O'REILLY®

C# 10 Programowanie

Tworzenie aplikacji Windows, internetowych i biurowych

Helion

Ian Griffiths

Tytuł oryginału: Programming C# 10: Build Cloud, Web, and Desktop Applications

Tłumaczenie: Piotr Rajca

ISBN: 978-83-8322-206-6

Authorized Polish translation of the English edition of *Programming C# 10*
ISBN 9781098117818 © 2022 Ian Griffiths.

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/c10pro
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 1.
Prezentacja C#

Język programowania C# (wymawiane jako „C szarp") jest używany do tworzenia wielu rodzajów aplikacji, w tym witryn internetowych, aplikacji działających w chmurze, aplikacji działających w urządzeniach internetu rzeczy, aplikacji dla komputerów stacjonarnych, aplikacji na telefony oraz narzędzi uruchamianych z poziomu wiersza poleceń. C# wraz ze swym systemem uruchomieniowym, bibliotekami oraz narzędziami tworzącymi wspólnie środowisko określane jako .NET już niemal od dwudziestu lat ma największe znaczenie dla programistów tworzących aplikacje dla systemu Windows. Obecnie środowisko .NET jest produktem wieloplatformowym i jednocześnie projektem otwartoźródłowym, który pozwala aplikacje i usługi napisane w języku C# uruchamiać nie tylko w systemie Windows, lecz także w systemach Android, iOS, macOS oraz Linux.

Udostępnienie wersji 10.0 języka C# oraz odpowiadającej jej wersji 6.0 środowiska uruchomieniowego .NET stanowi ważny kamień milowy: C# stał się w pełni wieloplatformowym językiem udostępnianym jako otwarte oprogramowanie. Choć implementacje open source były dostępne dla niemal wszystkich wersji języka C#, to jednak kluczowa zmiana nastąpiła w 2016 roku, kiedy to firma Microsoft udostępniła .NET Core 1.0 — pierwszą platformę, która w pełni wspierała korzystanie z C# nie tylko w systemie Windows, lecz także w systemach Linux i macOS. Wsparcie dla biblioteki i narzędzi .NET Core było początkowo niejednolite, dlatego Microsoft wciąż udostępniał nowe wersje wcześniejszego, własnościowego środowiska uruchomieniowego, .NET Framework, przeznaczonego wyłącznie dla systemu Windows. Jednak obecnie, po sześciu latach, to stare środowisko w praktyce zostało wycofane[1], gdyż w całości zastąpiła je nowsza, wieloplatformowa wersja. W .NET 5.0 zrezygnowano ze słowa „Core" w nazwie, zaznaczając w ten sposób, że teraz jest to podstawowa wersja środowiska, ale jego w pełni wieloplatformowa wersja pojawiła się dopiero w .NET 6.0, ponieważ dopiero ono uzyskała status pełnego **wsparcia długoterminowego** (ang. *Long Time Support* — LTS). Po raz pierwszy wersja C# i .NET niezależna od platformy systemowej zastąpiła stare środowisko .NET Framework.

Obecnie C# i .NET są projektami otwartoźródłowymi, jednak wcześniej było inaczej. W początkowym okresie istnienia języka C# firma Microsoft pilnie strzegła jego kodów źródłowych, ale w 2014 roku w celu przyspieszenia i popularyzacji tworzenia projektów open source na platformie .NET stworzono .NET Foundation (*https://dotnetfoundation.org*), która obecnie zarządza wieloma

[1] Ta stara wersja środowiska .NET Framework będzie wspierana jeszcze przez wiele lat, jednak Microsoft ogłosił, że nie będą do niej dodawane żadne nowe możliwości.

spośród najistotniejszych projektów związanych z językiem C# i platformą .NET rozwijanych przez Microsoft. Do tych projektów należą kompilator C# (*https://github.com/dotnet/roslyn*) oraz pakiet .NET (*https://github.com/dotnet/runtime*) obejmujący środowisko uruchomieniowe i bibliotekę klas. Obecnie niemal wszystko, co wiąże się z językiem C#, jest rozwijane jako projekty open source, a wkład zewnętrzny jest bardzo mile widziany. Nowe propozycje rozwoju języka są udostępniane w serwisie GitHub, co umożliwia zaangażowanie społeczności programistów w rozwój C# już od najwcześniejszych etapów.

Dlaczego C#?

Choć C# można używać na wiele sposobów, to zawsze istnieje możliwość wyboru innego języka programowania. Niby dlaczego mielibyśmy wybrać właśnie C#? Wszystko zależy od tego, co chcemy zrobić, oraz od tego, jakie możliwości i cechy języka programowania lubimy, a jakich nie lubimy. Osobiście uważam, że C# zapewnia znaczące możliwości, elastyczność i wydajność, a przy tym działa na wystarczająco wysokim poziomie abstrakcji, by nie trzeba było poświęcać znacznego wysiłku na niewielkie, szczegółowe problemy, które nie są bezpośrednio powiązane z problemami, jakie stara się rozwiązać tworzony program.

Znaczna część potęgi C# pochodzi z szerokiego zakresu technik programistycznych, które język ten udostępnia. Jest to język obiektowy, udostępnia typy ogólne oraz możliwość programowania funkcyjnego. Pozwala na stosowanie zarówno typowania dynamicznego, jak i statycznego. Dzięki technologii LINQ (ang. *Language Integrated Query*) udostępnia bogate możliwości operacji na listach i zbiorach. Dysponuje ona przy tym wbudowanym wsparciem dla programowania asynchronicznego. Oprócz tego liczne środowiska programistyczne obsługujące język C# udostępniają szeroki zakres różnorodnych narzędzi i możliwości poprawiających efektywność pracy programistów.

C# udostępnia opcje umożliwiające równoważenie łatwości pisania kodu i jego wysokiej jakości. Środowisko uruchomieniowe tego języka zawsze udostępniało **mechanizm oczyszczania pamięci** (ang. *garbage collector*, w skrócie GC), w przeważającej większości przypadków zwalniający programistów z konieczności samodzielnego odzyskiwania pamięci, która nie jest już używana przez program. Takie mechanizmy oczyszczania pamięci są rozwiązaniem powszechnie stosowanym w nowoczesnych językach programowania i choć są dobrodziejstwem dla większości programów, to jednak są pewne wyspecjalizowane scenariusze, w których ich wydajność może być problematyczna. Dlatego C# udostępnia możliwości pozwalające na bardziej jawne zarządzanie pamięcią i na rezygnację z łatwości programowania na rzecz wydajności działania aplikacji, jednak bez utraty bezpieczeństwa, jakie zapewnia kontrola typów. Modyfikacje te pozwoliły na rozpoczęcie stosowania języka C# do tworzenia aplikacji, dla których krytyczna jest wydajność i które od lat tradycyjnie już były tworzone w mniej bezpiecznych językach, takich jak C oraz C++.

Języki programowania nie istnieją w próżni — niezwykle istotne są także wysokiej jakości biblioteki zapewniające szeroką gamę możliwości. Istnieją niezwykle eleganckie i akademicko piękne języki, które są wprost cudowne aż do chwili, kiedy spróbujemy użyć ich do zrobienia czegoś trywialnego, takiego jak wymiana informacji z bazą danych lub określenie, gdzie można przechować ustawienia użytkownika. Niezależnie od tego jak mocny jest zestaw idiomów programistycznych

oferowany przez dany język, musi on także zapewniać pełny i wygodny dostęp do usług platformy systemowej. Dzięki swojemu środowisku uruchomieniowemu, wbudowanej bibliotece klas oraz niezwykle obszernej bibliotece klas tworzonych przez innych twórców język C# jest pod tym względem niezwykle mocny.

.NET obejmuje zarówno środowisko uruchomieniowe, jak i biblioteki klas, z których korzystają programy pisane w języku C#. Część uruchomieniowa .NET nosi nazwę *Common Language Runtime* (i jest zazwyczaj określana skrótowo jako CLR). Jej nazwa odzwierciedla fakt, że nie obsługuje ona wyłącznie C#, lecz wszystkie języki programowania używane w .NET. Na przykład Microsoft udostępnia takie języki jak Visual Basic i F# oraz dostosowaną do .NET wersję języka C++. CLR dysponuje specjalnym systemem typów — *Common Type System* (w skrócie CTS) — który sprawia, że kod pisany w różnych językach może ze sobą bez przeszkód współpracować; a to z kolei oznacza, że biblioteki .NET zazwyczaj mogą być używane w kodzie pisanym w dowolnym języku — F# może używać bibliotek napisanych w C#, C# może używać bibliotek Visual Basica i tak dalej.

Oprócz środowiska uruchomieniowego istnieje także biblioteka klas. Wraz z upływem lat miała ona kilka różnych nazw: Base Class Library (BCL), Framework Class Library oraz biblioteki frameworku, jednak obecnie wydaje się, że Microsoft zdecydował się określać ten element platformy .NET jako **biblioteki środowiska uruchomieniowego** (ang. *runtime libraries*). Biblioteki te udostępniają klasy pozwalające stosować wiele mechanizmów systemu operacyjnego, jak również udostępniają sporo innych rozwiązań funkcjonalnych, takich jak klasy kolekcji czy też narzędzia do przetwarzania formatu JSON.

Biblioteka klas wbudowana w .NET to jednak jeszcze nie wszystko — wiele innych platform udostępnia własne biblioteki klas przeznaczonych dla tej platformy. Na przykład dostępne są rozbudowane biblioteki pozwalające programom pisanym w C# na korzystanie z popularnych usług chmurowych. Zgodnie z tym, czego można się spodziewać, firma Microsoft udostępnia wyczerpujące biblioteki .NET przeznaczone do wykorzystania z usługami działającymi na platformie Azure. I podobnie firma Amazon udostępnia równie wyczerpujący pakiet SDK do wykorzystania w programach pisanych w C# oraz innych językach.NET z platformy Amazon Web Services (AWS). Jednak biblioteki wcale nie muszą być powiązane z konkretnymi platformami czy usługami. Istnieje obszerny ekosystem bibliotek .NET, zarówno komercyjnych, jak i dostępnych bezpłatnie, obejmujących rozwiązania matematyczne, do parsowania, komponenty interfejsu użytkownika oraz wiele innych. Nawet jeśli okaże się, że mamy pecha i musimy korzystać z mechanizmów systemu operacyjnego, dla których nie ma klas w bibliotece .NET, to język C# udostępnia mechanizmy pozwalające na stosowanie innych **interfejsów programowania aplikacji** (w skrócie **API**), takich jak stare API Win32, macOS oraz Linux bądź też API COM (Component Object Model) stosowane w systemie Windows.

Oprócz bibliotek dostępne są także liczne frameworki do tworzenia aplikacji. .NET zawiera wbudowane frameworki do tworzenia aplikacji i API internetowych, aplikacji biurowych oraz mobilnych. Dostępne są także frameworki wspierające różne style tworzenia systemów rozproszonych, takie jak Reaqtor (*https://reaqtive.net*), przeznaczony do obsługi ogromnych ilości zdarzeń, czy też globalnych systemów o bardzo dużej dostępności, takie jak Orleans (*https://dotnet.github.io/orleans/*).

I w końcu, ze względu na fakt, że platforma .NET istnieje i jest stosowana już od przeszło dwóch dekad, wiele organizacji zainwestowało w technologie bazujące na niej. Dlatego też język C# jest zazwyczaj naturalnym wyborem, jeśli chcemy czerpać korzyści z tych inwestycji.

Podsumowując, C# zapewnia bardzo obszerny zbiór abstrakcji wbudowanych w sam język, potężne środowisko uruchomieniowe oraz łatwy dostęp do niezwykle dużej liczby bibliotek i narzędzi ułatwiających korzystanie z możliwości funkcjonalnych platformy.

Kod zarządzany i CLR

C# był pierwszym językiem zaprojektowanym jako rodzimy język CLR. Dzięki temu C# ma unikalny charakter. Oznacza to także, że aby zrozumieć C#, trzeba zrozumieć CLR oraz to, w jaki sposób wykonuje ono kod.

Przez lata najczęstszym sposobem działania kompilatorów było przetwarzanie kodu źródłowego i generowanie wyników, których postać pozwalała na ich bezpośrednie wykonanie przez procesor komputera. Kompilatory generowały zatem **kod maszynowy** (ang. *machin code*) — serię instrukcji zapisanych w odpowiednim binarnym formacie wymaganym przez konkretny rodzaj procesora używanego w komputerze. Wiele kompilatorów wciąż działa właśnie w taki sposób, jednak kompilator C# do nich nie należy. Zamiast tego kompilator ten działa w modelu bazującym na generowaniu tak zwanego **kodu zarządzanego** (ang. *managed code*).

W przypadku kodu zarządzanego to środowisko uruchomieniowe, a nie kompilator generuje kod maszynowy wykonywany następnie przez procesor. Dzięki temu środowisko uruchomieniowe jest w stanie dostarczać usług, których udostępnianie w tradycyjnym modelu działania byłoby trudne lub nawet niemożliwe. Kompilator generuje pośrednią formę kodu binarnego, tak zwany **język pośredni** (ang. *intermediate language*, w skrócie: *IL*), natomiast środowisko uruchomieniowe tworzy wykonywalny kod binarny w trakcie działania programu.

