O'REILLY®

C# 8.0

Programowanie

Tworzenie aplikacji Windows, internetowych oraz biurowych

Helion

Ian Griffiths

Tytuł oryginału: Programming C# 8.0: Build Windows, Web, and Desktop Applications

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-6739-5

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/ch8pro
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/ch8pro.zip

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 dziesięciu lat ma najważniejsze znaczenie dla programistów tworzących aplikacje dla systemu Windows, a teraz zaczyna także być stosowany na innych platformach systemowych. W czerwcu 2016 roku firma Microsoft udostępniła wersję 1.0 .NET Core — wieloplatformowej wersji .NET, która pozwala na uruchamianie aplikacji internetowych, mikrousług oraz aplikacji wykonywanych z poziomu wiersza poleceń, pisanych w języku C# nie tylko w systemie Windows, lecz także w systemach macOS oraz Linux.

To udostępnienie obsługi dodatkowych platform sprzętowych idzie ręka w rękę z podjętą przez Microsoft decyzją o wsparciu dla programowania typu *open source*. W początkowym okresie istnienia języka C# firma Microsoft pilnie strzegła[1] jego kodów źródłowych, jednak obecnie niemal wszystko, co jest związane z tym językiem, jest tworzone w sposób otwarty, a wszelkie dodatki pochodzące od innych podmiotów są mile widziane. Propozycje nowych możliwości języka są publikowane w serwisie GitHub, pozwalając na wciągnięcie społeczności programistów w prace nad nimi już od najwcześniejszych etapów. W roku 2014 w celu przyspieszenia i popularyzacji tworzenia projektów typu *open source* na platformie .NET stworzono .NET Foundation (*https://dotnetfoundation.org*), która obecnie zarządza wieloma spośród najistotniejszych projektów związanych z językiem C# i platformą .NET rozwijanych przez firmę Microsoft (jak również wieloma innymi projektami, które nie są prowadzone przez Microsoft). Do tych projektów należą: kompilator C# (*https://github.com/dotnet/roslyn*) oraz pakiet .NET Core (*https://github.com/dotnet/core*) obejmujący środowisko uruchomieniowe, bibliotekę klas oraz narzędzia do tworzenia projektów .NET.

[1] Dotyczy to udostępnianej wcześniej wieloplatformowej wersji .NET. W 2008 roku firma Microsoft udostępniła pakiet Silverlight 2.0, który zawierał język C# i pozwalał na wykonywanie pisanego w nim kodu w przeglądarkach działających w systemach Windows i macOS. Silverlight stoczył, zakończoną porażką, wojnę z bezustannie poprawiającymi swoją zgodność i zasięg standardami HTML5 oraz JavaScript, a jego zamknięty charakter w ostatecznym efekcie działał na jego niekorzyść.

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.

Ostatnio rozszerzona została elastyczność języka C# w zakresie zarządzania pamięcią. Jego środowisko uruchomieniowe 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 też w wersji C# 7.2 (udostępnionej w 2017 roku) dodano różne 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++.

Oczywiście 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, własnej bibliotece klas oraz niezwykle obszernej bibliotece klas tworzonych przez innych twórców język C# jest pod tym względem niezwykle mocny.

.NET Framework obejmuje zarówno środowisko uruchomieniowe, jak i bibliotekę klas, z której korzystają programy pisane w języku C#. Część uruchomieniowa .NET Framework nosi nazwę *Common Language Runtime* (i jest zazwyczaj określana skrótowo jako CLR). Jej nazwa odzwierciedla fakt, że nie obsługuje ona wyłącznie języka C#, lecz wszystkie języki programowania używane w .NET Framework. Na przykład firma Microsoft udostępnia takie języki jak Visual Basic, F# oraz dostosowaną do .NET Framework 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. Zawiera ona klasy pozwalające na stosowanie wielu mechanizmów systemu operacyjnego, jak również udostępnia wiele innych rozwiązań funkcjonalnych, takich jak klasy kolekcji czy też narzędzia do przetwarzania formatu JSON.

Biblioteka klas wbudowana w .NET Framework to jednak jeszcze nie wszystko — wiele innych platform udostępnia własne biblioteki klas przeznaczonych dla .NET. 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 frameworkami. Istnieje obszerny ekosystem bibliotek .NET, zarówno komercyjnych, jak i typu *open source*; 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.

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.

Najważniejsze cechy C#

Choć pozornie najbardziej oczywistą cechą języka C# jest jego przynależność do rodziny języków, których składnia jest wzorowana na C, to jednak najprawdopodobniej jego najważniejszą cechą jest to, że jako pierwszy został zaprojektowany jako rodzimy język CLR. Zgodnie z tym, co sugeruje nazwa, CLR — Common Language Runtime — jest na tyle elastyczne, by umożliwiało obsługę wielu języków; istnieje jednak znacząca różnica pomiędzy językiem, który został rozbudowany, by można było z niego korzystać w CLR, a językiem, dla którego wykorzystanie CLR stało się jednym z głównych założeń projektowych. Doskonale obrazują to rozszerzenia .NET, jakie zostały dodane do kompilatora C++, wyraźnie rozgraniczające rodzimy świat C++ oraz zewnętrzny świat CLR.

