Spis treści

Wprowadzenie

C# nieustannie ewoluuje i się zmienia. Wraz z nim zmienia się też jego społeczność. Coraz więcej programistów rozpoczyna swoją profesjonalną karierę od języka C#. Ci członkowie naszej społeczności nie mają wstępnych nawyków powszechnych wśród tych z nas, którzy zaczęli używać C# po latach doświadczeń z innym językiem z rodziny C. Nawet tym deweloperom, którzy od lat używają C#, ostatnie przyśpieszenie tempa zmian również przyniosło potrzebę dostosowania się do wielu nowych zachowań. Szczególne przyśpieszenie tempa wprowadzania innowacji w języku C# nastąpiło od chwili, gdy kompilator stał się projektem o otwartym kodzie (open source). Przegląd proponowanych funkcjonalności języka C# angażuje dziś całą społeczność, a nie tylko małą grupkę ekspertów. Społeczność ta może również uczestniczyć w projektowaniu nowych funkcji.

Zmiany w zalecanych architekturach i wdrożeniach również zmieniają typowe idiomy języka, których używamy jako deweloperzy C#. Budowanie aplikacji poprzez łączenie mikrousług, rozproszone programy czy odseparowanie danych od algorytmów – wszystko to są części nowoczesnego projektowania aplikacji. Język C# zaczął zmierzać w stronę ogarnięcia tych, jakże odmiennych idiomów.

Drugie wydanie książki *Bardziej efektywny C#* przygotowałem uwzględniając zarówno zmiany w samym języku, jak i w społeczności C#. *Bardziej efektywny C#* nie jest wycieczką historyczną poprzez zmiany w języku, ale raczej ma na celu doradzać, jak używać bieżącego języka C#. Punkty, które usunąłem z tego wydania, to te, które nie są już istotne dla dzisiejszej wersji języka C# ani dla dzisiejszych zastosowań. Nowe punkty omawiają funkcjonalności języka i środowiska oraz te praktyki, które społeczność wypracowała poprzez budowanie wielu wersji oprogramowania przy użyciu C#. Czytelnicy poprzedniego wydania zauważą, że w tym wydaniu zawarte są niektóre fragmenty poprzedniej wersji, ale znacząca liczba punktów została usunięta. Tych

50 punktów, w większości nowych, to zbiór zaleceń, które powinny pomóc w skutecznym używaniu języka C# jako profesjonalny deweloper.

W książce tej zakładam, że Czytelnik używa C# 7, ale nie jest to wyczerpujące omówienie nowych funkcji języka. Podobnie jak inne książki z serii *Effective Software Development Series*, oferuje porady praktyczne na temat używania tych funkcji do rozwiązywania problemów, z którymi możemy spotkać się każdego dnia. Funkcjonalności C# 7 są omawiane wtedy, gdy nowe funkcje języka zapewniają nowe i lepsze sposoby pisania typowych idiomów. Wyszukiwanie w Internecie może nadal pokazywać wcześniejsze rozwiązania, za którymi stoją lata historii. W książce tej szczególnie wskazuję te starsze zalecenia i wyjaśniam, dlaczego nowsze ulepszenia języka dają lepsze rozwiązania.

Wiele spośród zaleceń zawartych w tej książce można weryfikować przy użyciu bazujących na Roslyn analizatorów i poprawek kodu. Repozytorium kodu prezentowanego w książce utrzymuję pod adresem *https://github.com/BillWagner/MoreEffectiveCSharpAnalyzers*.

Kto powinien przeczytać tę książkę?

Bardziej efektywny C# został napisany z myślą o profesjonalnych deweloperach, dla których C# jest podstawowym językiem programowania. Zakładam, że Czytelnik zna składnię i funkcjonalności C# i – w ogólności – biegle posługuje się tym językiem. Książka ta nie jest podręcznikiem na temat funkcjonalności języka. Zamiast tego omawia to, jak można włączyć wszystkie funkcje bieżącej wersji języka C# do naszej codziennej pracy.

Oprócz dobrej znajomości języka C# zakładam, że Czytelnik ma przynajmniej podstawową wiedzę na temat Common Language Runtime (CLR) i kompilatora *just-in-time* (JIT).

Zawartość książki

W dzisiejszym świecie dane są wszechobecne. Zorientowane obiektowo podejście traktuje dane i kod jako część typu i jego odpowiedzialności. Podejście funkcjonalne traktuje metody jako dane. Podejścia zorientowane usługowo rozdzielają dane od kodu, który nimi manipuluje. Ewolucja C# sprawiła, że język zawiera idiomy typowe dla każdego z tych paradygmatów – co może znacznie komplikować wybory projektowe. W rozdziale 1 omawiam

te wybory i staram się przedstawić wskazówki, kiedy należy wybierać różne idiomy języka dla różnych zastosowań.

Programowanie jest zasadniczo projektowaniem API. To w ten sposób komunikujemy się z oczekiwaniami naszych użytkowników dotyczącymi używania naszego kodu. Mówi też głośno, jak rozumiemy potrzeby i oczekiwania innych deweloperów. W rozdziale 2 pokazuję najlepsze sposoby wyrażenia naszych intencji przy użyciu bogatej palety funkcji języka C#. Można w nim zobaczyć, jak wykorzystać opóźnioną ewaluację, tworzyć kompozycyjne interfejsy i unikać nieporozumień dotyczących różnych elementów języka w publicznych interfejsach.

Asynchroniczne programowanie bazujące na zadaniach udostępnia nowe techniki komponowania aplikacji z asynchronicznych bloków konstrukcyjnych. Udoskonalenie tych technik oznacza, że możemy tworzyć API dla operacji asynchronicznych, które jasno odzwierciedlają to, jak kod będzie wykonywany, a przy tym są łatwe w użyciu. W rozdziale 3 pokazuję, jak używać wsparcia języka dla zadań asynchronicznych w celu pokazania, jak nasz kod będzie wykonywany względem wielu usług i przy użyciu różnych zasobów.

Rozdział 4 skupia się na jednym szczególnym podzbiorze programowania asynchronicznego: wielowątkowym wykonywaniu równoległym. Pokazuję w nim, jak PLINQ umożliwia łatwiejszą dekompozycję złożonych algorytmów pomiędzy wiele rdzeni i wiele procesorów.