Być może najbardziej zauważalną korzyścią, jaką zapewnia model bazujący na użyciu kodu pośredniego, jest to, że wyniki generowane przez kompilator nie są powiązane z żadną konkretną architekturą procesorów. Na przykład procesory używane w większości nowoczesnych komputerów obsługują zarówno 32-, jak i 64-bitowe zbiory instrukcji (które z powodów historycznych są określane, odpowiednio, jako instrukcje **x86** i **x64**). W przypadku starego modelu kompilacji kodu źródłowego do języka maszynowego trzeba wybrać, który z tych zestawów instrukcji ma być obsługiwany i, jeżeli chcemy, by komponent działał na procesorach o różnych architekturach, zbudować jego wiele wersji. Dzięki .NET można natomiast napisać pojedynczy komponent, który bez żadnych modyfikacji będzie działał na procesorach o architekturze zarówno x86, jak i x64. Co więcej, ten sam komponent można by nawet uruchamiać na procesorach o zupełnie innej architekturze (jest ona powszechnie stosowana w urządzeniach mobilnych, nowszych komputerach Mac, jak również niewielkich komputerach, takich jak Raspberry Pi). W przypadku języka, którego kod jest kompilowany bezpośrednio do kodu maszynowego, konieczne byłoby wygenerowanie osobnych plików binarnych dla każdej z tych architektur, choć w niektórych przypadkach można by także wygenerować jeden plik zawierający kopie kodu dla każdej z obsługiwanych architektur. .NET pozwala natomiast skompilować jeden komponent, zawierający tylko jedną wersję kodu, który nie tylko będzie mógł działać w tych wszystkich architekturach, lecz także w architekturach, które nawet nie

istniały w momencie, gdy był on tworzony (oczywiście zakładając, że zostanie dla nich opracowane odpowiednie środowisko uruchomieniowe). (Na przykład komponenty .NET napisane wiele lat przed pojawieniem się pierwszych komputerów Apple z procesorami ARM mogą na nich działać rodzimie bez korzystania z technologii Rosetta (*https://pl.wikipedia.org/wiki/Rosetta_(oprogramowanie)*), która normalnie zapewnia możliwość uruchamiania starego kodu na nowych procesorach). Ujmując rzecz bardziej ogólnie, jeśli tylko pojawi się jakiekolwiek usprawnienie w używanym przez CLR sposobie generowania kodu — czy to obsługa nowej architektury procesorów, czy jakieś usprawnienie wydajności — to wszystkie języki programowania .NET natychmiast będą mogły z niego skorzystać. Na przykład starsze wersje CLR nie korzystały z rozszerzeń do przetwarzania wektorowego, w które są wyposażone nowoczesne procesory x86 oraz x64, natomiast nowsze wersje CLR będą z nich zazwyczaj korzystać podczas generowania kodu pętli. Z możliwości tych korzysta cały kod działający w bieżącej wersji .NET, w tym także taki, który został napisany i zbudowany wiele lat przed udostępnieniem tych usprawnień.

Sam moment, w którym CLR generuje wykonywalny kod maszynowy, może się zmieniać. Zazwyczaj wykorzystywane jest podejście nazywane kompilacją *just in time* (JIT), w którym każda funkcja jest kompilowana do postaci kodu maszynowego w trakcie działania programu, przed jej pierwszym wywołaniem. Niemniej jednak mogą być używane także inne rozwiązania. Jedna z implementacji środowiska uruchomieniowego, Mono, może interpretować kod IL bezpośrednio, bez konwertowania go do nadającego się do wykonywania kodu maszynowego, co może być przydatne na platformach takich jak iOS, w których ograniczenia prawne mogą uniemożliwiać stosowanie kompilacji JIT. .NET SDK (*Software Development Kit*) udostępnia także narzędzie o nazwie crossgen, pozwalające budować prekompilowany kod oprócz kodu IL. Ten rodzaj kompilacji, nazywany **kompilacją z wyprzedzeniem** (ang. *ahead of time*, w skrócie **AoT**), może skrócić czas uruchamiania aplikacji. Dostępne jest także całkowicie odrębne środowisko uruchomieniowe o nazwie .NET Native, obsługujące wyłącznie kompilację wstępną; jest ono używane przez aplikacje Windows Store Apps, tworzone z myślą o platformie Universal Windows Platform (UWP). (Microsoft ogłosił jednak, że środowisko .NET Native, przeznaczone wyłącznie dla systemu Windows, zostanie w przyszłości wycofane i zastąpi je NativeAOT, jego wieloplatformowy następca).

Jednak nawet w przypadku stosowania narzędzi takich jak crossgen generowanie kodu wykonywalnego i tak może następować podczas wykonywania aplikacji — mechanizm **kompilacji warstwowej** (ang. *tiered compilation*) środowiska uruchomieniowego może zdecydować o ponownym, dynamicznym skompilowaniu metody w celu zoptymalizowania jej pod kątem sposobu, w jaki jest używana w środowisku, a co więcej, może to zrobić niezależnie od tego, czy używana jest kompilacja JIT, czy AoT[2].

W kodzie zarządzanym wszechobecne są informacje o typach. Środowisko uruchomieniowe .NET wymaga zamieszczania tych informacji, gdyż umożliwiają one stosowanie pewnych mechanizmów. Na przykład .NET udostępnia różne automatyczne usługi do serializacji danych; pozwalają one na zapisywanie obiektów w formie binarnych lub tekstowych reprezentacji ich stanu, które następnie można ponownie przekształcić na obiekty, być może nawet na innym komputerze.

[2] Ani .NET Native, ani NativeAOT tego nie robią — zostały one zaprojektowane specjalnie pod kątem unikania kompilacji JIT, dlatego nie udostępniają możliwości kompilacji warstwowej.

Działanie takich usług bazuje na pełnym i precyzyjnym opisie struktury obiektu — czyli informacjach, których dostępność w kodzie zarządzanym jest zagwarantowana. Jednak informacje o typach danych mogą być używane także na inne sposoby. Na przykład platformy do przeprowadzania testów jednostkowych mogą ich używać do sprawdzania kodu w projektach testowych i odnajdywania wszystkich napisanych testów. Operacje tego typu bazują na udostępnianych przez CLR usługach **odzwierciedlania** (ang. *reflection*), które zostały opisane w rozdziale 13.

Choć ścisły związek C# ze środowiskiem uruchomieniowym jest jedną z jego najważniejszych cech, to jednak nie jest jedyną. Taki projekt języka C# ma swoje uzasadnienie.

Ogólność jest preferowana względem specjalizacji

C# preferuje możliwości ogólnego przeznaczenia, a nie te bardziej wyspecjalizowane. Obecnie dostępna jest już 10. główna wersja języka, a w każdej, projektując nowe możliwości, jego twórcy zawsze mieli na myśli konkretne scenariusze. Pomimo to dokładali wszelkich starań, by każdy dodawany element języka był przydatny także poza tymi podstawowymi scenariuszami.

Na przykład kilka lat temu twórcy C# zdecydowali o dodaniu do niego mechanizmów, których celem było zapewnienie poczucia, że dostęp do baz danych jest lepiej zintegrowany z językiem. W efekcie została stworzona technologia *Language Integrated Query* (LINQ, opisana dokładniej w rozdziale 10.), która bez wątpienia spełnia ten cel, choć twórcom C# udało się to osiągnąć bez dodawania do języka jakiejkolwiek bezpośredniej obsługi dostępu do danych. Zamiast tego dodano grupę pozornie różnorodnych możliwości. Można do nich zaliczyć lepsze wsparcie dla idiomów programowania funkcyjnego, możliwość dodawania nowych metod do istniejących typów bez konieczności stosowania dziedziczenia, obsługę typów anonimowych, możliwość pobierania modelu obiektów reprezentującego strukturę wyrażenia oraz określenie składni zapytań. Ostatnia z tych możliwości jest w oczywisty sposób związana z dostępem do danych, jednak w przypadku pozostałych wskazanie takiego powiązania jest znacznie trudniejsze. Pomimo to wszystkich tych rozwiązań można używać wspólnie, znacznie sobie w ten sposób ułatwiając realizację niektórych zadań związanych z dostępem do danych. Jednak każda z tych możliwości jest także bardzo użyteczna sama w sobie, dzięki czemu poza dostępem do danych można ich także używać w wielu innych sytuacjach. Na przykład C# znacznie ułatwia przetwarzanie list, zbiorów oraz wszelkich innych grup obiektów, gdyż nowe możliwości pozwalają operować na kolekcjach obiektów pochodzących z dowolnych źródeł, a nie tylko z baz danych.

Jednym z przykładów tej filozofii preferującej ogólność jest pewna funkcjonalność, która została opracowana z myślą o języku C#, lecz której ostatecznie jej twórcy nie zdecydowali się wprowadzić. Miała ona pozwalać na umieszczanie kodu XML bezpośrednio w kodzie programu, dodając w ten sposób wyrażenia, które w trakcie działania programu będą używane do wyliczania wartości umieszczanych następnie w treści XML. Mechanizm ten kompilował taki kod do postaci kodu, który w trakcie działania programu generował kompletny kod XML. Dział badań firmy Microsoft zademonstrował to rozwiązanie publicznie, niemniej nie zostało one dodane do języka C#, choć nieco później zostało udostępnione w innym języku rozwijanym przez Microsoft, Visual Basicu, który dodatkowo został wyposażony w pewne wyspecjalizowane mechanizmy zapytań, służące do pobierania informacji z dokumentów XML. Jeśli chodzi o pobieranie danych z kodu XML,

język C# udostępnia te możliwości za pośrednictwem technologii LINQ bez konieczności wprowadzania jakichkolwiek rozwiązań powiązanych bezpośrednio z XML-em. Popularność XML-a znacznie przygasła, od kiedy zaczęto kwestionować jego koncepcję, co nastąpiło, gdy okazało się, że pod wieloma względami znacznie lepszy jest format JSON (choć bez wątpienia za kilka lat także i on straci popularność na korzyść jakiegoś innego rozwiązania). Gdyby możliwość osadzania kodu XML trafiła jednak do języka C#, to aktualnie byłaby traktowana jako nieco anachroniczna ciekawostka.

Nowe możliwości wprowadzane w kolejnych wersjach języka C# są zgodne z tą filozofią. Na przykład mechanizmy **dekonstrukcji** (ang. *deconstruction*) i dopasowywania wzorców dodane w nowszych wersjach C# mają za zadanie ułatwiać programistom życie w sposób subtelny i użyteczny, przy czym możliwości ich wykorzystania nie ograniczają się do żadnego konkretnego obszaru zastosowań.

Standardy oraz implementacje języka C#

Zanim będziemy mogli zacząć pisać jakikolwiek kod, musimy wiedzieć, jakiej wersji C# oraz środowiska uruchomieniowego będziemy używać. ECMA, instytucja zajmująca się standardami, przygotowała specyfikacje definiujące język C# oraz jego środowisko uruchomieniowe (są to odpowiednio specyfikacje ECMA-334 i ECMA-335). Umożliwiły one powstanie wielu implementacji samego języka C#, jak i jego środowiska uruchomieniowego. W czasie przygotowywania niniejszej książki dostępne są cztery powszechnie używane implementacje: Mono, .NET Native, .NET (wcześniej znane pod nazwą .NET Core) oraz .NET Framework. Choć to nieco mylące, wszystkie cztery są obecnie rozwijane przez Microsoft, mimo że początkowo sytuacja była nieco inna.

Wiele .NET-ów

Projekt Mono powstał w 2001 roku, przy czym początkowo nie był tworzony przez Microsoft. (To z tego powodu w jego nazwie nie ma „.NET" — może on używać nazwy C#, gdyż jest to nazwa języka używana w standardzie, jednak w czasach przed powołaniem .NET Foundation .NET był nazwą marki używaną jedynie przez Microsoft). Projekt Mono powstał w celu umożliwienia pisania aplikacji dla systemu Linux w języku C#, lecz został następnie rozszerzony na inne platformy systemowe — iOS i Android. Ten kluczowy krok pozwolił projektowi Mono znaleźć swoją niszę rynkową, gdyż obecnie jest on używany głównie do tworzenia w języku C# wieloplatformowych aplikacji mobilnych. Obecnie Mono udostępnia także kompilację kodu dla WebAssembly (WASM) i zawiera implementację CLR, która może działać w każdej przeglądarce zgodnej ze standardami, umożliwiając wykonywanie kodu C# po stronie klienta w ramach aplikacji internetowych. Rozwiązanie to jest często stosowane wraz z frameworkiem .NET o nazwie Blazor, który pozwala tworzyć interfejs użytkownika w HTML oraz implementować zachowania aplikacji w C#. Połączenie Blazora i WASM stanowi kombinację umożliwiającą także stosowanie języka C# podczas używania z takich frameworków jak Electron, korzystających z klienckich technologii internetowych do tworzenia wieloplatformowych aplikacji biurowych. (Blazor nie wymaga stosowania WASM — może także współdziałać z kodem C# kompilowanym w normalny sposób i wykonywanym w środowisku uruchomieniowym .NET; to standardowe rozwiązanie dla Multi-platform App UI (MAUI), technologii .NET umożliwiającej pisanie jednej aplikacji, którą można będzie uruchamiać w systemach Android, iOS, macOS oraz Windows).

Mono od samego początku był projektem open source, wspieranym przez wiele różnych firm. W 2016 roku Microsoft wykupił firmę Xamarin, która wcześniej zajmowała się rozwojem tego projektu. Obecnie Xamarin pozostaje odrębną marką, a Microsoft dba, by była ona kojarzona z tworzeniem w języku C# wieloplatformowych aplikacji przeznaczonych do uruchamiania na urządzeniach mobilnych. Podstawowa technologia Mono została dodana do bazy kodu .NET Microsoftu. Moment ten wyznaczył zakończenie kilkuletniego procesu, w trakcie którego Mono stopniowo miało coraz więcej wspólnego z .NET. Początkowo Mono udostępniało własne implementacje wszystkich składników platformy: kompilatora C#, bibliotek oraz CLR. Jednak kiedy Microsoft udostępnił swój kompilator C# jako projekt otwartoźródłowy, narzędzia Mono zaczęły z niego korzystać. Mono dysponowało wcześniej własną, kompletną implementacją bibliotek uruchomieniowych .NET, jednak kiedy Microsoft udostępnił .NET Core, oprogramowanie open source, Mono stopniowo zaczęło się coraz bardziej od niego uzależniać. Obecnie Mono jest praktycznie jedną z dwóch implementacji CLR dostępnych w repozytorium środowisk uruchomieniowych .NET i zapewnia wsparcie dla tworzenia aplikacji przeznaczonych dla środowisk mobilnych oraz WebAssembly.

A co z pozostałymi trzema implementacjami, z których każda jest określana jako .NET? .NET Native jest używana w aplikacjach UWP i, zgodnie z informacjami podanymi wcześniej w tym rozdziale, stanowi wyspecjalizowaną wersję .NET wspierającą wyłącznie kompilację AoT. Niemniej jednak .NET Native ma zostać zastąpiona przez NativeAOT, która ma się stać przyszłością .NET, a nie jej odrębną wersją. Oznacza to, że istnieją dwie wersje platformy, których los nie został jeszcze przesądzony: .NET Framework (tylko dla systemu Windows i jej kod źródłowy nie jest otwarty) oraz .NET (wieloplatformowa i rozwijana jako projekt open source, wcześniej nosząca nazwę .NET Core). Jednak, jak już wspomniałem, Microsoft nie planuje dodawania do .NET Framework żadnych nowych możliwości, przez co w praktyce .NET 6.0 jest jedyną aktualną wersją platformy.