Jednak nawet gdy nie jest stosowana odrębna składnia[2], to i tak będą się pojawiać tarcia, jeśli oba światy działają w inny sposób. Na przykład: jeśli będziemy potrzebowali kolekcji liczb, to czy powinniśmy używać standardowej klasy kolekcji języka C++, takiej jak `vector<int>`, czy też klas dostępnych w bibliotece klas .NET Framework, jak na przykład `List<int>`? Niezależnie od tego, co wybierzemy, będzie to zły wybór: biblioteki C++ nie będą miały dostępu do kolekcji .NET, natomiast API .NET nie będą w stanie korzystać z typu C++.

Jednak język C# jest ściśle powiązany z .NET, używa zarówno środowiska uruchomieniowego, jak i bibliotek kas, dzięki czemu podobny problem w ogóle nie występuje. Gdybyśmy powrócili do naszego przykładu, okazałoby się, że nie ma alternatywy dla użycia klasy `Lista<int>`. Nie występowałyby żadne problemy, gdyż biblioteki .NET zostały stworzone z myślą o tym samym świecie co język C#.

Pierwsza wersja C# udostępniała model programowania, który był ściśle powiązany z modelem CLR. Wraz z upływem czasu język stopniowo dodawał własne abstrakcje, które jednak były projektowane tak, by idealnie pasować do CLR. Dzięki temu język C# ma unikalny charakter. Oznacza to także, że aby zrozumieć C#, trzeba zrozumieć CLR oraz to, w jaki sposób wykonuje ono kod.

Kod zarządzany i CLR

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. Można zatem napisać komponent .NET, który będzie działał w 32-bitowej architekturze x86 używanej przez komputery PC od wielu lat, lecz również będzie go można używać w nowszej architekturze 64-bitowej (x64) oraz w całkowicie odmiennych architekturach,

[2] Pierwszy zestaw rozszerzeń dodanych przez Microsoft do języka C++ w większym stopniu przypominał rozszerzenia udostępniane przez sam język. W efekcie okazało się, że korzystanie z odrębnej składni do wykonywania operacji całkowicie odróżniających się od zwyczajnego C++ jest znacznie mniej kłopotliwe; dlatego też Microsoft wycofał pierwszy system (nazywany Managed C++) i zastąpił go nowszą, bardziej odmienną składnią, określaną jako C++/CLI.

takich jak ARM. (Na przykład .NET Core wprowadziło możliwość działania na urządzeniach wyposażonych w procesory ARM, 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. .NET pozwala natomiast skompilować jeden komponent, 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). Ujmując rzecz bardziej ogólnie, jeśli tylko pojawi się jakiekolwiek usprawnienie w używanym przez CLR sposobie generowania kodu — czy to obsługa nowej architektury procesorów, czy też jakieś usprawnienie wydajności — to wszystkie języki programowania .NET Framework 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 Core, w tym także taki, który został napisany 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 w trakcie działania programu, przed jej pierwszym wywołaniem. Niemniej jednak mogą być używane także inne rozwiązania. Istnieje wiele sposobów na skompilowanie kodu .NET **z wyprzedzeniem** (ang. *ahead of time*, w skrócie AOT). Dostępne jest narzędzie o nazwie NGen, które pozwala to robić w ramach czynności po zainstalowaniu aplikacji. Aplikacje Windows Store Apps, tworzone z myślą o platformie **Universal Windows Platform** (**UWP**), używają narzędzia do budowania o nazwie *.NET Native*, które wykonuje tę operację już wcześniej jako jeden z etapów procesu budowania. W platformie .NET Core 3.0 udostępniono nowe narzędzie o nazwie crossgen, które pozwala, by wszystkie aplikacje .NET Core (a nie jedynie aplikacje UPW) były kompilowane do kodu rodzimego już w czasie budowania. Niemniej jednak nawet w przypadku stosowania tych narzędzi[3] generowanie kodu wykonywalnego może nastąpić 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. (Może to zrobić niezależnie od tego, czy używana jest kompilacja JIT, czy AoT.) Zwirtualizowany charakter wykonywania zarządzanego został zaprojektowany, by umożliwiać zachowania niewidoczne dla naszego kodu; choć czasami można odczuć ich wykorzystanie i to nie tylko na podstawie wydajności.