Rozdział 5 omawia użycie C# jako języka dynamicznego. C# jest silnie typowanym, statycznie typowanym językiem. Jednak w dzisiejszym czasie coraz większa liczba programów zawiera zarówno statyczne, jak i dynamiczne typowanie. C# udostępnia sposoby wykorzystania idiomów dynamicznego programowania bez utraty zalet statycznego typowania w całym programie. W tym rozdziale pokazuję, jak używać funkcji dynamicznych i uniknąć rozlewania się typów dynamicznych na cały program.

Rozdział 6 podsumowuje książkę kilkoma sugestiami dotyczącymi zaangażowania się w globalną społeczność C#. Istnieje wiele sposobów na uczestniczenie w tej społeczności i kształtowania języka, którego używamy na co dzień.

Konwencje dotyczące kodu

Prezentowanie kodu w książce zawsze wymaga poczynienia pewnych kompromisów na rzecz oszczędności miejsca i przejrzyjstości. Starałem się wydobyć takie próbki, które ilustrują określone zagadnienie. Często oznacza to pominięcie znacznych fragmentów klasy czy metody. Niekiedy oznacza to pomijanie dla oszczędności miejsca konstrukcji przywracania po błędach. Publiczne metody powinny weryfikować swoje parametry i inne dane wejściowe, ale ten kod również jest zazwyczaj pominięty w książce ze względu na ograniczenia miejsca. Analogiczne uwarunkowania skłoniły mnie do usunięcia weryfikowania wywołań metod i klauzul `try/finally`, które powinny często być włączone do złożonych algorytmów.

Zakładam też, że większość Czytelników potrafi znaleźć właściwą przestrzeń nazw, gdy przykład używa jednej z typowych. Można bezpiecznie założyć, że każdy przykład niejawnie zawiera następujące instrukcje `using`:

```
using System;
using static System.Console;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

Informacje zwrotne

Niezależnie od moich najlepszych starań i wysiłków ludzi, którzy przeglądali tekst, do treści lub przykładów mogły wkraść się jakieś błędy. Jeśli ktoś jest przekonany, że znalazł błąd, proszę o kontakt pod adresem bill@thebillwagner.com lub na Twitterze pod @billwagner. Errata zostanie opublikowana pod adresem *http://thebillwagner.com/Resources/MoreEffectiveCS*. Wiele punktów w tej książce zostało zainspirowanych wiadomościami i konwersacjami na Twitterze z innymi programistami C#. Jeśli ktoś ma pytania lub komentarze dotyczące zaleceń, proszę o kontakt. Ogólne omówienia będę zamieszczał na moim blogu pod adresem *http://thebillwagner.com/blog*.

Podziękowania

Istnieje wiele osób, którym winien jestem podziękowania za ich wkład w tę książkę. Mam zaszczyt być częścią wspaniałej społeczności C# od wielu lat. Niemal każdy z listy mailingowej C# *Insiders* (zarówno z wnętrza firmy Microsoft, jak i spoza niej) wniósł jakieś idee i uwagi, które sprawiły, że ta książka jest lepsza.

Oddzielnie muszę wymienić kilku członków społeczności C#, którzy bezpośrednio pomogli mi w formułowaniu koncepcji i przekształcaniu ich na solidne zalecenia. Rozmowy z Jonem Skeetem, Dustinem Campbellem, Kevinem Pilchem, Jaredem Parsonsem, Scottem Allenem oraz, przede wszystkim, Madsem Torgersenem stały się podstawą wielu nowych pomysłów zawartych w tym wydaniu.

Przy tym wydaniu miałem fantastyczny zespół recenzentów technicznych. Jason Bock, Mark Michaelis i Eric Lippert ślęczeli nad tekstem i przykładami i znacząco poprawili jakość tej książki. Ich recenzje były szczegółowe i kompletne, co jest czymś najlepszym, na co można liczyć. Poza tym dodali sugestie, które pomogły mi lepiej wyjaśnić wiele zagadnień.

To zaszczyt po raz kolejny być autorem kolejnej pozycji w serii redagowanej przez Scotta Meyersa. Scott jest niebywale dokładny, a jego doświadczenie w programowaniu, choć niekoniecznie w C#, oznacza, że potrafił dostrzec wiele miejsc, w których nie wyjaśniłem jakiegoś punktu dostatecznie jasno lub nie dość uzasadniłem zalecenia. Jego uwagi, jak zawsze, były nieocenione podczas przygotowywania tego wydania.

Jak zawsze, moja rodzina podarowała mi czas, dzięki czemu mogłem ukończyć tę książkę. Moja żona, Marlene, cierpliwie czekała niezliczone godziny, gdy wychodziłem, aby pisać lub tworzyć próbki. Bez jej wsparcia nigdy nie ukończyłbym tej ani żadnej innej książki, ani też ukończenie ich nie byłoby tak satysfakcjonujące.

O autorze

Bill Wagner jest jednym z czołowych projektantów C#, członkiem ECMA C# Standards Committee i autorem trzech wydań *Effective C#*. Jest też prezesem Humanitarian Toolbox, został wyróżniony tytułem Microsoft Regional Director oraz .NET MVP od 11 lat, zaś ostatnio dołączył do .NET Foundation Advisory Council. Wagner wielokrotnie pomagał rozmaitym firmom, od start-upów po międzynarodowe korporacje w usprawnianiu procesów projektowych i rozwijaniu zespołów programistycznych. Obecnie jako członek zespołu .NET Core tworzy materiały szkoleniowe dla programistów z dziedziny języka C# i .NET Core. Uzyskał tytuł B.S. z informatyki na Uniwersytecie stanu Illinois w Champaign-Urbana.

1 Praca z typami danych

Język C# oryginalnie został zaprojektowany pod kątem wsparcia dla zorientowanych obiektowo technik programowania, łączących razem obsługę danych i funkcjonalność. W miarę rozwoju języka dodawano w nim nowe idiomy mające na celu wsparcie dla praktyk programistycznych, które w międzyczasie stały się bardziej powszechne. Jednym z tych trendów jest odseparowanie zagadnień przechowywania danych od metod, które manipulują tymi danymi. Wynika to z przechodzenia w stronę systemów rozproszonych, w których aplikacja jest rozdzielana na wiele mniejszych usług, z których każda implementuje albo pojedynczą funkcjonalność, albo (co najwyżej) niewielki zbiór powiązanych funkcjonalności. Przyjęcie nowej strategii rozdzielania zagadnień w naturalny sposób spowodowało rozwinięcie się nowych technik programistycznych. I analogicznie, korzystanie z nowych technik programowania powoduje pojawienie się nowych funkcji języka.

W tym rozdziale przedstawię techniki oddzielania danych od metod służących do manipulowania lub przetwarzania tych danych. Dane te nie zawsze muszą być obiektami – niekiedy będą to funkcje i pasywne kontenery danych.

Punkt 1: Korzystanie z właściwości zamiast dostępnych pól danych

Właściwości zawsze były funkcjonalnością języka C#, ale szereg usprawnień wprowadzonych od chwili początkowego wydania języka sprawiają, że właściwości stały się jeszcze bardziej przekonujące. Dla przykładu, można wyspecyfikować różne ograniczenia dostępu do modułów odczytu (akcesor `get`, potocznie nazywany *getter*) i modyfikowania (akcesor `set`, potocznie *setter*). Automatyczne właściwości minimalizują ręczne pisanie kodu (w odróżnieniu od pól członkowskich), włącznie z właściwościami tylko do odczytu. Elementy z wcielonymi wyrażeniami zapewniają jeszcze bardziej zwartą składnię. Jeśli ktoś nadal tworzy publiczne pola danych w swoich typach, powinien przestać. Jeśli nadal

ręcznie pisze metody `get` i `set`, również trzeba z tym skończyć. Właściwości pozwalają eksponować pola członkowskie jako część publicznego interfejsu, ale nadal zapewniają hermetyzację, którą chcielibyśmy mieć w środowisku zorientowanym obiektowo. Właściwości są elementami języka, do których dostęp uzyskujemy tak samo, jakby były polami danych, ale implementowanymi jako metody.

Niektóre pola członkowskie typu naprawdę najlepiej reprezentować jako dane: nazwa klienta, lokalizacja (x, y) punktu czy zeszłoroczne dochody. Właściwości pozwalają utworzyć interfejs, który działa tak, jakbyśmy uzyskiwali bezpośredni dostęp do pól danych, ale nadal zapewniają wszystkie zalety metody. Kod kliencki uzyskuje dostęp do właściwości tak samo, jak do pól publicznych. Jednak rzeczywista implementacja używa metod, w których możemy definiować zachowanie akcesorów właściwości.

Środowisko .NET Framework zakłada, że będziemy używać właściwości dla swoich publicznych członkowskich pól danych. W istocie klasy wiązania danych zawarte w .NET Framework wspierają właściwości, a nie publiczne pola danych. Stwierdzenie to jest prawdą dla wszystkich bibliotek wiązania danych: WPF, Windows Forms oraz Web Forms. Powiązanie danych (zgodnie z nazwą) wiąże właściwość obiektu z kontrolką interfejsu użytkownika. Mechanizm wiązania danych używa refleksji, aby odnaleźć nazwaną właściwość w danym typie:

```
textBoxCity.DataBindings.Add("Text",
    address, nameof(City));
```

Kod ten wiąże właściwość `Text` kontrolki `textBoxCity` z właściwością `City` obiektu `address`. Nie będzie działać z publicznym polem danych o nazwie `City`; projektanci Framework Class Library nie zdecydowali się na wspieranie takiej praktyki programowania. Używanie publicznych pól członkowskich danych jest postrzegane jak zła praktyka, zatem nie dodano wsparcia dla nich do Framework Class Library. To pominięcie jest jeszcze jednym powodem, aby stosować się do właściwych technik zorientowanych obiektowo.

Tak, wiązanie danych ma zastosowanie tylko do tych klas, które zawierają elementy wyświetlane przez naszą logikę interfejsu użytkownika (UI). Nie oznacza to jednak, że właściwości należy używać wyłącznie w kontekście logiki UI: należy również używać właściwości w innych klasach i strukturach. Właściwości można znacznie łatwiej zmieniać, gdy po pewnym czasie odkryjemy nowe wymagania lub potrzebę innych zachowań. Na przykład możemy zdecydować, że nasz typ opisujący klienta (nazwijmy go `Customer`) nigdy nie

powinien zawierać pustej nazwy. Jeśli używamy publicznej właściwości dla tej nazwy, wymaganie to można dodać, gdyż zmiana musi być wykonana tylko w jednym miejscu:

```
public class Customer
{
    private string name;
    public string Name
    {
        get => name;
        set
        {
            if (string.IsNullOrWhitespace(value))
                throw new ArgumentException(
                    "Name cannot be blank",
                    nameof(Name));
            name = value;
        }

        // Dalszy kod pominięty
    }
}
```

Gdybyśmy użyli publicznych członkowskich pól danych, konieczne byłoby szczegółowe przejrzenie każdego fragmentu kodu ustawiającego nazwę klienta i wprowadzić tam odpowiednią poprawkę. To zajmuje czas – znacznie więcej czasu.

Ponieważ właściwości są implementowane poprzez metody, dodanie wsparcia dla wielowątkowości również jest łatwiejsze. Można ulepszyć implementację akcesorów `get` i `set`, aby zapewnić synchronizowany dostęp do danych (więcej szczegółów zawiera punkt 39):

```
public class Customer
{
    private object syncHandle = new object();

    private string name;
    public string Name
    {
        get
```

```
        {
            lock (syncHandle)
                return name;
        }

        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "Name cannot be blank",
                    nameof(Name));
            lock (syncHandle)
                name = value;
        }
    }

    // Dalszy kod pominięty
}
```

Właściwości udostępniają te same funkcjonalności języka, co metody. W szczególności właściwości mogą być wirtualne:

```
public class Customer
{
    public virtual string Name
    {
        get;
        set;
    }
}
```

Zauważmy, że kilka ostatnich przykładów używa skróconej (niejawnej) składni właściwości. Tworzenie właściwości, aby opakować leżący w tle magazyn, jest powszechnie używanym wzorcem. Często nie potrzebujemy logiki weryfikacyjnej w getterach i setterach właściwości. Język C# obsługuje uproszczoną, niejawną składnię, co pozwala zmniejszyć ilość standardowego „etykietowego" kodu koniecznego do eksponowania pojedynczego kodu jako właściwości. Kompilator tworzy za nas pole prywatne (nazywane zwykle magazynem w tle – *backing store*) i implementuje oczywistą logikę obydwu akcesorów `get` i `set`.

Możemy rozszerzyć właściwości, aby były abstrakcyjne, a także definiować je jako część definicji interfejsu, używając składni analogicznej do używanej dla właściwości niejawnych. Kolejny przykład pokazuje definicję właściwości w interfejsie ogólnym. Choć składnia jest zgodna z używaną dla właściwości niejawnych, ta definicja interfejsu nie zawiera żadnej implementacji. Definiuje ona kontrakt, który musi zostać spełniony przez każdy typ implementujący ten interfejs.

```
public interface INameValuePair<T>
{
    string Name { get; }

    T Value { get; set; }
}
```

Właściwości są w pełni wyposażonymi, czołowymi elementami języka, stanowiącymi rozszerzenie metod uzyskujących dostęp lub modyfikujących dane wewnętrzne. Cokolwiek można zrobić poprzez funkcje pól członkowskich, możemy zrealizować przy użyciu właściwości. Dodatkowo właściwości unikają jeszcze jednego ważnego źródła problemów, które są możliwe przy korzystaniu z pól: nie można przekazać właściwości do metody, używając słów kluczowych `ref` lub `out`.

Akcesory właściwości są dwiema oddzielnymi metodami, które są włączone do naszego typu. Możemy wyspecyfikować różne modyfikatory dostępności dla akcesorów `get` i `set` tej samej właściwości. Ta elastyczność zapewnia jeszcze większą kontrolę nad widocznością tych elementów danych, które eksponujemy jako właściwości:

```
public class Customer
{
    public virtual string Name
    {
        get;
        protected set;
    }
    // Reszta implementacji pominięta
}
```

Składnia właściwości wykracza poza proste pola danych. Jeśli nasz typ zawiera indeksowane elementy jako część swojego interfejsu, możemy użyć

indeksatorów (które są parametryzowanymi właściwościami). Takie podejście jest wygodnym sposobem tworzenia właściwości, które zwracają elementy w określonej sekwencji:

```
public int this[int index]
{
    get => theValues[index];
    set => theValues[index] = value;
}
     private int[] theValues = new int[100];

// dostęp do indeksatora:
int val = someObject[i];
```

Indeksatory zapewniają takie same wsparcie języka, jak jednoelementowe właściwości: są implementowane jako napisane przez nas metody, zatem możemy zastosować wewnątrz indeksatora dowolne weryfikacje lub obliczenia. Indeksatory mogą być wirtualne lub abstrakcyjne, można je deklarować w interfejsach i mogą być dostępne tylko do odczytu albo do odczytu i zapisu. Jednowymiarowe indeksatory z numerycznymi parametrami mogą uczestniczyć w wiązaniu danych. Inne indeksatory mogą używać parametrów innych niż liczby całkowite, aby definiować mapowania:

```
public Address this[string name]
{
    get => addressValues[name];
    set => addressValues[name] = value;
}
private Dictionary<string, Address> addressValues;
```

Pozostając na chwilę przy wielowymiarowych tablicach w języku C#, możemy tworzyć wielowymiarowe indeksatory, przy czym dla każdej osi mogą być używane takie same, ale również różne typy:

```
public int this[int x, int y]
    => ComputeValue(x, y);

public int this[int x, string name]
    => ComputeValue(x, name);
```

Zauważmy, że wszystkie indeksatory są deklarowane przy użyciu słowa kluczowego `this`. W języku C# nie można nazwać indeksatora, zatem przy tworzeniu wielu różnych indeksatorów w naszym typie każdy z nich musi mieć unikatową listę parametrów, aby uniknąć niejednoznaczności. Niemal wszystkie funkcjonalności właściwości są również dostępne w przypadku indeksatorów: mogą być one wirtualne lub abstrakcyjne i mogą mieć oddzielne ograniczenia dostępu dla akcesorów. Jest jednak jedna różnica – nie możemy utworzyć indeksatorów niejawnych, tak jak niejawnych właściwości.

Funkcjonalność właściwości jest samym dobrem i jest atrakcyjnym usprawnieniem w porównaniu do wcześniejszych wersji C#. Mimo tego nadal możemy odczuwać pokusę, aby utworzyć wstępną implementację przy użyciu członkowskich pól danych, po czym zastąpić te pola właściwościami, gdy pojawi się potrzeba wykorzystania którejś z tych zalet. Może to się wydawać uzasadnioną i sensowną strategią, ale jest to błąd. Rozważmy poniższy fragment definicji klasy:

```
// Używanie publicznych pól członkowskich - zła praktyka:
public class Customer
{
    public string Name;

    // Pozostała część implementacji pominięta
}
```

Ta definicja klasy opisuje obiekt klienta z nazwą. Możemy odczytać lub ustawić nazwę, używając dobrze znanej notacji członkowskiej:

```
string name = customerOne.Name;
customerOne.Name = "This Company, Inc.";
```

Jest to proste i oczywiste. Moglibyśmy sobie wyobrażać, że będziemy mogli później zastąpić pole członkowskie `Name` odpowiednią właściwością i nasz kod powinien nadal działać bez żadnych zmian. No cóż, tylko częściowo jest to prawdą. Właściwości zostały pomyślane tak, aby *wyglądały jak* pola członkowskie przy uzyskiwaniu dostępu do nich – taki jest cel wybranej składni. Jednak właściwości *nie są* danymi; dostęp do właściwości generuje inne instrukcje Microsoft Intermediate Language (MSIL), niż dostęp do danych.

Choć właściwości i pola członkowskie są kompatybilne na poziomie kodu źródłowego, nie są kompatybilne binarnie. W oczywistym przypadku ograniczenie to oznacza, że jeśli dokonamy zamiany publicznego pola

członkowskiego na równoważną właściwość publiczną, musimy ponownie skompilować cały kod, który używał publicznego pola członkowskiego. C# traktuje binarne asemblacje jako obywateli pierwszej klasy. Jednym z celów stawianych przed projektantami języka było umożliwienie aktualizowania tylko pojedynczej, poprawionej asemblacji bez konieczności aktualizacji całej aplikacji. Jednak prosty akt zamiany pola członkowskiego na właściwość zrywa kompatybilność binarną, zatem powoduje, że aktualizowanie już wdrożonych pojedynczych asemblacji staje się znacznie trudniejsze.

Skoro już przyglądamy się instrukcjom MSIL dla właściwości, może pojawić się pytanie o względną wydajność właściwości i pól członkowskich. Wydajność przy korzystaniu z właściwości nie będzie większa, niż w przypadku dostępu do pól członkowskich, ale też nie powinna być gorsza. Kompilator just-in-time (JIT) wykonuje włamywanie (*inlining*) kodu wywołań (niektórych) metod, w tym metod akcesorów. Gdy kompilator JIT włamie akcesory właściwości, wydajność pól członkowskich i właściwości jest taka sama. Nawet jeśli akcesor właściwości nie zostanie włamany, rzeczywistą różnicę wydajności stanowi pomijalny koszt jednego wywołania funkcji. Taką różnicę da się zmierzyć jedynie w bardzo nielicznych, ekstremalnych przypadkach.

Właściwości są metodami, które z punktu widzenia wywołującego kodu wyglądają jak dane, co powoduje, że użytkownicy mogą mieć pewne oczekiwania. Będą postrzegać dostęp do właściwości, jakby to był dostęp do danych. Ostatecznie tak właśnie to wygląda. Nasze akcesory muszą zatem spełnić te oczekiwania. Akcesory `get` nie mogą dawać dających się zaobserwować efektów ubocznych. Dla kontrastu, akcesory `set` rzeczywiście modyfikują stan i użytkownicy powinni być w stanie widzieć te zmiany.

Akcesory właściwości określają również oczekiwania wydajności użytkowników. Dostęp do właściwości wygląda jak dostęp do pola danych. Nie powinien mieć charakterystyki wydajnościowej, która byłaby znacząco różna od typowej dla prostego dostępu do danych. Akcesory nie powinny zatem wykonywać długotrwałych obliczeń ani wykonywać wywołań międzyaplikacyjnych (takich jak zapytania do bazy danych) ani realizować żadnych innych długich operacji, które byłyby niespójne z oczekiwaniami użytkowników.

Ilekroć potrzebujemy eksponować dane w publicznych lub chronionych interfejsach naszych typów, należy używać właściwości. Dla sekwencji lub słowników używajmy indeksatorów. Wszystkie pola członkowskie danych powinny być prywatne, bez żadnych wyjątków. Dzięki takim wyborom natychmiast uzyskamy wsparcie dla wiązania danych i znacznie łatwiejsza będzie

modyfikacja implementacji metod w przyszłości. Wpisanie nadmiarowego kodu potrzebnego do opakowania dowolnej zmiennej we właściwość zajmie jedną lub dwie minuty dziennie. Dla porównania, późniejsze odkrycie, że należało użyć właściwości, aby odpowiednio wyrazić naszą koncepcję projektową, będzie oznaczało wiele godzin pracy. Lepiej poświęcić nieco czasu dziś, aby oszczędzić bardzo wiele w przyszłości.

Punkt 2: Preferowanie niejawnych właściwości dla zmiennych danych

Rozszerzenia składni właściwości w języku C# oznaczają, że można jasno wyrazić nasze zamierzenia projektowe przy użyciu właściwości. Nowoczesna składnia C# dodatkowo wspiera późniejsze zmiany naszego projektu. Zaczynajmy od właściwości, a udostępnimy sobie wiele przyszłych scenariuszy.

Gdy dodajemy do klasy dane, które mają być dostępne, akcesory właściwości często są prostymi opakowaniami dla naszych pól danych. Gdy mamy do czynienia z takim przypadkiem, można poprawić czytelność kodu, używając właściwości niejawnych:

```
public string Name { get; set; }
```

Kompilator utworzy leżące w tle pole prywatne, używając wygenerowanej przez siebie nazwy. Możemy nawet użyć akcesora `set` do modyfikowania wartości tego pola. Ponieważ nazwa pola w tle jest generowana przez kompilator, nawet wewnątrz naszej własnej klasy musimy wywołać akcesor właściwości i nie możemy bezpośrednio modyfikować tego pola. Nie stanowi to problemu: wywołanie akcesora właściwości wykonuje tę samą pracę, a ponieważ generowany akcesor jest prostą instrukcją przypisania, jego kod najprawdopodobniej zostanie włamany. Zachowanie wykonawcze niejawnej właściwości jest takie samo, jak przy uzyskiwaniu dostępu wprost do pola w tle, również pod względem wydajności.

Niejawnie definiowane właściwości wspierają takie same modyfikatory dostępu, jak ich jawne odpowiedniki. Możemy zdefiniować tak restrykcyjny setter, jak tego potrzebujemy:

```
public string Name
{
    get;
    protected set;
}
```

```
// albo
public string Name
{
    get;
    internal set;
}
// albo
public string Name
{
    get;
    protected internal set;
}
// albo
public string Name
{
    get;
    private set;
}
// albo może zostać ustawiona tylko w konstruktorze:
public string Name { get; }
```

Niejawne właściwości tworzą taki sam wzorzec z leżącym w tle polem, który trzeba było ręcznie wpisać we wcześniejszych wersjach C#. Zalety korzystania z niejawnych właściwości to zwiększona produktywność programisty, a jednocześnie tworzone klasy są bardziej czytelne. Niejawna deklaracja właściwości pokazuje każdemu, kto będzie czytał kod, dokładnie to, co zamierzaliśmy uzyskać, nie zaśmiecając pliku dodatkowymi informacjami, które zaciemniałyby prawdziwy sens.

Oczywiście, skoro niejawne właściwości generują taki sam kod, jak jawne, możemy użyć ich do definiowania właściwości wirtualnych, nadpisywania właściwości wirtualnych lub implementowania właściwości zdefiniowanej w interfejsie.

Po utworzeniu wirtualnej właściwości niejawnej potomne klasy nie będą miały dostępu do wygenerowanego przez kompilator magazynu w tle. Tym niemniej przesłaniające klasy mogą uzyskiwać dostęp do bazowych metod `get` i `set` właściwości, tak samo jak w przypadku dowolnej innej metody wirtualnej:

```
public class BaseType
{
    public virtual string Name
    {
        get;
        protected set;
    }
}

public class DerivedType : BaseType
{
    public override string Name
    {
        get => base.Name;
        protected set
        {
            if (!string.IsNullOrEmpty(value))
                base.Name = value;
        }
    }
}
```

Ponadto korzystanie z niejawnych właściwości daje dwie dodatkowe korzyści. Po pierwszej, gdy zajdzie potrzeba zastąpienia właściwości niejawnej solidną implementacją (na przykład ze względu na konieczność dodania weryfikacji danych lub innych działań), dokonywane zmiany w klasie będą kompatybilne binarnie. Po drugie, nasza weryfikacja pojawi się tylko w jednym miejscu.

We wcześniejszych wersjach języka C# większość programistów wykorzystywało bezpośredni dostęp do pola danych w tle, jeśli potrzebowali zmodyfikować je w ramach swojej własnej klasy. Taka praktyka tworzy kod, który rozprasza mechanizmy weryfikacji i sprawdzania błędów po całym pliku. Każda próba zmiany pola w tle niejawnej właściwości wywołuje odpowiedni akcesor (być może prywatny). Możemy zatem przekształcić akcesor właściwości niejawnej w akcesor właściwości jawnej, po czym wpisać logikę weryfikacji w tym nowym akcesorze:

```
// Werja oryginalna
public class Person
{
```

```
    public string FirstName { get; set;}
    public string LastName { get; set; }
    public override string ToString() =>
        $"{FirstName} {LastName}";
}

// Późniejsza aktualizacja w celu weryfikacji
public class Person
{
    public Person(string firstName, string lastName)
    {
        // Wykorzystanie weryfikacji w akcesorze:
        this.FirstName = firstName;
        this.LastName = lastName;
    }
    private string firstName;
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "First name cannot be null or empty");
            firstName = value;
        }
    }

    private string lastName;
    public string LastName
    {
        get => lastName;
        private set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "Last name cannot be null or empty");
            lastName = value;
        }
```

```
    }
    public override string ToString() =>
        $"{FirstName} {LastName}";
}
```

Przy korzystaniu z właściwości całą weryfikację musimy utworzyć w tylko jednym miejscu. Jeśli nadal będziemy używać swojego akcesora, zamiast wprost odwoływać się do pola w tle, cała praca weryfikacji zawartości może pozostać w jednym miejscu.

Niejawne właściwości mają jeszcze jedno, choć ważne ograniczenie: nie można użyć ich w typach oznaczonych atrybutem `Serializable`. Format trwałego przechowywania pliku zależy od nazwy generowanego przez kompilator pola używanego jako magazyn w tle. Nie ma jednak gwarancji, że nazwa ta pozostanie niezmieniona – inaczej mówiąc, może ulec zmianie po każdej modyfikacji klasy i jej ponownym skompilowaniu.

Niezależnie od swoich ograniczeń, niejawne właściwości oszczędzają czas programisty, pozwalają tworzyć czytelny kod i promują styl programowania, w którym wszystkie weryfikacje zmian pól odbywają się w jednym miejscu. Kiedy tworzymy bardziej przejrzysty kod, łatwiejsze jest jego późniejsze utrzymywanie.

Punkt 3: Preferowanie niezmienności typów wartościowych

Typy niezmienne łatwo można zrozumieć: gdy już zostaną utworzone, są stałe. Jeśli zweryfikujemy parametry użyte do konstruowania obiektu, możemy mieć pewność, że obiekt ten jest w poprawnym stanie cały czas od tego punktu; nie możemy zmienić wewnętrznego stanu obiektu, co spowodowałoby, że stałby się niepoprawny. Możemy oszczędzić sobie wiele (w innym przypadku niezbędnego) sprawdzania błędów, nie pozwalając na dowolne zmiany stanu po skonstruowaniu obiektu. Niezmienne typy są w naturalny sposób bezpieczne wątkowo: wiele wątków odczytujących może uzyskiwać dostęp do tej samej zawartości. Jeśli wewnętrzny stan nie może się zmienić, różne wątki nie mogą w żaden sposób zobaczyć różniącego się obrazu danych. Niezmienne typy można bezpiecznie wyeksportować z naszych obiektów, gdyż wywołujący kod nie może zmodyfikować wewnętrznego stanu obiektu.

Typy niezmienne działają lepiej w kolekcjach bazujących na haszowaniu. Wartość zwracana przez `Object.GetHashCode()` musi być niezmienna na

poziomie instancji (patrz punkt 10); warunek ten jest zawsze spełniony w niezmiennych typach.

W praktyce bardzo trudne byłoby uczynienie każdego typu niezmiennym. To dlatego zalecenie to ma zastosowanie do typów atomowych i o niezmiennych wartościach. Możemy wykonać rozkład naszych typów na struktury, które w naturalny sposób tworzą pojedynczy byt. Przykładem takiego bytu może być typ przeznaczony do przechowywania adresów. Adres jest pojedynczą rzeczą, złożoną z wielu powiązanych pól, przy czym zmiana zawartości jednego pola najprawdopodobniej oznacza zmiany również w pozostałych polach. Przeciwieństwem tego przykładu jest typ przeznaczony do przechowywania klientów, który nie jest typem atomowym. Taki typ będzie zapewne zawierał wiele porcji informacji – adres, nazwę, jeden lub więcej numerów telefonicznych – i każdy z tych niezależnych elementów informacji może ulec zmianie. Klient może zmienić numery telefonów, nie przeprowadzając się. Może zmienić siedzibę, zachowując ten sam numer telefonu. Klient może też zmienić swoją nazwę (nazwisko), bez przeprowadzania się ani zmiany numeru telefonu. *Obiekt* klienta nie jest atomowy; jest zbudowany z wielu różnych niezmiennych typów przy użyciu złożenia – adresu, nazwy lub kolekcji par numer telefonu/jego rodzaj. Typy atomowe są pojedynczymi jednostkami: w naturalnym działaniu będziemy wymieniać całą zawartość typu atomowego. Zmiana tylko jednego spośród składowych pól jest zdarzeniem wyjątkowym.

Oto typowa implementacja adresu jako typu zmiennego:

```
// Zmienna struktura Address
public struct Address
{
    private string state;
    private int zipCode;

    // Polegamy na domyślnym konstruktorze generowanym
    // przez system
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City { get; set; }
    public string State
    {
        get => state;
        set
        {
```

```
            ValidateState(value);
            state = value;
        }
    }

    public int ZipCode
    {
        get => zipCode;
        set
        {
            ValidateZip(value);
            zipCode = value;
        }
    }

    // Inne szczegóły pominięte
}

// Przykład użycia:
Address a1 = new Address();
a1.Line1 = "111 S. Main";
a1.City = "Anytown";
a1.State = "IL";
a1.ZipCode = 61111;
// Modyfikacja:
a1.City = "Ann Arbor"; // Kod pocztowy (ZipCode) oraz stan (State)
                       // są teraz nieprawidłowe
a1.ZipCode = 48103;    // Stan nadal jest nieprawidłowy
a1.State = "MI";       // Teraz jest dobrze
```

Zmiany wewnętrznego stanu oznaczają, że możliwe jest naruszenie niezmienników obiektu, przynajmniej tymczasowo. Po zastąpieniu pola `city` obiekt znalazł się w niewłaściwym stanie. Miasto się zmieniło i nie pasuje już ani do pola zawierającego stan, ani kod pocztowy. Na razie wydaje się to nieszkodliwe, ale załóżmy, że ten fragment jest częścią programu wielowątkowego: dowolna zmiana kontekstu, która nastąpi po zmianie miasta, ale przed zmianą stanu i kodu pocztowego może sprawić, że inny wątek zobaczy niespójny widok danych.

Nawet jeśli nie tworzymy wielowątkowego programu, nadal możemy popaść w kłopoty z powodu zmian wewnętrznego stanu. Załóżmy, że kod

pocztowy był nieprawidłowy i metoda set zgłosiła wyjątek. Jednak zdążyliśmy wykonać tylko część zamierzonych zmian i w efekcie pozostawiliśmy system w stanie nieprawidłowym. Aby to naprawić, będzie trzeba dodać znaczącą ilość kodu wykonującego wewnętrzną weryfikację do struktury Addres. Taki weryfikujący kod spowoduje z kolei powiększenie rozmiaru i złożoności całego kodu. Aby w pełni zaimplementować bezpieczeństwo wyjątków, konieczne będzie tworzenie ochronnych kopii wokół każdego bloku kodu, w którym będziemy zmieniać więcej niż jedno pole. Bezpieczeństwo wątkowe będzie wymagało dodania znaczącej liczby testów synchronizujących wątki dla każdego akcesora właściwości, zarówno odczytujących, jak i zapisujących. Przede wszystkim będzie to sporym przedsięwzięciem – i to takim, które najprawdopodobniej będzie się rozrastać z upływem czasu, w miarę dodawania nowych funkcjonalności.

Jeśli potrzebujemy, aby obiekt Address był strukturą (struct), lepszym podejściem będzie uczynienie go niezmiennym. Zacznijmy od zmiany wszystkich pól instancji na tylko do odczytu dla użycia zewnętrznego.

```
public struct Address
{
    // Pozostałe szczegóły pominięte
    public string Line1 { get; }
    public string Line2 { get; }
    public string City { get; }
    public string State { get; }
    public int ZipCode { get; }

    public Address(string line1,
        string line2,
        string city,
        string state,
        int zipCode) :
        this()
    {
        Line1 = line1;
        Line2 = line2;
        City = city;
        ValidateState(state);
        State = state;
        ValidateZip(zipCode);
```

```
            ZipCode = zipCode;
        }
    }
```

Teraz mamy typ niezmienny bazujący na publicznym interfejsie. Aby typ ten był użyteczny, musimy dodać wszystkie konstruktory potrzebne do pełnego zainicjowania struktury `Address`. Struktura ta potrzebuje tylko jednego dodatkowego konstruktora, specyfikującego każde pole. Konstruktor kopiujący nie jest potrzebny, gdyż operator przypisania jest równie wydajny. Przypomnijmy, że konstruktor domyślny nadal jest dostępny. Jest to adres domyślny, w którym wszystkie łańcuchy są puste, a kod pocztowy jest równy 0.

```
public Address(string line1,
    string line2,
    string city,
    string state,
    int zipCode) :
    this()
{
    Line1 = line1;
    Line2 = line2;
    City = city;
    ValidateState(state);
    State = state;
    ValidateZip(zipCode);
    ZipCode = zipCode;
}
```

Używanie typu niezmiennego wymaga nieco innej sekwencji wywoływania, aby zmodyfikować jego stan. Mówiąc ściślej, będziemy tworzyć nowy obiekt, zamiast modyfikować istniejącą instancję:

```
// Tworzenie nowego adresu:
Address a1.Line2 = new Address("111 S. Main",
    "", "Anytown", "IL", 61111);

// Aby go zmienić, musimy go ponownie zainicjować:
a1.Line2 = new Address(a1.Line1,
    a1.Line, "Ann Arbor", "MI", 48103);
```

Wartość zmiennej `a1` jest w jednym z dwóch stanów: albo zawiera oryginalną lokalizację w Anytown, albo uaktualnioną w Ann Arbor. Nie modyfikujemy istniejącego adresu, tworząc dowolny z nieprawidłowych tymczasowych stanów, pokazanych w poprzednim przykładzie. Te pośrednie stany istnieją tylko w czasie wykonywania konstruktora `Address` i nie są widoczne spoza tego konstruktora. Gdy tylko nowy obiekt `Address` zostanie skonstruowany, jego wartość jest ustalona na cały czas. Kod ten jest również bezpieczny ze względu na wyjątki: `a1` jest albo w swoim oryginalnym stanie, albo ma nową wartość. Jeśli wyjątek zostanie zgłoszony podczas konstruowania nowego obiektu `Address`, oryginalna wartość `a1` jest niezmieniona.

Aby utworzyć typ niezmienny, musimy zapewnić, że nasz kod nie zawiera żadnych luk, które pozwoliłyby kodowi klienckiemu zmieniać stan wewnętrzny. Typy wartościowe nie wspierają typów pochodnych, zatem nie musimy zabezpieczać się przed możliwością modyfikowania pól przez typy pochodne. Zamiast tego musimy zwrócić uwagę na dowolne pola w typie niezmiennym, które są zmiennymi typami referencyjnymi. Przy implementowaniu konstruktorów dla tych typów konieczne jest wykonanie ochronnej kopii tego zmiennego typu. Wszystkie kolejne przykłady zakładają, że `Phone` jest niezmiennym typem wartościowym, gdyż zajmujemy się tylko niezmiennością typów wartościowych:

```
// Prawie niezmienny: są tu luki, które pozwalają na zmiany stanu.
public struct PhoneList
{
    private readonly Phone[] phones;

    public PhoneList(Phone[] ph)
    {
        phones = ph;
    }

    public IEnumerable<Phone> Phones
    {
        get { return phones; }
    }
}

Phone[] phones = new Phone[10];
// Inicjowanie zmiennej phones
PhoneList pl = new PhoneList(phones);
```

```
// Modyfikowanie listy phone:
// powoduje również zmianę zawartości (podobno)
// niezmiennego obiektu.
phones[5] = Phone.GeneratePhoneNumber();
```

Tablice są typem referencyjnym. W tym przykładzie tablica wskazywana wewnątrz struktury `PhoneList` odnosi się do pewnej tablicy (numerów telefonów), której pamięć alokowana jest na zewnątrz naszego obiektu. Tak jak jest to teraz, programista mógłby zmodyfikować naszą niezmienną strukturę poprzez inną zmienną odwołującą się do tego samego obszaru pamięci. Aby wyeliminować tę możliwość, musimy wykonać ochronną kopię tablicy. `Array` jest typem zmiennym, zatem jedną z alternatyw będzie użycie klasy `ImmutableArray`, należącej do przestrzeni nazw `System.Collections.Immutable`. Poprzedni przykład pokazuje zagrożenia stwarzane przez zmienną kolekcję. Jeszcze więcej możliwości szkód może powstać, jeśli typ `Phone` jest zmiennym typem referencyjnym. Kod kliencki mógłby zmodyfikować wartości zawarte w kolekcji, nawet jeśli kolekcja jest zabezpieczona przed dowolnymi modyfikacjami. Remedium dla tego problemu może być użycie typu kolekcji `ImmutableList`:

```
public struct PhoneList
{
    private readonly ImmutableList<Phone> phones;

    public PhoneList(Phone[] ph)
    {
        phones = ph.ToImmutableList();
    }

    public IEnumerable<Phone> Phones => phones;
}
```

To, której z trzech strategii należy użyć do zainicjowania typu niezmiennego, zależy od złożoności tego typu. Po pierwsze, możemy zdefiniować jeden konstruktor, pozwalający klientom na inicjowanie obiektu, tak jak struktura `Address` definiowała konstruktor pozwalający klientom zainicjować adres. Utworzenie rozsądnego zestawu konstruktorów często jest najprostszym podejściem.

Po drugie, możemy utworzyć metody fabrykujące do inicjowania struktury. Metody takie sprawiają, że łatwiejsze jest tworzenie typowych wartości.

Typ `Color` w .NET Framework stosuje się do tej strategii przy inicjowaniu kolorów systemowych. Statyczne metody `Color.FromKnownColor()` oraz `Color.FromName()` zwracają kopię wartości koloru, która reprezentuje bieżącą wartość wskazanego koloru systemowego.

Po trzecie, możemy utworzyć zmienną klasę towarzyszącą dla tych instancji, w których do pełnego skonstruowania typu potrzebne są wielokrokowe operacje. Klasa `string` w .NET wykorzystuje tę strategię poprzez klasę `System.Text.StringBuilder`. Klasy `StringBuilder` używamy do tworzenia łańcuchów poprzez wiele operacji. Po wykonaniu wszystkich działań koniecznych do zbudowania łańcucha możemy wydobyć niezmienny wynikowy łańcuch z obiektu `StringBuilder`.

Typy niezmienne są łatwiejsze w kodowaniu i utrzymywaniu. Nie należy bezmyślnie tworzyć akcesorów `get` i `set` dla każdej właściwości w naszych typach. Zamiast tego, gdy typy mają przechowywać dane, powinniśmy je implementować jako niezmienne, atomowe typy wartościowe. Następnie z tych jednostek możemy łatwo budować bardziej złożone struktury.

Punkt 4: Rozróżnianie pomiędzy typami wartościowymi a referencyjnymi

Typy wartościowe czy referencyjne? Struktury czy klasy? Kiedy należy użyć których? C# to nie C++, w którym wszystkie typy definiujemy jako wartościowe, a potem możemy tworzyć referencje do nich. C# nie jest również Javą, w której wszystko jest typem referencyjnym (chyba że jest się jednym z projektantów języka). Musimy zdecydować, jak będą zachowywać się wszystkie instancje naszego typu, kiedy go tworzymy. To ważne, aby od razu podjąć właściwą decyzję. Będziemy musieli żyć z konsekwencjami tej decyzji, gdyż późniejsza zmiana może (i tak prawdopodobnie będzie) spowodować, że całkiem spora część kodu „rozpadnie się" w subtelny sposób. W chwili tworzenia typu jest po prostu decyzja, czy wybrać słowo `struct`, czy `class`, ale jeśli postanowimy zmienić to później, czeka nas wiele pracy przy uaktualnianiu całego kodu klienckiego używającego tego typu.

Dokonanie najlepszego wyboru nie jest czymś tak prostym, jak preferowanie jednego lub drugiego podejścia. Właściwy wybór zależy od tego, jak planujemy używać nowego typu. Typy wartościowe nie są polimorficzne, zatem lepiej nadają się do przechowywania danych, które przekształca nasza aplikacja. Typy referencyjne mogą być polimorficzne i powinny być używane do definiowania zachowania naszej aplikacji. Trzeba rozważyć oczekiwane