To dążenie do uzyskania tylko jednej bieżącej wersji było jednym z głównych celów prowadzenia .NET 6, co sprawia, że ta wersja ma szczególne znaczenie. Jednak inne wersje też warto znać, gdyż wciąż można znaleźć działające systemy, które z nich korzystają. Jednym z powodów nieustającej popularności .NET Framework jest to, że wciąż dysponuje on kilkoma możliwościami, których nie ma w .NET 6.0. .NET Framework działa tylko w systemie Windows, natomiast .NET 6.0 obsługuje systemy Windows, macOS oraz Linux i choć sprawia to, że .NET Framework nie ma aż tak szerokiego zastosowania, to równocześnie oznacza, że pozwala on korzystać z kilku możliwości dostępnych wyłącznie w systemie Windows. Na przykład jedna z części biblioteki klas .NET Framework jest przeznaczona do obsługi technologii COM+ Component Services, funkcjonalności systemu operacyjnego Windows służącej do uruchamiania komponentów integrujących się z produktem Microsoft Transaction Server. Możliwości te nie są dostępne w nowszych, wieloplatformowych wersjach .NET, gdyż kod mógłby być wykonywany w systemie Linux, który nie zapewnia analogicznych możliwości bądź są one zbyt odmienne, by można było korzystać z nich przy użyciu tego samego API .NET.

W kilku ostatnich wydaniach platformy liczba możliwości dostępnych wyłącznie w .NET Framework uległa drastycznemu zmniejszeniu, gdyż Microsoft dokładał starań, by nawet aplikacje przeznaczone tylko dla systemu Windows mogły używać najnowszej wersji .NET 6.0. Na przykład biblioteka `System.Speach` była niegdyś dostępna wyłącznie w .NET Framework, ponieważ umożliwiała korzystanie z funkcjonalności związanych z rozpoznawaniem i syntezą mowy dostępnych

wyłącznie w systemie Windows, obecnie natomiast jest ona dostępna także w bibliotece .NET 6.0. Biblioteka ta działa wyłącznie w systemie Windows, jednak jej dostępność oznacza, że korzystający z niej twórcy aplikacji mogą przejść z .NET Framework do .NET. Pozostałe możliwości .NET Framework, których nie przeniesiono do innych wersji platformy, są zazwyczaj używane na tyle rzadko, że nakład pracy konieczny do ich przeniesienia byłby nieuzasadniony. Wsparcie dla COM+ nie ograniczało się do biblioteki — miało także implikacje związane ze sposobem wykonywania kodu przez CLR, zatem obsługiwanie ich w nowoczesnych wersjach .NET wiązałoby się z kosztami zbyt wysokimi w stosunku do ich wykorzystania.

W ciągu kilku ostatnich lat zdecydowanie najwięcej pracy włożono w rozwój wieloplatformowej wersji .NET. .NET Framework wciąż jest obsługiwany, lecz od pewnego czasu traci na znaczeniu. Na przykład ASP.NET Core — framework do tworzenia aplikacji internetowych — już w 2019 roku przestał być wspierany na platformie .NET Framework. Wszystko to oznacza, że emerytura dla .NET Framework i uznanie .NET 6.0 za jedyną prawdziwą wersję platformy są nieuniknionym zakończeniem procesu trwającego już od kilku lat.

Cykle wydawnicze i wsparcie długoterminowe

Obecnie Microsoft udostępnia jedną wersję .NET każdego roku, zazwyczaj w listopadzie lub grudniu, ale nie wszystkie te wersje są porównywalne. Niektóre z nich dysponują **wsparciem długoterminowym** (ang. *Long Time Support*, LTS), co oznacza, że Microsoft będzie je wspierać przez okres przynajmniej trzech lat. W tym czasie wszystkie narzędzia, biblioteka oraz środowisko uruchomieniowe będą regularnie aktualizowane poprzez wydawanie „łatek" (ang. *patches*) bezpieczeństwa. Platforma .NET 6.0 została wydana w październiku 2021 roku i jest wersją LTS. Poprzednią wersją LTS platformy była .NET Core 3.1, udostępniona w grudniu 2019 roku, co oznacza, że będzie wspierana aż do grudnia 2022 roku. Jeszcze wcześniejszą wersją LTS platformy była .NET Core 2.1, której wsparcie zakończyło się we wrześniu 2021 roku[3].

A co z wersjami, które nie mają wsparcia długoterminowego? Są one wspierane od momentu wydania do momentu, w którym upływa sześć miesięcy od wydania kolejnej wersji LTS. Na przykład wersja .NET 5.0 była wspierana od wydania w grudniu 2020 roku aż do maja 2022 roku, kiedy minęło sześć miesięcy od wydania .NET 6.0. Oczywiście może się zdarzyć, że Microsoft zdecyduje się na przedłużenie wsparcia dla którejś wersji .NET, jednak na potrzeby planowania warto przyjąć, że wszystkie wersje niemające wsparcia długoterminowego (określane potocznie jako wersje nie-LTS) przestają nadawać się do użycia po 18 miesiącach od daty wydania.

Pełne wykorzystanie nowej wersji .NET przez ekosystem zajmuje czasami nawet kilka miesięcy. W praktyce zapewne nie będziesz w stanie skorzystać z najnowszej wersji .NET już w dniu jej wydania, gdyż dostawca platformy chmurowej nie będzie jej jeszcze obsługiwać bądź nie będzie ona zgodna z używanymi bibliotekami. Czynniki te znacząco skracają okres, w jakim w praktyce

[3] Jeśli zastanawiasz się, w jaki sposób te numery wersji oraz daty ich wydań pasują do corocznych, naprzemiennych wydań platformy, to wiedz, że aktualny harmonogram wydań został wprowadzony wraz z pojawieniem się wersji .NET Core 3.1 oraz że nie było żadnej wersji .NET Core 4. Wraz ze zmianą nazwy .NET Core na .NET kolejnym numerem wersji po 3.1 był 5.0, co miało podkreślić odejście do .NET Framework, którego ostatnia wersja miała numer 4.8.

można korzystać z wersji nie-LTS, i mogą sprawić, że będziemy mieć dość mało czasu na zaktualizowanie aplikacji przed udostępnieniem kolejnej wersji platformy. Jeśli czas dostosowywania wszelkich używanych narzędzi, platform oraz bibliotek do nowej wersji .NET wynosi kilka miesięcy, to będziesz mieć bardzo niewiele czasu na jej wykorzystanie, nim okres jej wspierania się zakończy. W ekstremalnych przypadkach to „okno czasowe", w którym można skorzystać z wersji .NET, w ogóle się nie pojawi — wsparcie dla .NET Core 2.2 zakończyło się, zanim usługa Azure Functions zapewniła pełne wsparcie dla .NET Core 3.0 lub 3.1, zatem okazało się, że programiści, którzy w usłudze Azure Functions używali .NET Core 2.2, znaleźli się w sytuacji, w której numer ostatniej obsługiwanej wersji cofnął się, musieli więc wybierać pomiędzy powrotem do .NET Core 2.1 oraz kilkumiesięcznym stosowaniem nieobsługiwanej wersji CLR w środowisku produkcyjnym. Z tego względu wielu programistów postrzega wersje nie-LTS jako swoiste wersje demonstracyjne — można eksperymentalnie korzystać z ich nowych możliwości i oczekiwać na zastosowanie ich w wersji produkcyjnej dopiero po wydaniu kolejnej wersji LTS platformy .NET.

Użycie .NET Standard w celu tworzenia projektów działających w różnych wersjach .NET

Istnienie wielu wersji środowiska uruchomieniowego, z których każda posiada odmienne wersje biblioteki klas, stwarza problemy dla wszystkich, który chcieliby udostępniać swój kod innym programistom. Choć zmiany, które w końcu zostały wprowadzone w .NET 6.0, mogą nieco zmniejszyć ten problem, to jednak programiści jeszcze często będą chcieli wspierać systemy działające na starej wersji platformy .NET Framework. Oznacza to, że przez pewnie czas wciąż przydatne będzie tworzenie komponentów przeznaczonych dla różnych wersji środowiska uruchomieniowego. W serwisie *http://nuget.org* dostępne jest repozytorium pakietów z komponentami .NET, w którym Microsoft umieszcza wszystkie biblioteki .NET niewchodzące bezpośrednio w skład platformy i w którym większość programistów .NET publikuje i udostępnia własne biblioteki klas. Ale którą wersję platformy należy wybrać? To pytanie ma dwa aspekty. Z jednej strony istnieją implementacje środowiska uruchomieniowego (.NET, .NET Framework), a z drugiej ich wersje (na przykład .NET Core 3.1 lub .NET 6.0 albo .NET Framework 4.7.2 lub 4.8). Wielu autorów popularnych pakietów udostępnianych w serwisie NuGet jako oprogramowanie open source także zapewnia wsparcie dla wielu typów i wersji frameworków, zarówno starszych, jak i nowszych.

Programiści tworzący komponenty radzili sobie z wieloma wersjami poprzez budowanie kilku wariantów własnych bibliotek. W przypadku dystrybuowania bibliotek .NET za pośrednictwem serwisu NuGet w pakietach przeznaczonych do wykorzystania w różnych wersjach .NET można umieszczać wiele zestawów plików binarnych. Takie rozwiązanie ma jednak tę wadę, że wraz z pojawianiem się nowych wersji .NET istniejące biblioteki nie będą działać we wszystkich nowszych środowiskach uruchomieniowych. Komponent napisany z myślą o .NET Framework 4.0 będzie działać we wszystkich nowszych wersjach .NET Framework, lecz nie na .NET 6.0. Nawet jeśli kod źródłowy komponentu był w pełni zgodny z nowszym środowiskiem uruchomieniowym, konieczne było przygotowanie osobnej wersji biblioteki, skompilowanej specjalnie dla danej platformy. Jeśli autor

używanej biblioteki nie przygotował jej wersji jawnie przeznaczonej dla .NET[4], to korzystanie z tej biblioteki w danej wersji platformy .NET nie było możliwe. Taka sytuacja nie była korzystna dla nikogo. Na przestrzeni lat pojawiło się i zostało porzuconych wiele wersji .NET (takich jak Silverlight lub wersja dla Windows Phone), co oznaczało, że twórcy komponentów zostali wprzęgnięci w kierat ciągłego przygotowywania nowych wariantów swoich produktów, a ponieważ wymagało to od nich zaangażowania i czasu na wykonywanie niezbędnych prac, mogło się okazać, że komponenty, które inni programiści chcieli zastosować, i tak nie były dostępne w docelowej wersji platformy.

Aby uniknąć tego problemu, Microsoft wprowadził .NET Standard, który definiuje wspólny podzbiór zewnętrznych klas API środowiska uruchomieniowego .NET. Jeśli pakiet NuGet jest przeznaczony na przykład dla .NET Standard 1.0, to stanowi to gwarancję, że będzie go można używać w .NET Framework 4.5 lub nowszych, .NET Core 1.0 lub nowszych, .NET 5.0 lub nowszych oraz w Mono 4.6 lub nowszych. A co ważniejsze, jeśli w przyszłości pojawi się jeszcze inny wariant platformy .NET, to o ile tylko będzie on wspierał .NET Standard 1.0, istniejące komponenty będą na nim działać bez konieczności wprowadzania jakichkolwiek modyfikacji, i to nawet jeśli w czasie, gdy były tworzone, ta wersja platformy jeszcze nie istniała.

Obecnie najlepszym wyborem dla twórców komponentów, którzy chcą, by ich produkty były dostępne dla wielu wersji platformy, będzie zapewne .NET Standard 2.0, gdyż jest on obsługiwany przez wszystkie udostępnione ostatnio wersje .NET oraz zapewnia dostęp do bardzo szerokiego zestawu możliwości. Niemniej jednak liczba różnych wersji .NET obsługiwanych przez Microsoft jest obecnie dużo mniejsza niż w czasie, kiedy wprowadzono .NET Standard, dlatego jego znaczenie jest nieco mniejsze. Obecnie podstawową zaletą tworzenia komponentów zgodnych z .NET Standard jest to, że będą one działać w .NET Framework oraz .NET Core i .NET. Jeśli jednak obsługa .NET Framework nie jest konieczna, to zdecydowanie lepszym rozwiązaniem będzie tworzenie komponentów przeznaczonych dla .NET Core 3.1 oraz .NET 6.0. Uwarunkowania związane z tworzeniem oprogramowania dla .NET Standard zostały przedstawione dokładniej w rozdziale 12.

Jednak Microsoft udostępnia nie tylko język oraz różne środowiska uruchomieniowe wraz z bibliotekami klas — oferuje także środowiska programistyczne, ułatwiające pisanie, testowanie, debugowanie oraz pielęgnację kodu.

Visual Studio, Visual Studio Code oraz JetBrains Rider

Microsoft udostępnia trzy środowiska programistyczne przeznaczone dla komputerów biurowych: Visual Studio Code, Visual Studio oraz Visual Studio for Mac. Wszystkie trzy oferują pewne możliwości podstawowe, takie jak edytor kodu, narzędzia do budowania oraz debuger, jednak to Visual Studio udostępnia najbardziej rozbudowane wsparcie dla pisania programów w języku C#, niezależnie od tego, czy aplikacje te będą działać w systemie Windows, czy na innej platformie systemowej. Visual Studio jest także najstarszym z tych narzędzi — istnieje tak długo, jak sam język C# — pochodzi z czasów, kiedy C# nie był otwartym oprogramowaniem, dlatego też

[4] Albo .NET Core. Zmiany nazwy mogą wprowadzać tu pewne zamieszanie. Komponent obsługujący .NET Framework 3.1 będzie działał w .NET 5.0 oraz .NET 6.0, gdyż są to nowsze wersje tego samego środowiska uruchomieniowego; w momencie udostępniania .NET 5.0 pominięto jedynie słowo „Core" w nazwie oraz jeden numer wersji.

samo nie jest oprogramowaniem o otwartym kodzie źródłowym. Dostępne jest jednak w kilku wersjach, zaczynając do darmowej, aż po taką, której koszt może przerażać. Jednak do dyspozycji mamy nie tylko produkty Microsoftu — firma JetBrains, zajmująca się tworzeniem narzędzi dla programistów, sprzedaje doskonałe IDE dla platformy .NET o nazwie Rider, które może działać w systemach Windows, Linux oraz macOS.