Na przykład zwirtualizowane wykonywanie pozostawia pewną dowolność wyboru momentu, w którym środowisko wykonawcze będzie przeprowadzać określone czynności inicjalizacyjne; czasami wyniki takich optymalizacji można zaobserwować pod postacią zaskakującej kolejności kodzie zarządzanym wszechobecne są informacje o typach. Format plików narzucany przez CLI wymaga zamieszczania tych informacji, gdyż umożliwiają one stosowanie pewnych mechanizmów w środowisku uruchomieniowym. Na przykład .NET udostępnia różne automatyczne usługi do serializacji danych; pozwalają one na zapisywanie obiektów w formie binarnych lub tekstowych

[3] Wyjątkiem jest .NET Native: nie obsługuje ono kompilacji JIT, dlatego też nie pozwala na stosowanie mechanizmu kompilacji warstwowej.

reprezentacji ich stanu, które następnie można ponownie przekształcić na obiekty, być może nawet na innym komputerze. 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. Od momentu powstania tego języka Microsoft aktualizował go już kilkukrotnie, a tworząc nowe możliwości, jego projektanci zawsze mieli na myśli konkretne scenariusze. Pomimo to dokładali wszelkich starań, by zapewnić, że każdy nowy, dodawany element języka będzie przydatny także poza tymi podstawowymi scenariuszami.

Microsoft zdecydował się na przykład na dodanie do języka C# 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 firmie Microsoft 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# 3.0 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 wersjach 7. i 8. języka 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. Dostępne są specyfikacje definiujące możliwości języka oraz działanie środowiska uruchomieniowego dla wszystkich implementacji C# (patrz ramka pt. „C#, CLR oraz standardy"). Dzięki temu mogło powstać wiele implementacji języka C# oraz jego środowiska uruchomieniowego. W czasie pisania tej książki dostępne są trzy popularne implementacje: .NET Framework, .NET Core oraz Mono. Choć to nieco mylące, wszystkie trzy są obecnie rozwijane przez firmę Microsoft, mimo że początkowo sytuacja była nieco inna.

C#, CLR oraz standardy

ECMA, instytucja zajmująca się standardami, opublikowała niezależne od systemu operacyjnego specyfikacje wszelkich elementów języka C# oraz środowiska uruchomieniowego. Chodzi konkretnie o dwa dokumenty: ECMA-334 określający specyfikację języka oraz ECMA-335 definiujący *Common Language Infrastructure* (CLI, architektura wspólnego języka), czyli wirtualne środowisko, w którym są wykonywane programy pisane w C# (oraz w innych językach .NET). Zostały one także opublikowane przez Międzynarodową Organizację Normalizacyjną (ISO), jako dokumenty ISO/IEC 23270:2018 oraz ISO/IEC 23271:2012. Liczba 2018 sugeruje, że specyfikacja C# jest nowocześniejsza niż w rzeczywistości: standardy języka opublikowane przez ECMA oraz ISO odpowiadają bowiem wersji C# 5.0. W czasie powstawania książki ECMA pracuje nad zaktualizowaną specyfikacją języka, trzeba jednak mieć świadomość, że poszczególne standardy odstają od rzeczywistości o kilka lat. Choć standard IEC CLI jest jeszcze starszy, opublikowano go bowiem w roku 2012 (podobnie jak ECMA-335), to jednak specyfikacje środowiska uruchomieniowego zmieniają się rzadziej niż język, dlatego też specyfikacja CLI znacznie lepiej odpowiada bieżącym implementacjom, choć jej nazwa może sugerować co innego.

Standard ECMA-335 definiuje CLI, które zawiera jedynie zachowania wymagane od środowiska uruchomieniowego (takiego jak CLR wchodzący w skład .NET lub środowisko uruchomieniowe Mono), lecz nie tylko. Definiuje on sposób działania środowiska uruchomieniowego (określany jako *Virtual Execution System*, w skrócie VES), a także format zapisu programów wykonywalnych oraz plików bibliotek, Common Type System. Oprócz tego standard ten definiuje także podzbiór Common Type System, określany jako Common Language Specification (CLS), który powinny obsługiwać różne języki, tak by mogło być zapewnione współdziałanie pomiędzy nimi.

Widzimy zatem, że implementacja CLI firmy Microsoft obejmuje całość .NET Framework, a nie jedynie CLR, choć .NET zawiera także wiele dodatkowych możliwości, które nie należą do specyfikacji CLI. (Na przykład biblioteka klas wymagana przez CLI stanowi jedynie niewielki fragment biblioteki klas .NET). CLR pełni w efekcie rolę środowiska wykonawczego — VES — platformy .NET, choć skrót ten rzadko kiedy jest używany poza specyfikacjami; to właśnie dlatego w tej książce zazwyczaj będę wspominać właśnie o CLR (albo o **środowisku uruchomieniowym**). Natomiast terminy CTS oraz CLS są stosowane znacznie częściej, dlatego też będę ich używał w tekście książki.

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 .NET jest nazwą marki używaną przez firmę 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. Mono od samego początku był projektem typu *open source*, wspieranym przez wiele różnych firm. Od roku 2011 pracami nad projektem zarządza firma o nazwie Xamarin. W 2016 roku została ona wykupiona przez Microsoft, lecz wciąż istnieje jako nazwa marki udostępniającej Mono jako środowisko pozwalające na wykonywanie kodu C# na urządzeniach mobilnych.

A co z pozostałymi dwiema implementacjami, z których każda jest określana jako .NET?

