowane.

W książce między innymi:

- najnowsze implementacje wzorców projektowych w językach C# 8 i F# 5
- zasady tworzenia nowoczesnej architektury oprogramowania
- refaktoryzacja kodu do wzorców projektowych
- sprawdzone odmiany wzorców projektowych
- najnowsze implementacje języka C# oraz środowiska Visual Studio, Rider i ReSharper

Dmitri Nesteruk — jest analitykiem giełdowym i programistą, a także autorem książek i kursów. Sporadycznie występuje jako prelegent na konferencjach. Zawodowo interesuje się integracją rozwiązań w dziedzinie obliczeń, finansów i algorytmicznego handlu. Z upodobaniem programuje w C# i C++ i implementuje wysokowydajne przetwarzanie danych za pomocą takich technologii jak CUDA i FPGA. W latach 2009 – 2018 posiadał tytuł MVP w dziedzinie języka C#.

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-8137-7
9 788328 381377
Cena: 69,00 zł

Apress