Visual Studio jest zintegrowanym środowiskiem programistycznym (IDE), stworzonym jako narzędzie „kompletne". Zostało ono wyposażone nie tylko w doskonały edytor tekstów, ale także w wizualne narzędzie do tworzenia graficznych interfejsów użytkownika. Zapewnia ono głęboką integrację z systemami kontroli wersji, takimi jak git, z internetowymi systemami udostępniającymi repozytoria kodów źródłowych i śledzenie problemów oraz innymi narzędziami typu AML (Application Lifecycle Management), takimi jak GitHub lub Azure DevOps firmy Microsoft. Visual Studio udostępnia także wbudowane narzędzia do monitorowania wydajności oraz diagnostyczne. Dysponuje mechanizmami do współpracy z aplikacjami przeznaczonymi i wdrożonymi na Azure — chmurową platformę firmy Microsoft. Spośród wszystkich opisywanych tu IDE Microsoftu Visual Studio udostępnia także najbardziej rozbudowany zestaw narzędzi do refaktoringu kodu. Warto zwrócić uwagę, że Visual Studio działa wyłącznie w systemie Windows.

W 2017 roku firma Microsoft udostępniła Visual Studio for Mac. Nie jest to jednak żadna zmodyfikowana wersja Visual Studio. Początkowo była to platforma o nazwie Xamarin, zintegrowane środowisko programistyczne dla komputerów Mac przeznaczone do pisania w języku C# aplikacji mobilnych działających w środowisku uruchomieniowym Mono. Xamarin był początkowo niezależną technologią, jednak kiedy Microsoft przejął firmę, która go rozwijała (o czym wspominałem już wcześniej), i udostępnił jako produkt marki Visual Studio, został on zintegrowany z wieloma rozwiązaniami dostępnymi w wersji IDE przeznaczonej dla systemu Windows.

Rider firmy JetBrains jest pojedynczym IDE działającym w trzech systemach operacyjnych. To produkt o przeznaczeniu określonym dokładniej niż w przypadku Visual Studio — służy wyłącznie do tworzenia aplikacji dla platformy .NET. (Visual Studio obsługuje także język C++). Rider stworzono również zgodnie z filozofią narzędzia kompletnego, przy czym udostępnia on wyjątkowo rozbudowany zestaw narzędzi do refaktoryzacji kodu.

Visual Studio Code (określane skrótowo jako VS Code) zostało udostępnione po raz pierwszy w 2015 roku. Jest to program open source działający na wielu platformach systemowych — Linux, Windows i Mac. VS Code zostało stworzone w oparciu o platformę Electron i napisane w głównej mierze w języku TypeScript. (Oznacza to, że w odróżnieniu od Visual Studio naprawdę jest to ten sam program, który może działać w różnych systemach operacyjnych). VS Code jest nieco prostszym produktem niż Visual Studio: jego podstawowa instalacja nie zawiera wiele więcej niż edytor tekstów. Niemniej jednak, kiedy go uruchomimy, szybko odkryjemy, że pozwala ono na pobranie i instalowanie wielu rozszerzeń, zapewniających wsparcie dla pisania kodu w bardzo wielu językach, takich jak C#, F#, TypeScript, PowerShell, Python i inne. (Mechanizm stosowany do tworzenia tych rozszerzeń jest otwarty, więc każdy, kto tylko chce, może tworzyć i publikować własne rozszerzenia). Choć jego początkowa postać jest bardziej edytorem tekstów niż zintegrowanym środowiskiem programistycznym, to jednak model rozszerzeń, w jaki jest wyposażone VS Code, stwarza z niego naprawdę potężne narzędzie. Ogromna liczba dostępnych rozszerzeń

sprawiła, że VS Code stało się bardzo popularnych narzędziem wśród programistów, którzy nie używają języków programowania firmy Microsoft, a to z kolei doprowadziło do jeszcze większego wzrostu liczby dostępnych rozszerzeń.

Pisanie programów w języku C# najłatwiej jest zaczynać, używając Visual Studio oraz JetBrain Riders — nie trzeba w tym celu instalować jakichkolwiek rozszerzeń ani modyfikować ustawień konfiguracyjnych. Z drugiej strony Visual Studio Code jest dostępne dla szerszego grona odbiorców i dlatego użyję go w krótkiej prezentacji pisania programów w C# zamieszczonej w dalszej części tego rozdziału. Podstawowe rozwiązania i możliwości są wspólne dla wszystkich prezentowanych tu IDE, dlatego większość informacji będzie przydatna także w przypadku, gdy używasz Visual Studio lub Ridera.

Darmową wersję Visual Studio Code można pobrać ze strony *https://code.visualstudio.com*. Dodatkowo konieczne będzie także pobranie .NET SDK (*https://dotnet.microsoft.com/en-us/download*).

Jeśli używasz systemu Windows i wolisz skorzystać z Visual Studio, to jego darmową wersję (która jest określana jako Visual Studio Community) można pobrać ze strony *https://visualstudio.microsoft.com/*. W tym przypadku wraz z IDE zostanie zainstalowany również pakiet .NET SDK, o ile podczas instalacji wybierzesz przynajmniej jedno **obciążenie** (ang. *workload*) związane z .NET.

Każdy projekt aplikacji pisanej w C#, może z wyjątkiem tych najbardziej trywialnych, będzie zawierał wiele plików źródłowych, a wszystkie te pliki będą należały do *projektu*. Każdy projekt generuje pojedynczy wynik, nazywany **celem** (ang. *target*). W najprostszym przypadku tym celem może być pojedynczy plik, plik wykonywalny bądź biblioteka. Jednak projekty C# mogą także generować znacznie bardziej złożone wyniki. Na przykład niektóre projekty tworzą witryny WWW. Taka witryna będzie się zazwyczaj składać z wielu plików, jednak łącznie reprezentują one jedną całość — konkretną witrynę. Wyniki projektu będą zazwyczaj wdrażane jako jedna jednostka, nawet jeśli składa się na nie wiele plików.

W systemie Windows pliki wykonywalne mają zazwyczaj rozszerzenie *.exe*, natomiast biblioteki używają rozszerzenia *.dll* (historycznie jest to skrót od angielskich słów *dynamic link library* — biblioteka dołączana dynamicznie). .NET SDK może także generować ładujący program wykonywalny (w systemie Windows będzie on miał rozszerzenie *.exe*), którego działanie sprowadza się jednak do uruchomienia środowiska wykonawczego i wczytania biblioteki *.dll* zawierającej główny skompilowany kod aplikacji. (W przypadku generowania programów dla .NET Framework wygląda to nieco inaczej: kod aplikacji kompilowany jest bezpośrednio do samodzielnie uruchamianego pliku *.exe*, bez wykorzystania dodatkowego pliku *.dll*). W każdym z przypadków podstawowa różnica pomiędzy głównym skompilowanym kodem aplikacji oraz biblioteką polega na tym, że ten pierwszy określa punkt wejścia aplikacji. Oba te typy plików mogą eksportować funkcjonalności, które następnie będą wykorzystywane przez inne komponenty. Oba są także przykładami **podzespołów** (ang. *assembly*), które zostały dokładniej opisane w rozdziale 12.

Pliki projektów C# mają rozszerzenie *.csproj*, a jeśli przejrzymy te pliki przy użyciu edytora tekstów, przekonamy się, że zazwyczaj zawierają kod XML. Pliki *.csproj* opisują zawartość projektu oraz konfigurują sposób, w jaki jest on budowany. Pliki te są rozpoznawane zarówno przez Visual Studio, jak i przez rozszerzenia .NET dla VS Code. Mogą być także przetwarzane przy użyciu różnych programów narzędziowych wykonywanych z poziomu wiersza poleceń, takich jak *dotnet* wchodzący w skład .NET SDK oraz starszy program o nazwie *msbuild*. (Program msbuild obsługuje różne języki programowania i cele, a nie jedynie .NET. W rzeczywistości polecenie `dotnet build` używane do budowania projektów C# jest uproszczonym sposobem wywołania programu *msbuild*).

Bardzo często będziemy chcieli pracować nad całymi grupami projektów. Na przykład dobra praktyka programistyczna nakazuje tworzenie testów jednostkowych dla pisanego kodu, jednak przeważająca część tych testów nie musi być udostępniana jako element aplikacji; z tego względu zautomatyzowane testy są zazwyczaj tworzone w ramach odrębnych projektów. Mogą się także pojawić inne powody, które skłonią nas do rozdzielenia kodu aplikacji. Być może tworzony system składa się z klasycznej aplikacji oraz witryny WWW, jednak istnieją pewne komponenty używane w obu tych aplikacjach. W takim przypadku będziemy potrzebowali jednego projektu do utworzenia biblioteki zawierającej wspólny kod, kolejnego projektu do utworzenia pliku wykonywalnego aplikacji, kolejnego, w ramach którego będzie tworzona witryna, oraz trzech dodatkowych, zawierających testy dla trzech projektów głównych.

Narzędzia do budowania oraz IDE umożliwiające tworzenie aplikacji na platformę .ENT ułatwiają nam pracę nad powiązanymi ze sobą projektami za pomocą tak zwanych **rozwiązań** (ang. *solution*). Rozwiązanie, czyli plik z rozszerzeniem *.sln*, jest po prostu kolekcją projektów; i choć zazwyczaj projekty te są ze sobą powiązane, to wcale nie muszą być.

W przypadku korzystania z Visual Studio trzeba pamiętać, że środowisko to wymaga, by każdy projekt należał do jakiegoś rozwiązania, nawet gdyby miało to oznaczać, że w jednym rozwiązaniu znajdzie się tylko jeden projekt. Visual Studio Code pozwala otwierać pojedyncze projekty, choć jego rozszerzenia dla .NET rozpoznają i obsługują rozwiązania.

Projekt może należeć do więcej niż jednego rozwiązania. W przypadku dużej bazy kodu stosunkowo często zdarza się korzystać z kilku rozwiązań zawierających różne kombinacje projektów. W takich przypadkach zazwyczaj istnieje jakieś główne rozwiązanie zawierające wszystkie projekty, jednak nie wszyscy programiści będą chcieli dysponować ciągłym dostępem do całego kodu. Kontynuując nasz hipotetyczny przykład, można założyć, że osoby pracujące nad klasyczną aplikacją dla komputerów stacjonarnych także chcą mieć dostęp do wspólnych bibliotek, jednak najprawdopodobniej nie interesuje ich projekt witryny WWW.

W dalszej części rozdziału, w ramach wstępu do prezentacji możliwości języka, przedstawię, jak utworzyć nowy projekt, otworzyć go w Visual Studio Code, a potem uruchomić. Następnie opiszę różne możliwości projektów C#. Pokażę także, w jaki sposób można dodać nowy projekt z testami jednostkowymi oraz jak utworzyć rozwiązanie.

Anatomia prostego programu

Po zainstalowaniu .NET SDK, bezpośrednio bądź wraz z jakimś IDE, możemy przystąpić do tworzenia nowego programu dla platformy .NET. Zaczniemy od utworzenia na dysku twardym nowego katalogu, *WitajSwiecie*, przeznaczonego do przechowywania kodów naszego programu. Następnie należy otworzyć okno wiersza poleceń, przejść do utworzonego wcześniej katalogu i wykonać polecenie:

```
dotnet new console
```

Wykonanie tego polecenia spowoduje utworzenie nowej aplikacji konsolowej w języku C#, która początkowo będzie się składać z dwóch plików. Pierwszym z nich jest plik projektu, którego nazwa jest określana na podstawie nazwy bieżącego katalogu; w naszym przypadku będzie to *WitajSwiecie.csproj*. Drugim z utworzonych plików jest *Program.cs*, zawierający kod źródłowy aplikacji. Kiedy otworzymy ten plik w edytorze, okaże się, że jego zawartość jest bardzo prosta, jak pokazałem na listingu 1.1.

Listing 1.1. Nasz pierwszy program

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Ten program można skompilować i uruchomić przy użyciu następującego polecenia:

```
dotnet run
```

Jak łatwo się domyślić, program wyświetla w oknie wiersza poleceń napis „Hello World!”.

Jeśli już masz doświadczenie w pisaniu programów w języku C#, a po niniejszą książkę sięgnąłeś jedynie po to, by dowiedzieć się, jakie nowości wprowadzono w C# 10.0, to ten przykład może być pewną niespodzianką. We wcześniejszych wersjach języka kod klasycznego programu typu „Witaj, świecie!”, który obowiązkowo rozpoczyna wszystkie książki poświęcone językom programowania, był znacząco dłuższy. Ten kod wygląda tak odmiennie, że twórcy .NET SDK uznali za konieczne opatrzyć go stosownymi wyjaśnieniami — ponad połowę kodu tego przykładu stanowi komentarz zawierający odnośnik do strony WWW z wyjaśnieniami, gdzie podziała się reszta kodu. My potrzebujemy jednak tylko tego drugiego, krótszego wiersza.

Pokazuje on jedną ze zmian wprowadzonych w języku C# 10.0: ma ona na celu umożliwienie programistom przejścia do razu do tego, co najważniejsze, poprzez zredukowanie ilości powtarzającego się, **szablonowego kodu** (ang. *boilerplate code*). Określenie to odnosi się do kodu, który musi zostać użyty w celu spełnienia pewnych reguł lub konwencji, lecz w każdym projekcie wygląda mniej więcej tak samo. Na przykład język C# wymaga, by kod był definiowany wewnątrz **metod**, które z kolei muszą być definiowane wewnątrz **typów**. Przykłady tych reguł możemy zobaczyć na listingu 1.1. W celu wygenerowania wyników nasz przykładowy program korzysta z możliwości wyświetlania tekstów na ekranie zapewnianych przez środowisko uruchomieniowe .NET, które w kodzie są reprezentowane przez metodę `WriteLine`. Jednak w kodzie nie użyliśmy jedynie nazwy tej metody, `WriteLine`, gdyż w C# metody zawsze należą do typów — to właśnie dlatego w kodzie programu pojawił się fragment `Console.WriteLine`.