Kilka .NET-ów (chwilowo)

Przez mniej więcej siedem lat w danym okresie istniała tylko jedna wersja .NET, jednak od 2008 roku sytuacja nieco się skomplikowała. Powodem tej zmiany było przede wszystkim udostępnienie wyspecjalizowanych wariantów .NET przeznaczonych dla różnych platform graficznych, takich jak Silverlight, oraz kilku wersji systemu Windows Phone, a także wprowadzenie (w systemie Windows 8) specjalnego typu aplikacji — Store Applications. Choć niektóre z tych wersji są jeszcze wspierane, to jednak wszystkie, z wyjątkiem Store Application (które obecnie zostały przemianowane na aplikacje Universal Windows Platform, w skrócie UPW), są już martwe. Tworzenie aplikacji UPW zostało przeniesione na .NET Core, więc pozostałe wersje .NET stały się przestarzałe.

Oprócz tych starszych wersji platformy .NET w czasie pisania niniejszej książki Microsoft wciąż udostępnia dwie wersje .NET: .NET Framework (wersję o zamkniętym kodzie źródłowym, przeznaczoną tylko dla systemu Windows) oraz .NET Core (wieloplatformową, typu *open source*). W maju 2019 roku firma Microsoft ogłosiła, że od listopada 2020 roku zamierza wrócić do pojedynczej wersji .NET. W dłuższej perspektywie czasu ten krok zmniejszy zamieszanie, choć w bliższej przyszłości dodatkowo skomplikuje on sprawy, gdyż doprowadzi do powstania jeszcze jednej wersji .NET, o której trzeba będzie wiedzieć.

Kłopotliwym aspektem całej tej sytuacji są drobne różnice w nazewnictwie różnych wersji .NET. Przez blisko 15 pierwszych lat określenie .NET Framework oznaczało kombinację dwóch elementów: środowiska uruchomieniowego oraz biblioteki klas. Środowisko uruchomieniowe nosiło nazwę CLR. Biblioteka klas miała różne nazwy: Base Class Library (BCL; nieco mylący termin, gdyż standard ECMA definiuje „BCL" jako rozwiązanie o znacznie mniejszym zakresie) oraz Framework Class Library.

Obecnie dostępna jest także platforma .NET Core. Wchodzące w jej skład środowisko uruchomieniowe nosi nazwę .NET Core Common Language Runtime (albo w skrócie: CoreCLR), co jest całkiem zrozumiałe: możemy mówić o .NET Core CLR lub o .NET Framework CLR i będzie zupełnie jasne, co mamy na myśli. Gdy w tej książce pojawi się termin CLR lub środowisko uruchomieniowe bez żadnych dodatkowych określeń, będzie to oznaczać, że opisywane rozwiązania lub możliwości odnoszą się do obu implementacji. Niestety, biblioteka klas wchodząca w skład .NET Core jest określana jako .NET Core Framework (bądź też CoreFX). To nieco kłopotliwa nazwa, gdyż przed wprowadzeniem .NET Core terminem **Framework** określano kombinację CLR i biblioteki klas. Co więcej, aby dodatkowo zagmatwać sytuację, pracownicy firmy Microsoft określają obecnie .NET Framework jako platformę „dla komputerów biurkowych" (ang. *desktop*), podkreślając w ten sposób, że nie mają na myśli .NET Core. (Sytuację dodatkowo komplikuje fakt, że wiele osób używa tej „biurkowej" wersji platformy do tworzenia aplikacji serwerowych. Co więcej, pierwsza wersja .NET Core była przeznaczona dla UWP i obsługiwała wyłącznie aplikacje przeznaczone dla systemu Windows. Cały rok minął, zanim firma Microsoft udostępniła kolejną wersję, która miała cokolwiek większe możliwości[4]. Obecnie, kiedy .NET Core 3.0 udostępnia w systemie Windows wsparcie dla dwóch frameworków do tworzenia graficznego interfejsu użytkownika — Windows Presentation Foundation (WPF) oraz Windows Forms — większość nowych aplikacji na komputery biurkowe będzie zapewne tworzona w .NET Core, a nie w wersji .NET „dla komputerów biurkowych". Na wypadek, gdyby to wszystko nie było zupełnie jasne, bieżąca sytuacja została podsumowana w tabeli 1.1.

Tabela 1.1. Nazwy komponentów dostępnych platform .NET

Platforma	Środowisko uruchomieniowe	Biblioteka klas
.NET Framework (określana także jako .NET desktop)	.NET CLR	.NET Framework Class Library
.NET Core	.NET Core CLR	.NET Core Framework

Zakładając, że firma Microsoft nie zmieni planów, w 2020 roku te nazwy znowu zostaną zmienione, a .NET Core oraz .NET Framework zostaną zastąpione zwyczajnym „.NET". W czasie pisania niniejszej książki Microsoft nie ustalił jeszcze konkretnie, jakie nazwy będą nosiły środowisko uruchomieniowe oraz biblioteka klas.

Jednak aż do momentu, kiedy ta zmiana nastąpi, istnieją dwie „aktualne" wersje platformy .NET. Każda z nich dysponuje możliwościami, których nie ma druga, i właśnie dlatego są dostępne jednocześnie. .NET Framework działa tylko w systemie Windows, natomiast .NET Core w systemach Windows, macOS oraz Linux. Choć oznacza to, że możliwości zastosowania .NET Framework są bardziej ograniczone, to z drugiej strony platforma ta może obsługiwać niektóre możliwości charakterystyczne wyłącznie dla systemu Windows. Na przykład istnieje część .NET Framework Class Library przeznaczona do obsługi usług syntezy i rozpoznawania mowy, w jakie jest wyposażony system Windows. Analogiczne możliwości nie są dostępne w .NET Core, gdyż platforma ta może działać w systemie Linux, gdzie tego typu rozwiązania systemowe bądź to nie istnieją, bądź też są na tyle odmienne, że nie można ich obsługiwać przy użyciu tego samego API.

[4] Co dziwne, ta pierwsza wersja platformy udostępniona w roku 2015 i obsługująca UWP nigdy nie otrzymała oficjalnego numeru wersji. Platform .NET Core 1.0 pojawiła się w czerwcu 2016 roku, czyli mniej więcej rok później.

Platforma .NET, która ma się pojawić w 2020 roku, będzie w zasadzie kolejną wersją .NET Core, opatrzoną jedynie bardziej chwytliwą nazwą. .NET Core jest tą wersją platformy, która w ciągu kilku ostatnich lat jest w przeważającym stopniu używana do tworzenia nowych projektów. Platforma .NET Framework wciąż jest w pełni obsługiwana, jednak obecnie traci na popularności. Na przykład wersja 3.0 frameworka ASP.NET Core przeznaczonego do tworzenia aplikacji internetowych będzie działać na platformie .NET Core, lecz nie na .NET Framework. Dlatego wycofanie .NET Framework i przekształcenie .NET Core w jedyną prawdziwą wersję .NET jest nieuniknionym zakończeniem procesu, który trwa już od kilku lat.

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. 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 przecież konkretne implementacje (.NET Core, .NET Framework oraz Mono), a z drugiej ich wersje (na przykład .NET Core 2.2 lub 3.0, .NET Framework 4.7.2 lub 4.8). Dostępne są także wcześniejsze warianty .NET, takie jak Windows Phone lub Silverlight — Microsoft wciąż wspiera wiele z nich, co obejmuje udostępnianie odpowiednich bibliotek klas w serwisie NuGet. Wielu autorów popularnych pakietów udostępnianych w serwisie NuGet jako oprogramowanie typu *open source* także zapewnia wsparcie dla wielu typów i wersji starszych frameworków.

Początkowo programiści 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 Core. Nawet jeśli kod źródłowy komponentu był w pełni zgodny z .NET Core, konieczne było przygotowanie osobnej wersji biblioteki, skompilowanej specjalnie dla danej platformy. Jeśli autor używanej biblioteki nie przygotował jej wersji jawnie przeznaczonej dla .NET Core, to korzystanie z tej biblioteki w danej wersji platformy .NET nie było możliwe. Taka sytuacja nie była korzystna dla nikogo. Twórcy komponentów musieli przygotowywać ich nowe wersje, a ponieważ wymagało to ich zaangażowania i czasu, mogło się okazać, że komponenty, które inni programiści chcieli zastosować, i tak nie były dostępne w docelowej wersji platformy.

Aby uniknąć tego problemu, firma Microsoft wprowadziła **.NET Standard**, który definiuje wspólny podzbiór zewnętrznych klas API .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 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.

Aby biblioteki .NET rozpowszechniane przy użyciu serwisu NuGet uzyskały jak największy zasięg, będą wspierać najniższą możliwą wersję .NET Standard. Wersje od 1.1 do 1.6 stopniowo zwiększały zakres możliwości funkcjonalnych kosztem obsługi coraz to mniejszej liczby obsługiwanych wersji platformy. (Na przykład, aby można było używać komponentu .NET Standard 1.3 na platformie .NET Framework, będzie to musiała być .NET Framework 4.6 lub nowsza). Wersja .NET Standard 2.0 była wielkim krokiem w przód i stanowi ważny punkt w ewolucji platformy .NET Standard: zgodnie z bieżącymi planami firmy Microsoft będzie to najwyższa wersja platformy, która będzie mogła działać na .NET Framework. Wersje .NET Framework wyższe lub równe 4.7.2 będą dysponować pełnym wsparciem dla .NET Standard 2.0, jednak już .NET Standard 2.1 i późniejsze nie będą obsługiwać żadnej wersji .NET Framework ani teraz, ani w przyszłości. Z kolei .NET Core 3.0 oraz .NET (czyli przyszłe wersje .NET Core) będą obsługiwane.

Jest wysoce prawdopodobne, że środowisko uruchomieniowe Mono firmy Xamarin także będzie obsługiwać .NET Standard 2.0, jednak będzie to już koniec drogi dla klasycznej wersji .NET Framework.