Oczywiście regułom tym podlega każdy kod pisany w języku C#, dlatego przykładowy kod wywołujący metodę `Console.WriteLine` musi istnieć wewnątrz jakiejś metody jakiegoś typu. W kodzie pisanym

w C# typ i metoda przeważnie są zapisane jawnie: w większości przypadków możemy zobaczyć kod podobny do przedstawionego na listingu 1.2.

Listing 1.2. Przykład „Witaj, świecie!" z widocznym szablonowym, powtarzającym się kodem

```
using System;

internal class Program
{
    private static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

Także w tym przypadku za określenie działania aplikacji odpowiada tylko jeden wiersz kodu i jest on dokładnie taki sam jak na listingu 1.1. Oczywistą zaletą pierwszego z przykładów jest to, że możemy w nim od razu skoncentrować się na tym, co robi program, natomiast jego wadą jest ukrycie znacznej części tego, co się w nim dzieje. Jawny zapis, przedstawiony na listingu 1.2, niczego nie ukrywa. W przypadku kodu z listingu 1.1 kompilator wciąż umieszcza kod wewnątrz metody zdefiniowanej w typie `Program`, jednak dla nas ten kod jest niewidoczny. W kodzie z listingu 1.2 zarówno metoda, jak i typ są wyraźnie widoczne.

W praktyce większość kodu pisanego w języku C# przypomina kod z listingu 1.2, a nie kod z listingu 1.1. Wynika to z faktu, że większość możliwości eliminowania powtarzającego się kodu wprowadzonych w C# 10.0 dotyczy wyłącznie punktu wejścia do programu. Zawsze gdy piszemy kod, który ma być wykonywany podczas uruchamiania programu, możemy pominąć metodę i klasę, w których będzie on umieszczony. Jednak każdy program ma tylko jeden punkt wejścia, dlatego cały pozostały kod wciąż musi być jawnie umieszczany w metodach i klasach.

W praktyce projekty składają się z wielu plików, a zazwyczaj także z wielu projektów, dlatego teraz przedstawię nieco bardziej realistyczny przykład, a mianowicie napiszemy program, który wylicza średnią liczb (oczywiście mam tu na myśli średnią arytmetyczną). Oprócz tego przygotujemy od razu drugi projekt, który testuje działanie pierwszego. Będziemy zatem mieli do czynienia z dwoma projektami, a to oznacza, że tym razem będziemy potrzebować rozwiązania. Zaczniemy od utworzenia katalogu o nazwie *Averages*. Jeśli czytając tekst, od razu wykonujesz opisywane w nim czynności, zwróć uwagę, że nie ma znaczenia, gdzie utworzysz ten katalog, choć umieszczanie go w katalogu pierwszego projektu jest *naprawdę kiepskim* pomysłem — *nie* należy tak robić. A zatem otwórz okno wiersza poleceń, przejdź w nim do utworzonego wcześniej katalogu *Averages*, a potem wykonaj następujące polecenie:

```
dotnet new sln
```

Jego wykonanie spowoduje utworzenie nowego pliku rozwiązania o nazwie *Averages.sln*. (Polecenie `dotnet new` zazwyczaj nadaje nowym rozwiązaniom i projektom nazwy odpowiadające nazwie bieżącego katalogu, w którym zostało wykonane, choć nazwy te można także podawać samodzielnie). Teraz do tego rozwiązania dodamy dwa projekty. Możemy to zrobić, wykonując dwa poniższe polecenia:

```
dotnet new console -o Averages
dotnet new mstest -o Averages.Tests
```

Zastosowana w tych poleceniach opcja `-o` (skrót od `output`) oznacza, że chcemy, by każdy z projektów został utworzony w nowym podkatalogu — w przypadku korzystania z wielu projektów każdy z nich musi być umieszczony we własnym, odrębnym katalogu.

Teraz musimy dodać te projekty do rozwiązania:

```
dotnet sln add ./Averages/Averages.csproj
dotnet sln add ./Averages.Tests/Averages.Tests.csproj
```

Ten drugi projekt posłuży nam do zdefiniowania testów sprawdzających działanie kodu w pierwszym projekcie (to właśnie z tego powodu podczas jego tworzenia jawnie użyliśmy typu `mstest`, zaznaczając w ten sposób, że chodzi nam o projekt korzystający z platformy do tworzenia i obsługi testów jednostkowych). Aby te testy mogły działać, drugi projekt potrzebuje dostępu do kodu umieszczonego w pierwszym projekcie. Możemy go zapewnić, wykonując następujące polecenie:

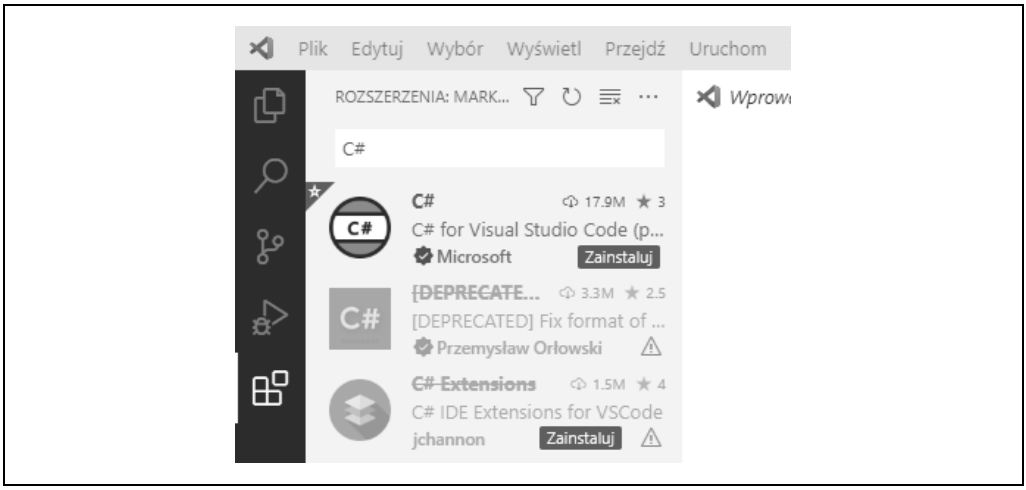
```
dotnet add ./Averages.Tests/Averages.Tests.csproj reference ./Averages/Averages.csproj
```

I w końcu, aby przejść do edycji kodu, możemy uruchomić w bieżącym katalogu program VS Code, używając w tym celu poniższego polecenia:

```
code .
```

Jeśli podczas lektury tekstu wykonujesz opisywane w nim czynności i jeśli jest to pierwszy raz, kiedy korzystasz z programu VS Code, to zostaniesz poproszony o podjęcie kilku decyzji, takich jak wybranie schematu kolorów. Być może będzie Cię kusić, by zignorować to pytanie, lecz jedną z możliwości, jakie będą dostępne w tym momencie, będzie zainstalowanie rozszerzeń związanych z obsługą języka. VS Code jest używany do pisania kodu w bardzo wielu językach, a jego program instalacyjny nie stara się domyślić, którego z nich Ty zechcesz używać. Oznacza to, że aby korzystać z języka C#, musisz zainstalować odpowiednie rozszerzenie. Jeśli jednak będziesz postępować zgodnie z instrukcjami wyświetlanymi przez VS Code, to program sam zainstaluje przygotowane przez Microsoft rozszerzenie do obsługi języka C#. Jeżeli VS Code nie zaproponuje Ci zainstalowania tego rozszerzenia, nie panikuj, bo być może już jest zainstalowane, więc zadawanie tych wstępnych pytań nie jest konieczne, albo od czasu kiedy przygotowywałem tę książkę, zmienił się sposób działania VS Code podczas jego pierwszego uruchamiania. Na szczęście rozszerzenia można wyszukiwać i instalować bardzo łatwo. Wystarczy kliknąć ikonę *Rozszerzenia*[5] (*Extensions*), umieszczoną na pasku z lewej strony okna VS Code, a zostanie wyświetlony panel z listą wszystkich rozszerzeń, które, według programu, mogą się nam przydać w bieżącym kontekście. Jeśli uruchomimy program w katalogu zawierającym plik *.csproj*, to na tej liście powinno być widoczne rozszerzenie do obsługi C#. Jeśli jednak tak się nie stanie, zawsze możesz samodzielnie poszukać potrzebnego rozszerzenia. Na rysunku 1.1 pokazałem panel rozszerzeń VS Code — można go wyświetlić, klikając ikonę na pasku z lewej strony okna programu (ostatnią z ikon widocznych na rysunku).

[5] Podczas pierwszego uruchamiania VS Code program automatycznie proponuje zainstalowanie rozszerzenia polonizującego interfejs użytkownika, dlatego w dalszej części tekstu będę się posługiwał polskimi nazwami opcji, podając ich angielskie odpowiedniki w nawiasach — *przyp. tłum.*



Rysunek 1.1. Panel rozszerzeń VS Code z widocznym rozszerzeniem do obsługi języka C#

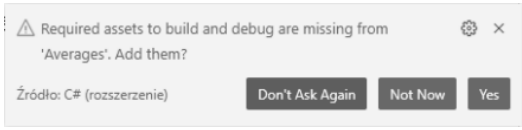
Jak widać na rysunku, w polu wyszukiwania na górze panelu wpisałem *C#*, a pierwszym z wyświetlonych wyników jest rozszerzenie do obsługi C# przygotowane przez Microsoft. Oprócz niego na liście wyników pojawiło się kilka innych rozszerzeń. Jeśli czytasz książkę i od razu wykonujesz opisywane czynności, to upewnij się, że wybierasz odpowiednie rozszerzenie. Kliknięcie jednego z rozszerzeń wyświetlonych na liście spowoduje wyświetlenie szczegółowych informacji na jego temat. W naszym przypadku pełna nazwa rozszerzenia powinna brzmieć „C# for Visual Studio Code (powered by OmniSharp)", a jego wydawcą (ang. *publisher*) powinna być firma Microsoft. Aby zainstalować rozszerzenie, wystarczy kliknąć przycisk *Install*.

Pobranie i zainstalowanie rozszerzenia dla języka C# może zająć kilka minut, a kiedy zostanie zakończone, to lewy dolny róg okna VS Code powinien wyglądać tak jak na rysunku 1.2 — powinny być na nim widoczne nazwa pliku rozwiązania oraz ikona płomienia, oznaczająca, że OmniSharp, system obsługujący język C# w VS Code, jest gotowy. Może się także zdarzyć, że w górnej części okna VS Code zostanie wyświetlone okienko do wyboru projektu — rozszerzenie C# przeskanuje bowiem katalog rozwiązania i znajdzie w nim nasze rozwiązanie oraz dwa projekty wchodzące w jego skład. Normalnie VS Code otworzy jedynie plik rozwiązania, ale w zależności od konfiguracji systemu może również poprosić o wybranie jednego z dostępnych projektów. My mamy zamiar procować z oboma projektami, więc wybierz opcję *Averages.sln*.

Rysunek 1.2. Pasek statusu Visual Studio Code

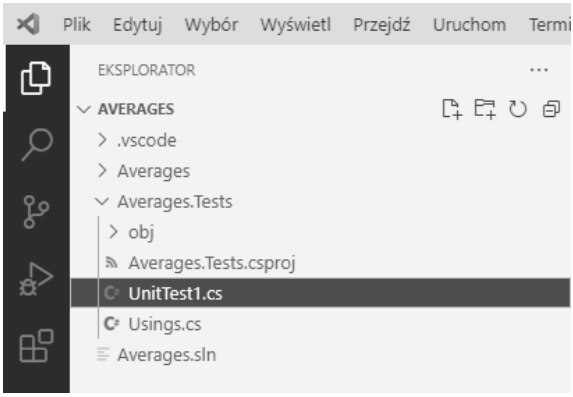
Kolejną czynnością, jaką wykona VS Code, jest przeskanowanie wszystkich kodów źródłowych we wszystkich projektach należących do rozwiązania. Oczywiście w naszym przypadku nie ma ich jeszcze zbyt wiele, jednak VS Code będzie analizować kod podczas jego wpisywania, co pozwoli mu wykrywać problemy i wyświetlać przydatne sugestie. W trakcie tego procesu program wykryje, że nie dysponujemy jeszcze plikiem konfiguracyjnym z informacjami o sposobie budowania

i debugowania projektów. Wyświetli także (w prawym dolnym rogu okna) okienko dialogowe umożliwiające jego dodanie (zobacz rysunek 1.3). Warto kliknąć w nim przycisk *Yes*, a następnie wybrać opcję *Averages.csproj*, aby wskazać program główny, dzięki czemu VS Code będzie wiedzieć, którego kodu ma użyć, gdy poprosimy o jego uruchomienie lub rozpoczęcie debugowania.



Rysunek 1.3. Wyświetlane przez rozszerzenie C# pytanie o dodanie zasobów do budowania i debugowania

Pliki wchodzące w skład rozszerzenia i projektów są prezentowane w panelu *Eksplorator* (*Explorer*), który można wyświetlić, klikając ikonę umieszczoną na samej górze paska z lewej strony okna VS Code. Jak widać na rysunku 1.4 panel ten prezentuje katalogi i pliki. Na rysunku rozwinąłem zawartość katalogu *Averages.Tests* i zaznaczyłem plik *UnitTest1.cs*.



Rysunek 1.4. Panel Eksplorator Visual Studio Code

Pojedyncze kliknięcie pliku w panelu *Explorer* powoduje, że VS Code wyświetla plik na **karcie podglądu** (ang. *preview tab*), co oznacza, że nie będzie on prezentowany zbyt długo: gdy tylko klikniemy jakiś inny plik, umieszczony w nim kod zastąpi dotychczasową zawartość karty. Rozwiązanie to zostało zaprojektowane po to, by uniknąć otwierania setek kart, jednak gdy często przełączamy się pomiędzy dwoma plikami, może ono być denerwujące. Można go jednak uniknąć: wystarczy dwukrotnie kliknąć plik, który chcemy otworzyć — powoduje to otworzenie zawartości pliku w normalnej karcie, która pozostanie otwarta aż do momentu, kiedy ją celowo zamkniemy. Ewentualnie, jeśli interesujący nas plik już jest otwarty w karcie podglądu, wystarczy dwukrotnie kliknąć gdziekolwiek w obszarze tej karty, by przekształcić ją w normalną. VS Code wyświetla nazwy plików na karcie podglądu kursywą; łatwo zauważyć, że po dwukrotnym kliknięciu czcionka w nazwie karty zostaje zmieniona na normalną.

Być może się zastanawiasz, dlaczego na rysunku 4.1 wyświetliłem zawartość katalogu *Averages.Tests*. Otóż zrobiłem tak dlatego, że ten projekt testowy pozwoli nam upewnić się, iż główny projekt, *Averages*, będzie działał zgodnie z oczekiwaniami. Tak się składa, że jestem zwolennikiem stylu programowania, w którym najpierw pisane są testy, a dopiero potem kod, który testują. Właśnie dlatego zaczniemy od projektu testowego.

Pisanie testu jednostkowego

Nieco wcześniej, w poleceniu tworzącym drugi projekt, jawnie określiłem jego typ: `mstest`. Użycie tego szablonu projektu spowodowało wygenerowanie klasy testowej umieszczonej w pliku *UnitTest1.cs*. My jednak będziemy chcieli wybrać bardziej opisową nazwę. Istnieje wiele różnych szkół określania struktury tworzonych testów jednostkowych. Niektórzy programiści opowiadają się za tworzeniem jednej klasy testowej dla każdej klasy, którą chcemy testować, jednak ja preferuję rozwiązanie polegające na tworzeniu odrębnej klasy testowej dla każdego **scenariusza**, w jakim klasa ma być testowana, oraz odrębnych metod dla wszystkich warunków, które w danym scenariuszu powinny być spełnione przez nasz kod. Nasz program główny będzie działał tylko w jeden sposób: będzie obliczał średnią arytmetyczną liczb wpisanych jako dane wejściowe. Dlatego zmienimy nazwę pliku testu jednostkowego z *UnitTest1.cs* na *WhenCalculatingAverages.cs*. (Nazwę pliku można zmienić poprzez kliknięcie go prawym przyciskiem myszy w panelu *Explorer* VS Code i wybranie z menu podręcznego opcji *Rename*). Ten test pozwoli nam upewnić się, że wyniki obliczone dla kilku określonych liczb będą prawidłowe. Kompletny kod testu jednostkowego przedstawiłem na listingu 1.3; zawiera on dwa testy, których kod został wyróżniony pogrubieniem.

Listing 1.3. Klasa testu jednostkowego dla naszego pierwszego programu

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Averages.Tests;

[TestClass]
public class WhenCalculatingAverages
{
    [TestMethod]
    public void SingleInputShouldProduceSameValueAsResult()
    {
        string[] inputs = { "1" };
        double result = AverageCalculator.ArithmeticMean(inputs);
        Assert.AreEqual(1.0, result, 1E-14);
    }

    [TestMethod]
    public void MultipleInputsShouldProduceAverageAsResult()
    {
        string[] inputs = { "1", "2", "3" };
        double result = AverageCalculator.ArithmeticMean(inputs);
        Assert.AreEqual(2.0, result, 1E-14);
    }
}
```