A co to wszystko oznacza dla programistów C#? Jeśli ktoś pisze kod, który nigdy nie będzie używany poza danym projektem, to jako platformę docelową może wybrać ostatnią wersję .NET Core, jeśli natomiast konieczne będzie korzystanie z pewnych możliwości charakterystycznych dla systemu Windows, może wybrać .NET Framework, dzięki czemu będzie mieć możliwość używania dowolnych pakietów NuGet przeznaczonych dla .NET Standard aż do wersji 2.0 włącznie (a to oznacza dostęp do ogromnej większości wszystkich pakietów dostępnych w serwisie NuGet). W przypadku tworzenia bibliotek, które mają być udostępniane, docelową wersją platformy powinna być .NET Standard. W przypadku tworzenia nowych bibliotek klas narzędzia programistyczne firmy Microsoft domyślnie wybierają wersję .NET Standard 2.0, co jest sensownym wyborem — można by udostępnić bibliotekę szerszemu gronu odbiorców, wybierając niższą docelową wersję platformy, jednak obecnie wersje platformy obsługujące .NET Standard 2.0 są powszechnie dostępne, dlatego też nad wyborem tych starszych wersji warto się zastanawiać wyłącznie wtedy, gdy chcemy zapewnić wsparcie programistom używającym starszych wersji .NET Framework. (Microsoft postępuje w ten sposób w większości własnych bibliotek udostępnianych w serwisie NuGet, jednak nie oznacza to wcale, że każdy musi dostosowywać się do tego samego reżimu obsługi starszych wersji). Jeśli będziemy chcieli używać jakichś nowszych możliwości (takich jak typy, które w efektywny sposób zarządzają pamięcią, opisane w rozdziale 18.), to najprawdopodobniej konieczne będzie wybranie jednaj z nowszych wersji .NET Standard jako platformy docelowej. Narzędzia programistyczne firmy Microsoft zawsze zagwarantują, że używane będą wyłącznie API dostępne w wybranej do użycia wersji .NET Standard.

Jednak firma 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 oraz Visual Studio Code

Firma Microsoft udostępnia trzy środowiska programistyczne przeznaczone dla komputerów biurkowych: Visual Studio, Visual Studio for Mac oraz Visual Studio Code. Wszystkie trzy udostępniają 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 oprogramowaniem typu *open source*, dlatego też samo nie jest oprogramowaniem o otwartym kodzie źródłowym. Dostępne jest jednak w kilku wersjach, zaczynając do darmowej, aż po taką, której koszt może przerażać.

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. Mechanizm **LiveShare** zapewnia wygodne rozwiązania do pracy zdalnej, świetnie nadające się do pracy w parach lub zdalnego przeglądania kodu. Spośród wszystkich opisywanych tu IDE Visual Studio udostępnia także najbardziej rozbudowany zestaw narzędzi do refaktoringu kodu.

W 2017 roku firma Microsoft udostępniła Visual Studio for Mac. Nie jest to jednak żadna zmodyfikowana wersja Visual Studio. Początkowo był to produkt 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 niezależnym produktem, jednak kiedy Microsoft przejął firmę, która go rozwijała (o czym wspominałem już wcześniej) i udostępnił jako produkt marki Visual Studio, został on zintegrowanych z wieloma rozwiązaniami dostępnymi w wersji IDE przeznaczonej dla systemu Windows.

Visual Studio Code (określane skrótowo jako VS Code) zostało udostępnione po raz pierwszy w 2015 roku. Jest to program typu *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 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 — nie trzeba w tym celu instalować jakichkolwiek rozszerzeń ani modyfikować ustawień konfiguracyjnych. Dlatego też zacznę właśnie od krótkiego przedstawienia sposobów korzystania z tego IDE.

Darmową wersję Visual Studio (która jest określana jako Visual Studio Community) można pobrać ze strony *https://www.visualstudio.com/*.

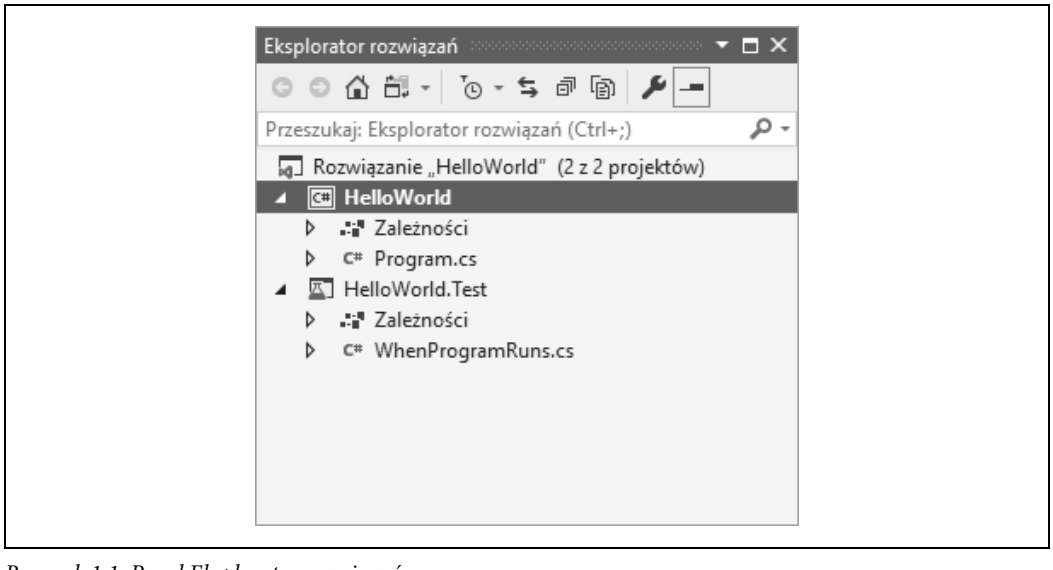
Każdy projekt aplikacji pisanej w C#, może z wyjątkiem tych najbardziej trywialnych, będzie zawierał wiele plików źródłowych, a w Visual Studio wszystkie te pliki będą należały do *projektu*. Każdy projekt generuje pojedynczy wynik, nazywany **celem** (ang. *target*). W najprostszym przypadku tym celem 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). Zaczynając od .NET Core 3.0, platforma ta może 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. .NET Framework kompiluje kod aplikacji bezpośrednio do samodzielnie uruchamianego pliku *.exe* (bez wykorzystania dodatkowego pliku *.dll*). W obu przypadkach 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 zazwyczaj mają rozszerzenie kończące się na *proj*. Na przykład projekty C# mają rozszerzenie *.csproj*, a projekty C++ rozszerzenie *.vcxproj*. Jeśli przejrzymy te pliki przy użyciu edytora tekstów, przekonamy się, że zazwyczaj zawierają one kod XML. (Choć nie zawsze tak się dzieje. Visual Studio zapewnia duże możliwości rozszerzania, a każdy rodzaj projektu jest definiowany przez **system projektu**, który może korzystać z dowolnie wybranego formatu, niemniej jednak domyślnym językiem jest właśnie XML). Pliki te opisują zawartość projektu oraz konfigurują sposób, w jaki jest on budowany. Format XML używany przez Visual Studio w plikach projektów C# może być także przetwarzany przy użyciu narzędzia *msbuild* oraz przez narzędzie *dotnet* w przypadku używania .NET Core SDK, pozwalające budować projekty z poziomu wiersza poleceń. Program VS Code także potrafi korzystać z tych plików.

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 jednostkowe dla trzech projektów głównych.

Visual Studio ułatwia nam pracę nad powiązanymi ze sobą projektami za pomocą tak zwanych **rozwiązań** (ang. *solution*). Rozwiązanie jest po prostu kolekcją projektów; i choć zazwyczaj projekty te są ze sobą powiązane, to jednak wcale nie muszą być — rozwiązanie jest w rzeczywistości jedynie pojemnikiem. Aktualnie wczytane rozwiązanie oraz wszystkie należące do niego projekty są wyświetlane w panelu *Eksplorator rozwiązań* (*Solution Explorer*) Visual Studio. Rysunek 1.1 przedstawia rozwiązanie zawierające dwa projekty. (W niniejszej książce używam Visual Studio 2019, które wtedy gdy powstawała ta książka, było najnowszą dostępną wersją). Panel ten przedstawia rozwijalne drzewo, pozwalające na wyświetlanie wszystkich projektów oraz tworzących je plików. Standardowo panel ten jest wyświetlany w prawym, górnym wierzchołku okna Visual Studio, niemniej jednak można go także ukryć lub zamknąć. Po zamknięciu można go ponownie wyświetlić, wybierając z menu opcje *Widok/Eksplorator rozwiązań* (*View/Solution*).



Rysunek 1.1. Panel Eksplorator rozwiązań

Visual Studio może wczytać projekt, wyłącznie jeśli stanowi on część rozwiązania. Tworząc zupełnie nowy projekt, można go dodać do istniejącego rozwiązania, jeśli jednak tego nie zrobimy, to Visual Studio utworzy nowe rozwiązanie. Jeśli spróbujemy otworzyć istniejący projekt, Visual Studio poszuka skojarzonego z nim rozwiązania, a jeśli nie będzie w stanie go znaleźć, to utworzy nowe.

Dzieje się tak dlatego, że wiele operacji wykonywanych w Visual Studio jest realizowanych właśnie w obrębie rozwiązania. Jeśli budujemy kod, to zazwyczaj budujemy cały kod należący do rozwiązania. Ustawienia konfiguracyjne, takie jak wybór celu (*Debug* lub *Release*), są kontrolowane na poziomie rozwiązania. Globalne operacje wyszukiwania także obejmują wszystkie pliki wchodzące w skład rozwiązania.