Każdy z fragmentów powyższego kodu zostanie opisany nieco później, po przedstawieniu samego programu. Na razie najbardziej interesującym aspektem tego przykładu są dwie metody. Pierwszą z nich jest `SingleInputShouldProduceSameValueAsResult`, która upewnia się, że program prawidłowo oblicza średnią w przypadku, gdy zostanie wpisana tylko jedna liczba. Pierwszy wiersz kodu w metodzie opisuje dane wejściowe — w tym przypadku jest nimi pojedyncza liczba. (Nieco dziwnie może wyglądać to, że w teście liczby są reprezentowane w formie łańcuchów znaków. Dzieje się tak, gdyż do programu głównego dane wejściowe będą przekazywane w postaci argumentów wiersza poleceń, a test musi to prawidłowo odzwierciedlać). Drugi wiersz metody wywołuje testowany kod (którego jeszcze nie napisaliśmy). Trzeci wiersz stwierdza, że wyliczona średnia musi być równa jedynej danej wejściowej. Jeśli średnia będzie inna, test zgłosi błąd. Druga z metod klasy testowej, `MultipleInputsShouldProduceAverageAsResult`, testuje nieco bardziej złożony przypadek, w którym dane wejściowe zawierają trzy liczby, jednak jego ogólna postać jest taka sama jak w przypadku pierwszego testu.

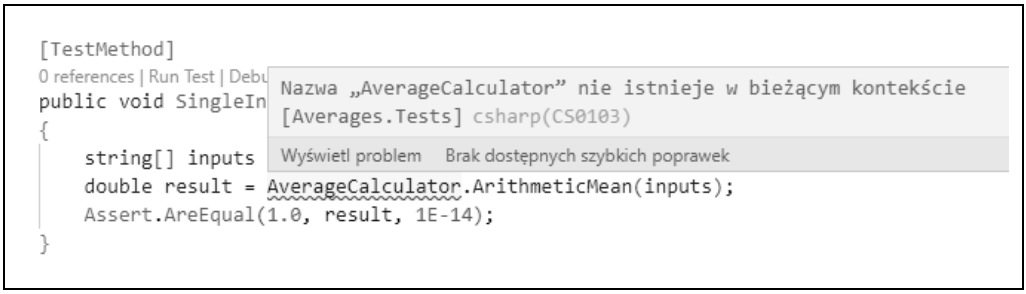
W tym przypadku korzystamy z typu `double` języka C#, czyli liczb zmiennoprzecinkowych o podwójnej precyzji, co pozwoli nam obsługiwać wyniki, które nie są liczbami całkowitymi. Wbudowane typy danych języka C# opisałem dokładniej w następnym rozdziale, musisz jednak pamiętać, że podobnie jak w większości innych języków programowania, także w C# arytmetyka zmiennoprzecinkowa ma ograniczoną dokładność. Metoda `Assert.AreEqual`, której używamy do sprawdzania wyników, uwzględnia te ograniczenia i pozwala określić maksymalną tolerancję dla niedokładności porównania. Ostatni argument obu jej wywołań, `1E-14`, oznacza liczbę 1 podzieloną przez 10 do potęgi 14. Oznacza to, że testy wymagają, by odpowiedź była poprawna z dokładnością do 14 miejsc po przecinku dziesiętnym.

Skupmy się na jednym, konkretnym wierszu obu tych testów, a mianowicie wierszu wywołującym kod, który chcemy testować. Wiersz ten przedstawiłem na listingu 1.4. Pokazuje on, w jaki sposób w języku C# można wywoływać metodę, która zwraca wynik. Rozpoczyna się on od zadeklarowania zmiennej, która będzie przechowywać wynik. (Słowo `double` określa typ danych, a `result` jest nazwą zmiennej). W C# wszystkie metody muszą być definiowane wewnątrz jakiegoś typu, a zatem używamy tu zapisu podobnego jak w przedstawionym wcześniej przykładzie wywołującym metodę `Console.WriteLine`: nazwa typu, kropka i nazwa metody. Za nazwą metody, w nawiasach, podawane są argumenty wywołania.

Listing 1.4. Postać wywołania metody

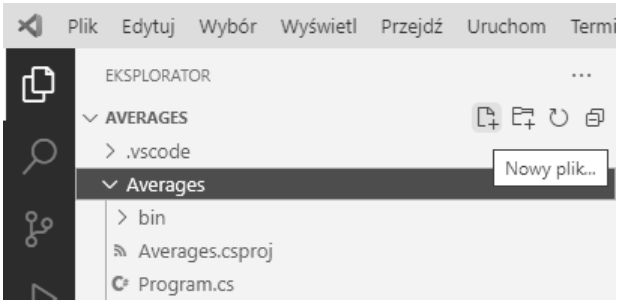
```
double result = AverageCalculator.ArithmeticMean(inputs);
```

Jeśli wpisujesz prezentowany tu kod podczas lektury książki, to po pierwsze należy Ci się szacunek! A po drugie: jeżeli przyjrzysz się dwóm miejscom, w których występuje przedstawiony wiersz kodu (a występuje w obu metodach), to zapewne zauważysz, że VS Code każde wystąpienie nazwy `AverageCalculator` podkreśla cienką, falistą linią. Jeśli umieścimy wskaźnik myszy na takim wyróżnionym tekście, VS Code wyświetli komunikat błędu, taki jak na rysunku 1.5.



Rysunek 1.5. Oznaczenie nierozpoznanego typu

Ten błąd informuje nas o tym, o czym już wiemy: kod, którego działanie test ma sprawdzać, nie został jeszcze napisany. Spróbujmy zatem rozwiązać ten problem. W tym celu musimy dodać nowy plik. Możemy to zrobić w panelu *Explorer* VS Code, klikając skrajny lewy przycisk wyświetlony z prawej strony paska z tytułem panelu. Jak pokazałem na rysunku 1.6, umieszczenie wskaźnika myszy na tym przycisku powoduje wyświetlenie etykiety ekranowej potwierdzającej przeznaczenie przycisku. Kliknij przycisk, a następnie wpisz nazwę tworzonego pliku: *AverageCalculator.cs*.



Rysunek 1.6. Dodawanie nowego pliku

W efekcie VS Code utworzy nowy, pusty plik. Dodamy doń najkrótszy możliwy kod niezbędny do rozwiązania problemu zasygnalizowanego przez błąd pokazany na rysunku 1.5. Kod przedstawiony na listingu 1.5 spełni wymagania kompilatora C#. Nie będzie on jednak kompletny, nie będzie bowiem wykonywał niezbędnych obliczeń (tym zajmiemy się nieco później).

Listing 1.5. Prosta klasa

```
namespace Averages;

public static class AverageCalculator
{
    public static double ArithmeticMean(string[] args)
    {
        return 1.0;
    }
}
```

Teraz kod można już skompilować, a to oznacza, że możemy także przeprowadzić testy. W tym celu wykonaj poniższe polecenie:

```
dotnet test
```

Jego wykonanie spowoduje wyświetlenie następujących wyników:

```
    Niepowodzenie MultipleInputsShouldProduceAverageAsResult [26 ms]
    Komunikat o błędzie:
     Assert.AreEqual failed. Expected a difference no greater than <1E-14> between expected
     ↪value <2> and actual value <1>.
    Ślad stosu:
       at Averages.Tests.UnitTest1.MultipleInputsShouldProduceAverageAsResult() in H:\
       ↪Programming C# 10.0\RO1\Averages\Averages.TeTests\WhenCalculatingAverages.cs:line 19


Niepowodzenie! – niepowodzenie:     1, powodzenie:     1, pominięto:     0, łącznie:     2,
czas trwania: 95 ms - Averages.Teststs.dll (net6.0)
```

Zgodnie z oczekiwaniami testy nie zostaną wykonane poprawnie, gdyż nie napisaliśmy jeszcze kodu implementującego obliczanie średniej. Jednak zanim to zrobimy, chciałem wyjaśnić wszystkie elementy kodu z listingu 1.5, gdyż stanowi on doskonały przykład prezentujący kilka ważnych elementów składni oraz struktury kodu C#. Pierwszym elementem umieszczonym w tym pliku jest **deklaracja przestrzeni nazw**.

Przestrzenie nazw

Przestrzenie nazw (ang. *namespace*) wnoszą porządek i strukturę do świata, który bez nich byłby jednym wielkim chaosem. Biblioteka uruchomieniowa .NET zawiera bardzo wiele typów, a wiele innych istniejących bibliotek udostępnia kolejne, nie wspominając w ogóle o tych, które samodzielnie napiszesz. W przypadku tak wielkiej liczby elementów posiadających własne nazwy pojawiają się dwa podstawowe problemy. Przede wszystkim bardzo trudno jest zagwarantować niepowtarzalność nazw. Poza tym sporych problemów może przysporzyć odszukanie potrzebnego API — jeśli nie znamy lub nie jesteśmy w stanie zgadnąć odpowiedniej nazwy, to znalezienie jej na liście liczącej tysiące pozycji może być bardzo trudne. Przestrzenie nazw rozwiązują oba te problemy.

Większość typów platformy .NET została zdefiniowana w przestrzeniach nazw. Istnieją pewne konwencje związane z ich tworzeniem i stosowaniem, z którymi będziesz się bardzo często stykać. Na przykład typy umieszczone w bibliotekach uruchomieniowych .NET należą do przestrzeni nazw, których nazwy zaczynają się od `System`. Co więcej, Microsoft stworzył bardzo wiele przydatnych bibliotek, które nie wchodzą w skład podstawowej części platformy .NET; ich nazwy zazwyczaj zaczynają się od `Microsoft`, bądź też, jeśli są przeznaczone do użycia z jakąś konkretną technologią, ich nazwy mogą odpowiadać nazwie tej technologii. (Na przykład biblioteki pozwalające korzystać z chmurowej platformy Microsoftu, Azure, definiują typy umieszczone w przestrzeniach nazw, których nazwy rozpoczynają się od `Azure`). Biblioteki stworzone i dostarczane przez inne firmy także zazwyczaj mają nazwy rozpoczynające się od nazwy firmy lub produktu, natomiast nazwy bibliotek tworzonych w ramach otwartych projektów rozpoczynają się przeważnie od nazwy projektu. Nie ma żadnego nakazu, który by nas zmuszał do umieszczania naszych własnych typów w przestrzeniach nazw, jednak zaleca się, by właśnie tak postępować. C# nie traktuje przestrzeni

nazw `System` w żaden specjalny sposób, zatem nic nie stoi na przeszkodzie, byśmy używali jej w swoich własnych typach, choć jeśli nie piszemy kodu rozbudowującego bibliotekę uruchomieniową .NET, który zostanie opublikowany jako **prośba wprowadzenia zmian** (ang. *pull request*) do repozytorium środowiska uruchomieniowego .NET (*https://github.com/dotnet/runtime*), to nie będzie to dobry pomysł, gdyż może zmylić innych programistów. Na potrzeby definiowania własnego kodu należy raczej wybrać coś bardziej unikatowego, na przykład nazwę firmy lub produktu. Jak widać w pierwszym wierszu listingu 1.5, klasę `AverageCalculator` zdefiniowaliśmy w przestrzeni nazw `Averages`, co odpowiada nazwie naszego projektu.

Styl deklarowania przestrzeni nazw przedstawiony na listingu 1.5 jest nowością wprowadzoną w C# 10.0. Obecnie w przeważającej większości kodu, z jakim będziesz się stykać, stosowany jest starszy, nieco dłuższy zapis, przedstawiony na listingu 1.6. Różnica polega na tym, że w starszym sposobie deklarowania przestrzeni nazw za nazwą jest umieszczana para nawiasów klamrowych ({}), a efekt deklaracji odnosi się do jej zawartości. Takie rozwiązanie pozwala umieszczać kilka deklaracji przestrzeni nazw w jednym pliku. Jednak w praktyce przeważająca większość plików z kodem C# będzie zawierać deklarację tylko jednej przestrzeni nazw. A to oznacza, że w przypadku stosowania tego starszego stylu deklarowania przestrzeni nazw przeważająca większość kodu zapisanego w każdym z plików musi być umieszczona w nawiasach klamrowych i odpowiednio wcięta. W nowym stylu deklarowania, przedstawionym na listingu 1.5, przestrzeń nazw będzie zawierać wszystkie typy zadeklarowane w pliku, i to bez konieczności jawnego umieszczania ich w nawiasach klamrowych. Ten nowy styl deklarowania przestrzeni nazw jest jednym z nowych rozwiązań wprowadzonych w C# 10.0, mających na celu ograniczenie nieproduktywnego bałaganu w kodzie źródłowym.

Listing 1.6. Deklaracja przestrzeni nazw przed wprowadzeniem C# 10.0

```
namespace Averages
{
    public static class AverageCalculator
    {
        ...reszta jak wcześniej...
    }
}
```

Przestrzenie nazw zazwyczaj stanowią także pewną podpowiedź dotyczącą przeznaczenia typu. Na przykład wszystkie typy związane z obsługą plików dostępne w bibliotece uruchomieniowej należą do przestrzeni nazw `System.IO`, natomiast typy związane z komunikacją sieciową — do przestrzeni `System.Net`. Przestrzenie nazw mogą także tworzyć hierarchie. Dlatego przestrzeń nazw `System` zawiera typy, jak również kilka innych przestrzeni nazw, na przykład `System.Net`, która z kolei zawiera dalsze przestrzenie, takie jak `System.Net.Sockets` oraz `System.Net.Mail`. Przykłady te pokazują, że przestrzenie nazw pełnią także rolę opisów pomagających w korzystaniu z zawartości biblioteki. Poszukując narzędzi do obsługi wyrażeń regularnych, możemy przejrzeć listę dostępnych przestrzeni nazw i zwrócić uwagę, że jest wśród nich dostępna przestrzeń `System.Text`. Przeglądając jej zawartość, znajdziemy kolejną przestrzeń nazw — `System.Text.Regular` ↪`Expressions` — i w tym momencie uzyskamy już pewność, że trafiliśmy we właściwe miejsce.

Przestrzenie nazw pozwalają także zapewniać niepowtarzalność. Przestrzeń, do jakiej należy dany typ, jest elementem jego pełnej nazwy. Dzięki temu biblioteki mogą nadawać swoim typom krótkie

i proste nazwy. Na przykład API do obsługi wyrażeń regularnych zawiera klasę `Capture`, reprezentującą wyniki zwrócone podczas próby dopasowania wyrażenia. Jeśli jednak pracujemy nad oprogramowaniem związanym z przetwarzaniem obrazów, to ten sam angielski termin **capture** będzie najczęściej używany w kontekście pobierania pewnych danych obrazu, możemy zatem uznać, że nazwa `Capture` będzie najbardziej odpowiednia dla którejś z naszych klas. Byłoby bardzo denerwujące, gdybyśmy musieli wymyślać jakieś inne nazwy tylko dlatego, że te, które nam najbardziej odpowiadają, zostały już wykorzystane; zwłaszcza skoro nasz kod do przetwarzania obrazów w ogóle nie korzysta z wyrażeń regularnych, a co za tym idzie — nie mamy zamiaru korzystać z istniejącego typu `Capture`.

Ale w rzeczywistości nie ma większego problemu. Oba typy mogą nazywać się `Capture`, a jednocześnie ich nazwy mogą być różne. Pełną nazwą klasy `Capture` związanej z obsługą wyrażeń regularnych jest w rzeczywistości `System.Text.RegularExpressions.Capture`; analogicznie pełna nazwa naszej klasy także będzie zawierać określenie przestrzeni nazw (na przykład `SpiffingSoftworks.Imaging.Capture`).

Jeśli naprawdę będziemy tego chcieli, to podczas każdego użycia typu możemy podawać jego pełną nazwę, jednak większość programistów nie chce robić czegoś tak męczącego i tu właśnie przydają się dyrektywy `using` umieszczone na początku kodu z listingów 1.2 i 1.3. Bardzo często na początku pliku źródłowego będzie umieszczona cała lista takich dyrektyw, które określają przestrzenie nazw zawierające typy używane w danym pliku. Zazwyczaj będziemy edytować tę listę, dostosowując ją do wymagań kodu w konkretnym pliku. Na przykład w naszym przypadku podczas generowania projektu testowego polecenie `dotnet` dodało dyrektywę `using Microsoft.VisualStudio.TestTools.UnitTesting`. W zależności od kontekstu spotkamy się z różnymi zestawami dyrektyw `using`. Na przykład, jeśli do projektu dodamy klasę reprezentującą kontrolkę interfejsu użytkownika, to Visual Studio umieści na liście różne przestrzenie nazw związane z obsługą interfejsu użytkownika.

Projekty korzystające z możliwości C# 10.0 przeważnie będą zawierać mniej dyrektyw `using`, niż możemy zobaczyć w kodzie pisanym w starszych wersjach języka (czyli obecnie niemal we wszystkich). Dzieje się tak ze względu na nową możliwość wprowadzoną w C# 10.0, tak zwane **globalne dyrektywy using** (ang. *global using directives*). Jeśli przed dyrektywą umieścimy słowo kluczowe `global`, jak pokazałem na listingu 1.7, to dyrektywa ta będzie się odnosić do wszystkich plików źródłowych w projekcie. .NET SDK przesuwa tę możliwość jeszcze o krok dalej: generuje w projekcie plik ukryty zawierający zestaw dyrektyw `global using`, aby zapewnić, że w projekcie będą dostępne takie przestrzenie nazw jak `System` i `System.Collections.Generic`. (Dokładny zestaw przestrzeni nazw dodawanych w ten sposób zmienia się zależnie od typu projektu, na przykład projekty aplikacji internetowych zawierają kilka dodatkowych przestrzeni. Jeśli zastanawiasz się, dlaczego rozwiązanie z listingu 1.7 nie zostało zastosowane w naszym przykładowym projekcie testowym, to wiedz, że powodem jest brak odrębnego typu projektu dla projektów zawierających testy jednostkowe — .NET SDK traktuje je jako pewien rodzaj bibliotek klas).

Listing 1.7. Dyrektywa global using

```
global using Microsoft.VisualStudio.TestTools.UnitTesting;
```

Dzięki użyciu deklaracji `using` (bądź to bezpośrednio w pliku, bądź też globalnie) będzie w nim można stosować skrócone, a nie pełne nazwy klas. Wiersz, który sprawia, że kod z listingu 1.1 realizuje swoje zadanie, korzysta z klasy `System.Console`, jednak dzięki temu, że SDK dodało

niejawną dyrektywę `global using` odwołującą się do przestrzeni nazw `System`, można w nim używać skróconej nazwy — `Console`.

Wcześniej w tym rozdziale użyliśmy programu narzędziowego `dotnet`, by dodać do projektu `Averages.Tests` odwołanie do projektu `Averages`. Można by sądzić, że określanie tych odwołań jest zbyteczne — w końcu czy kompilator nie jest w stanie określić niezbędnych zewnętrznych bibliotek na podstawie podanych w kodzie dyrektyw `using`? Mógłby, gdyby istniało bezpośrednie odwzorowanie pomiędzy przestrzeniami nazw oraz bibliotekami lub pakietami. Jednak takiego odwzorowania nie ma. Czasami jednak istnieje zauważalny związek, na przykład w popularnym pakiecie NuGet `Newtonsoft.Json` jest biblioteka *Newtonsoft.Json.dll* zawierająca klasy należące do przestrzeni nazw `Newtonsoft.Json`. Jednak często takiego związku nie ma — biblioteka uruchomieniowa .NET zawiera plik *System.Private.CoreLib.dll*, ale nie ma przestrzeni nazw `System.Private.CoreLib`. Dlatego konieczne jest przekazanie kompilatorowi informacji o tym, których bibliotek potrzebuje nasz program, jak również określenie, które przestrzenie nazw będą używane. Szczegółowe informacje dotyczące natury oraz struktury plików bibliotek zostały zamieszczone w rozdziale 12.

Jednak nawet pomimo korzystania z przestrzeni nazw mogą się pojawiać niejednoznaczności. Może się zdarzyć, że w jedynym pliku źródłowym zechcemy używać dwóch przestrzeni nazw, a każda z nich będzie definiować tę samą klasę. Chcąc używać takiej klasy, będziemy musieli zrobić to jawnie, czyli podać jej pełną nazwę. Jeśli takie klasy będą się pojawiały w pliku bardzo często, to istnieje pewien sposób pozwalający nieco skrócić ilość wpisywanego kodu: nazwę klasy będziemy musieli podać tylko raz, gdyż zdefiniujemy dla niej **nazwę zastępczą**. Kod przedstawiony na listingu 1.8 korzysta z nazw zastępczych, by rozwiązać konflikt, z którym spotkałem się już kilka razy: Windows Presentation Foundation (WPF) — platforma do obsługi interfejsu użytkownika wchodząca w skład .NET — definiuje klasę `Path` służącą do pracy z krzywymi Béziera, wielobokami oraz innymi kształtami, jednak istnieje także klasa `Path`, ułatwiająca operacje na ścieżkach dostępu do plików i katalogów, a może się zdarzyć, że będziemy chcieli używać obu tych typów jednocześnie, by w graficznie wyświetlić zawartość pliku. W tym przypadku dodanie dyrektyw `using` dla obu przestrzeni nazw sprawi, że nazwa `Path` podana w skróconej formie będzie niejednoznaczna. Jednak jak pokazuje listing 1.8, dla każdej z tych klas można zdefiniować unikatową nazwę zastępczą.

Listing 1.8. Usuwanie niejednoznaczności poprzez użycie nazw zastępczych

```
using System.IO;
using System.Windows.Shapes;
using IoPath = System.IO.Path;
using WpfPath = System.Windows.Shapes.Path;
```

Po określeniu nazw zastępczych możemy używać nazwy `IoPath` jako synonimu klasy `Path` związanej z systemem plików oraz nazwy `WpfPath` jako synonimu klasy graficznej.

Swoją drogą, korzystając z nazw typów należących do własnej przestrzeni nazw, nie trzeba jej jawnie podawać, i to bez konieczności stosowania odpowiedniej dyrektywy `using`. To właśnie z tego powodu kod przedstawiony na listingu 1.3 nie używał dyrektywy `using Averages;`. Być może jednak zastanawiasz się, jak to wszystko działa, gdyż kod testu deklaruje inną przestrzeń nazw: `Averages.Tests`. Aby to zrozumieć, musisz poznać zagnieżdżone przestrzenie nazw.

Zagnieżdżone przestrzenie nazw

Jak już wiesz, biblioteka uruchomieniowa .NET korzysta z zagnieżdżonych przestrzeni nazw, i to korzysta powszechnie. Również Ty zapewne zechcesz z nich często korzystać. Można to zrobić na dwa sposoby. Pierwszym jest zagnieżdżanie deklaracji przestrzeni nazw, przedstawione na listingu 1.9.

Listing 1.9. Zagnieżdżanie deklaracji przestrzeni nazw

```
namespace MyApp
{
    namespace Storage
    {
        ...
    }
}
```

Alternatywnym rozwiązaniem jest podanie pełnej nazwy przestrzeni w jednej deklaracji, jak pokazałem na listingu 1.10. To rozwiązanie jest stosowane znacznie częściej. Ten styl deklarowania można stosować zarówno korzystając z nowej składni deklaracji przestrzeni nazw wprowadzonej w C# 10.0 (zobacz listing 1.10), jak i w przypadku korzystania ze starego stylu używającego nawiasów klamrowych.

Listing 1.10. Określanie zagnieżdżonych przestrzeni nazw w jednej deklaracji

```
namespace MyApp.Storage;
```

Kod umieszczany w zagnieżdżonej przestrzeni nazw będzie mógł korzystać z typów należących do tej samej przestrzeni, jak i do przestrzeni zewnętrznej, bez konieczności podawania ich pełnych nazw. Kod umieszczony w przestrzeniach z listingów 1.9 i 1.10 nie wymagałby podawania pełnych nazw ani stosowania dyrektywy `using` w odwołaniach do typów należących do przestrzeni `MyApp.Storage` i `MyApp`. To właśnie dzięki temu w przykładzie z listingu 1.3 nie musieliśmy dodawać dyrektywy `using Averages;`, by móc odwoływać się do klasy `AverageCalculator` zadeklarowanej w przestrzeni nazw `Averages`: nasz test został zadeklarowany w przestrzeni `Averages.Tests`, a ponieważ była ona zagnieżdżona w przestrzeni `Averages`, to jej kod automatycznie miał dostęp do zewnętrznej przestrzeni nazw.

W przypadku stosowania zagnieżdżonych przestrzeni nazw używana jest konwencja polegająca na tworzeniu struktury katalogów odpowiadającej hierarchii przestrzeni. Niektóre narzędzia oczekują, że będziemy stosować się do tej konwencji. Choć VS Code nie ma pod tym względem żadnych oczekiwań, to jednak Visual Studio z tej konwencji korzysta. Jeśli nasz projekt nosi nazwę *MyApp*, to wszystkie nowe klasy dodawane do projektu Visual Studio będzie domyślnie umieszczać w przestrzeni nazw `MyApp`. Gdy w takim projekcie utworzymy nowy katalog, na przykład o nazwie *Storage*, to wszystkie nowe klasy umieszczane w tym katalogu Visual Studio będzie umieszczało w przestrzeni nazw `MyApp.Storage`. Nie jest to jednak żaden wymóg — Visual Studio po prostu dodaje deklaracje przestrzeni nazw do każdego tworzonego pliku, nic jednak nie stoi na przeszkodzie, by je zmienić. Kompilator w żaden sposób nie sprawdza, czy nazwa przestrzeni nazw odpowiada strukturze katalogów. Ponieważ jednak różna narzędzia, w tym Visual Studio, stosują tę konwencję, to ułatwimy sobie życie, jeśli i my z niej skorzystamy.

Klasy

W naszym pliku *AverageCalculator.cs*, poniżej deklaracji przestrzeni nazw, została umieszczona definicja **klasy**. Ten fragment pliku przedstawia listing 1.11. W tym przypadku definicja klasy rozpoczyna się od słowa kluczowego `public`, które sprawia, że dana klasa będzie mogła być używana przez inne komponenty programu. Następnie umieszczone jest słowo kluczowe `static`, które oznacza, że nie przewidujemy możliwości tworzenia instancji czy też obiektów danej klasy — będzie ona udostępniać jedynie operacje charakterystyczne dla klasy, a nie możliwości zależne od poszczególnych obiektów. Kolejnym elementem deklaracji klasy jest słowo kluczowe `class`, po którym została podana nazwa klasy. Ponieważ kod klasy należy do przestrzeni nazw, pełna nazwa typu ma postać `Averages.AverageCalculator`. Jak widać, w języku C# do ograniczania wszelkiego rodzaju bloków kodu używane są nawiasy klamrowe ({}) — widzieliśmy je już przy okazji starszego (choć wciąż powszechnie używanego) stylu deklaracji przestrzeni nazw, a w tym przykładzie służą do określenia zasięgu klasy oraz umieszczonej w niej metody.

Listing 1.11. Klasa wraz z metodą

```
public static class AverageCalculator
{
    public static double ArithmeticMean(string[] args)
    {
        return 1.0;
    }
}
```

Klasy są mechanizmem, którego język C# używa w celu definiowania elementów posiadających stan oraz zachowanie, czyli podstawowego idiomu programowania obiektowego. Jednak w naszym przykładzie klasa zawiera tylko i wyłącznie jedną metodę. W języku C# nie ma możliwości tworzenia metod globalnych — cały pisany kod musi być umieszczony wewnątrz jakiegoś typu. Przedstawiona klasa nie jest szczególnie interesująca — pełni jedynie rolę pojemnika zawierającego punkt wejścia do programu. Znacznie bardziej interesujące zastosowania klas zostaną przedstawione w rozdziale 3.

Podobnie jak wcześniej w przypadku klasy, także metodę oznaczyliśmy jako publiczną (`public`), by zapewnić możliwość dostępu do niej z poziomu innych komponentów. Oprócz tego zadeklarowaliśmy ją jako **statyczną**, używając słowa kluczowego `static`, dzięki czemu jej wywołanie nie będzie wymagało tworzenia instancji typu, w którym została zdefiniowana (czyli w naszym przypadku klasy `AverageCalculator`). Umieszczone w następnej kolejności słowo kluczowe `double` oznacza typ danych wyniku zwracanego przez metodę — w tym przypadku będzie to liczba zmiennoprzecinkowa o podwójnej precyzji.

Poniżej deklaracji metody znajduje się jej ciało, które w przedstawionym przykładzie ogranicza się do instrukcji zwracającej konkretną wartość. Pozostaje nam zatem zmodyfikować kod umieszczony w nawiasach klamrowych wyznaczających ciało metody. Na listingu 1.12 przedstawiłem kod, który zamiast zwracać `1.0` oblicza i zwraca średnią.

Listing 1.12. Obliczanie średniej

```
return args.Select(numText => double.Parse(numText)).Average();
```

Ten kod bazuje na funkcjach bibliotecznych służących do operowania na kolekcjach, które stanowią jeden z elementów mechanizmu określanego potocznie jako LINQ (opisałem go dokładniej w rozdziale 10.). Wyjaśnię jednak pobieżnie, jak działa ten wiersz kodu. Metoda `Select` pozwala zastosować pewną operację dla każdego elementu kolekcji. W naszym przypadku operacją tą jest wywołanie metody `double.Parse` — jest to funkcja biblioteki uruchomieniowej .NET, która konwertuje liczbę zapisaną w formie łańcucha znaków na odpowiadającą jej wartość zmiennoprzecinkową o podwójnej precyzji. Następnie przekształcone w ten sposób wyniki przekazujemy do metody `Average`, która przeprowadza za nas obliczenie średniej.

Po uzupełnieniu tego kodu, jeśli teraz ponownie wykonamy polecenie `dotnet`, test okaże się, że oba testy zostaną wykonane prawidłowo. Wszystko zatem wskazuje na to, że nasz kod działa. Kiedy jednak spróbujemy zweryfikować działanie programu, wykonując go nieco nieformalnie przy użyciu poniższego polecenia, przekonamy się, że pojawia się jakiś problem:

```
./Averages/bin/Debug/net6.0/Averages 1 2 3 4 5
```

Wykonanie tego polecenia spowoduje wyświetlenie na ekranie tekstu `Hello, World!`. Napisaliśmy i przetestowaliśmy kod, który wykonuje niezbędne obliczenia, jednak na razie nie połączyliśmy go z punktem wejścia do programu. Kod wykonywany podczas uruchamiania programu jest umieszczony w pliku *Program.cs*, choć nie ma w nim nic szczególnego. Punkt wejścia do programu można bowiem umieścić w dowolnym pliku. W starszych wersjach języka C# punkt wejścia do programu był oznaczany przy użyciu statycznej metody `Main`, takiej jak przedstawiona na listingu 1.2. Jednak od wersji C# 10.0 można także tworzyć pliki zawierające wykonywalne instrukcje bez jawnego umieszczania ich w metodzie i typie — kompilator C# potraktuje je jako punkt wejścia do programu. (W projekcie może się znaleźć tylko jeden plik napisany w taki sposób — każdy program może mieć tylko jeden punkt wejścia). Kiedy zastąpimy dotychczasową zawartość pliku *Program.cs* kodem z listingu 1.13, program będzie działał w oczekiwany sposób.

Listing 1.13. Punkt wejścia do programu pobierający argumenty

```
using Averages;

Console.WriteLine(AverageCalculator.ArithmeticMean(args));
```

Warto zwrócić uwagę na zastosowanie dyrektywy `using` w powyższym kodzie — w razie korzystania z uproszczonej składni punktu wejścia do programu wprowadzonej w C# 10.0 kod umieszczany w tym pliku domyślnie nie należy do żadnej przestrzeni nazw, dlatego musimy jawnie zaznaczyć, że chcemy używać klas zdefiniowanych w przestrzeni nazw `Averages`. W drugim wierszu kodu wywołujemy napisaną wcześniej metodę, przekazując do niej `args` jako argument, a następnie zwrócony wynik wyświetlamy na ekranie przy użyciu metody `Console.WriteLine`. W przypadku pisania programów korzystających z punktu wejścia tego rodzaju `args` jest nazwą specjalną — stanowi w praktyce niejawnie zdefiniowaną zmienną lokalną zapewniającą dostęp do argumentów wiersza poleceń. Będzie to tablica łańcuchów znaków, której elementami będą kolejne argumenty. Aby uruchomić program i przekazać do niego te same argumenty co wcześniej, musimy go najpierw ponownie zbudować przy użyciu polecenia `dotnet build`.

W językach należących do rodziny języka C pierwszym argumentem jest zawsze nazwa pliku programu, gdyż także i ją użytkownik wpisuje w wierszu poleceń. C# nie korzysta z tej konwencji. Jeśli program został uruchomiony bez żadnych argumentów, to tablica przekazywana do metody `Main` będzie pusta (jej długość będzie wynosić `0`). Być może zauważyłeś, że nasz program nie radzi sobie dobrze w takiej sytuacji. Możesz jednak dodać nowy scenariusz testowy definiujący odpowiednie zachowanie, a następnie odpowiednio zmodyfikować kod programu.

Testy jednostkowe

Skoro nasz program już działa, chciałbym powrócić do przedstawionych wcześniej testów jednostkowych, gdyż ilustrują one pewne cechy C#, których nie można pokazać na przykładzie programu głównego. Jeśli ponownie spojrzymy na listing 1.3, zauważymy, że rozpoczyna się on w standardowy sposób, czyli od dyrektywy `using` oraz deklaracji przestrzeni nazw, której nazwa, `Averages.Tests`, odpowiada nazwie naszego testowego projektu. Jednak sama klasa wygląda nieco inaczej. Listing 1.14 przedstawia interesujący nas aktualnie fragment kodu z listingu 1.3.

Listing 1.14. Klasa testu jednostkowego z atrybutem

```
[TestClass]
public class WhenCalculatingAverages
{
```

Bezpośrednio przed deklaracją klasy został umieszczony tekst `[TestClass]`. Jest to tak zwany **atrybut**. Atrybuty są adnotacjami, które można dodawać do klas, metod oraz innych elementów kodu. Większość z nich sama w sobie nic nie robi — kompilator pamięta jedynie, że zostały podane, i zamieszcza odpowiednie informacje o nich w wygenerowanym kodzie wynikowym — i to wszystko. Atrybuty okazują się pożyteczne wyłącznie wtedy, gdy ktoś ich szuka; dlatego zazwyczaj są używane przez różne platformy. W naszym przypadku korzystamy z platformy testów jednostkowych firmy Microsoft, która poszukuje klas oznaczonych atrybutem `[TestClass]`. Platforma ta zignoruje wszystkie klasy, które nie posiadają tego atrybutu. Atrybuty są zazwyczaj charakterystyczne dla konkretnej platformy, a jak się przekonasz, czytając rozdział 14., można także definiować swoje własne atrybuty.

Dwie metody zdefiniowane w klasie naszego testu jednostkowego także zostały opatrzone atrybutami. Odpowiednie fragmenty kodu z listingu 1.3 przedstawiłem na listingu 1.15. Mechanizm wykonujący testy odnajdzie i wykona wszystkie metody oznaczone atrybutem `[TestMethod]`.

Listing 1.15. Metody z atrybutami

```
[TestMethod]
public void SingleInputShouldProduceSameValueAsResult()
...
[TestMethod]
public void MultipleInputsShouldProduceAverageAsResult()
...
```

W ten sposób poznałeś każdy element programu głównego oraz projektu testowego, który sprawdza, czy program działa zgodnie z założeniami.

Podsumowanie

W tym rozdziale przedstawiłem podstawową strukturę programów pisanych w języku C#. Stworzyliśmy w nim rozwiązanie zawierające dwa projekty — projekt testu i projekt samego programu. Zamieszczony w rozdziale przykład był bardzo prosty, dlatego każdy projekt składał się wyłącznie z jednego lub dwóch plików źródłowych, którymi mogliśmy się zainteresować. Tam, gdzie to było konieczne, pliki źródłowe rozpoczynały się od dyrektyw `using`, określających typy używane w kodzie umieszczonym w danym pliku. Punkt wejścia do programu został zapisany w nowy, skrócony sposób, wprowadzony w C# 10.0, natomiast pozostałe pliki źródłowe miały już klasyczną strukturę i zawierały deklaracje przestrzeni nazw, do których należała zawartość danego pliku, oraz klasy zawierające jedną lub więcej metod albo innych składowych, takich jak pola.

Typy oraz ich składowe zostały wyczerpująco opisane w rozdziale 3., jednak zanim do niego dotrzemy, w rozdziale 2. zajmiemy się kodem umieszczanym wewnątrz metod, który pozwala nam wyrażać to, co program ma robić.

Skorowidz

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

T

U

V

W

X

Z

Ź

Poznaj tajniki C# i zostań mistrzem dobrego kodu!

Język C#, sztandarowe dzieło Microsoftu, jest rozwijany stopniowo i ze starannością. Wciąż cechuje go prostota, a jego możliwości konsekwentnie rosną. Każda nowa funkcjonalność idealnie integruje się z resztą języka. W efekcie C# jest dojrzały, nowoczesny, wszechstronny i bezpieczny. Jego kolejne wydanie, oznaczone numerem 10.0, różni się od poprzednich, więc aby programowanie w C# pozostało efektywne i satysfakcjonujące, trzeba opanować niuanse nowej wersji języka i narzędzi z nim związanych.

Ta książka jest uaktualnionym wydaniem wyczerpującego przewodnika dla doświadczonych programistów. Omawia ważne koncepcje języka C# i te jego tajniki, które rzadko kiedy są opisywane w poświęconych mu publikacjach. Uwzględnia najnowsze możliwości .NET 6 i C# w wersjach 9.0 i 10.0, z czego warto wymienić: rekordy, rozszerzone możliwości dopasowywania wzorców, a także nowe techniki upraszczania kodu w celu poprawy jego efektywności. Dokładnie opisano tu typy ogólne, LINQ i techniki programowania asynchronicznego. Wyjaśniono, jak w praktyce skorzystać z tych możliwości podczas tworzenia różnego rodzaju aplikacji: chmurowych, internetowych i klasycznych dla komputerów biurkowych.

W książce między innymi:

- zmiany wprowadzone w ostatnich wersjach języka C#
- zasady korzystania z nowych możliwości języka
- zastosowanie nowych funkcjonalności C# w tworzeniu aplikacji
- nowe możliwości bibliotek klas .NET
- zastosowanie bibliotek klas .NET do praktycznych zadań programistycznych
- zwiększanie siły ekspresji kodu w C#

Ian Griffiths jest uważany za świetnego nauczyciela języka C#. Pracuje w firmie konsultingowej endjin jako doradca do spraw technicznych. Zdobył tytuł Microsoft MVP w dziedzinie technologii programistycznych. Pasjonuje się technologią, na specjalistycznych forach dla programistów jest znany z pisania długich odpowiedzi na krótkie pytania. Mieszka w Hove w Anglii.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-206-6
9 788383 222066
Cena: 169,00 zł