Samo rozwiązanie jest kolejnym plikiem tekstowym posiadającym rozszerzenie *.sln*. Co ciekawe, nie jest to plik XML — plik rozwiązania jest zwyczajnym plikiem tekstowym, choć format jego zapisu także jest rozpoznawany przez program *msbuild* oraz przez VS Code. Jeśli zajrzymy do katalogu zawierającego rozwiązanie, znajdziemy w nim także katalog o nazwie *.vs*. (Visual Studio oznacza go jako katalog ukryty, jeśli jednak skonfigurujemy Eksploratora Windows tak, by wyświetlał pliki ukryte, co programiści często robią, to go zobaczymy). Zawiera on ustawienia użytkownika, takie jak informacje o ostatnio otworzonych plikach oraz projekcie (lub projektach), który należy uruchomić w ramach sesji debugera. To właśnie ten plik zapewnia, że kiedy otworzymy projekt, wszystko będzie wyglądało mniej więcej tak jak w momencie ostatniego zamykania Visual Studio. Ponieważ są to ustawienia powiązane z użytkownikami, dlatego katalogu *.vs* zazwyczaj nie obejmuje się kontrolą wersji.

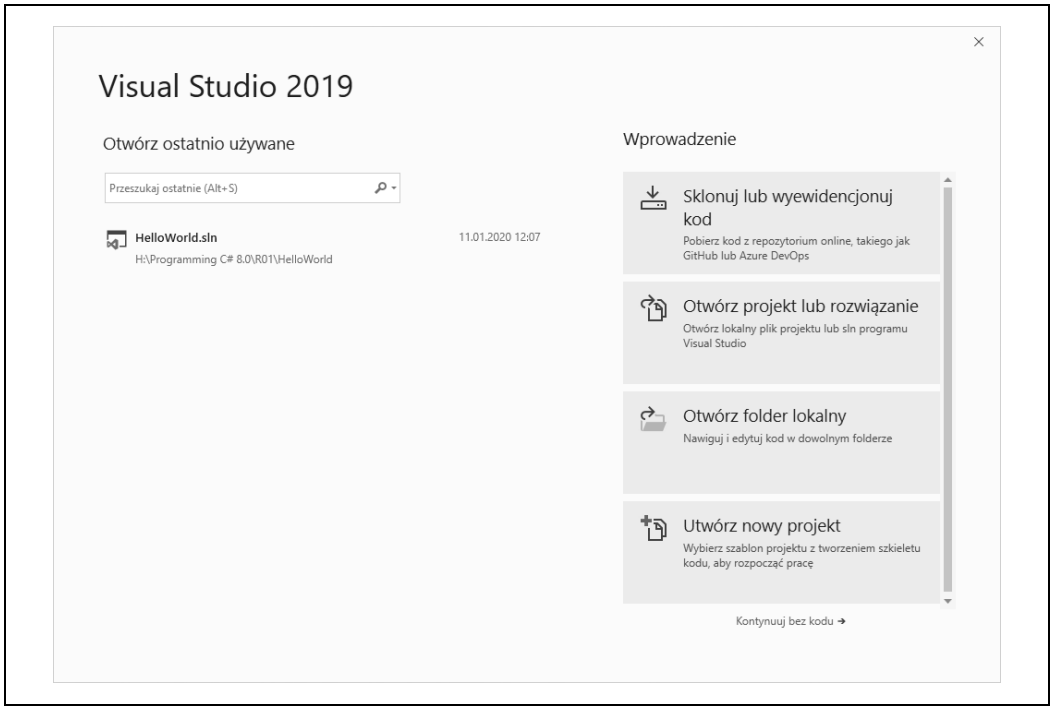
Projekt może należeć do więcej niż jednego rozwiązania. W przypadku dużej bazy kodu stosunkowo często zdarza się korzystać z kilku rozwiązań zawierających różne kombinacje projektów. W takich przypadkach zazwyczaj istnieje jakieś główne rozwiązanie zawierające wszystkie projekty, jednak nie wszyscy programiści 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 języka pokażę, jak można tworzyć nowe projekty i rozwiązania, a następnie opiszę różne możliwości, jakie Visual Studio dodaje do nowych projektów C#. Pokażę także, w jaki sposób można dodać do rozwiązania nowy projekt z testami jednostkowymi.

Kolejny podrozdział jest przeznaczony dla osób, które dopiero zaczynają używać Visual Studio — niniejsza książka jest skierowana dla doświadczonych programistów, jednak nie wymaga posiadania żadnych wcześniejszych doświadczeń związanych z programowaniem w C# ani znajomości Visual Studio. Jeśli więc czytelnik zna już podstawowe sposoby korzystania z Visual Studio, może jedynie pobieżnie przejrzeć następny podrozdział.

Anatomia prostego programu

W przypadku używania Visual Studio 2019 najprostszym sposobem utworzenia nowego projektu jest skorzystanie z powitalnego okna dialogowego *Visual Studio 2019*, wyświetlanego bezpośrednio po uruchomieniu IDE. Przedstawiłem je na rysunku 1.2.

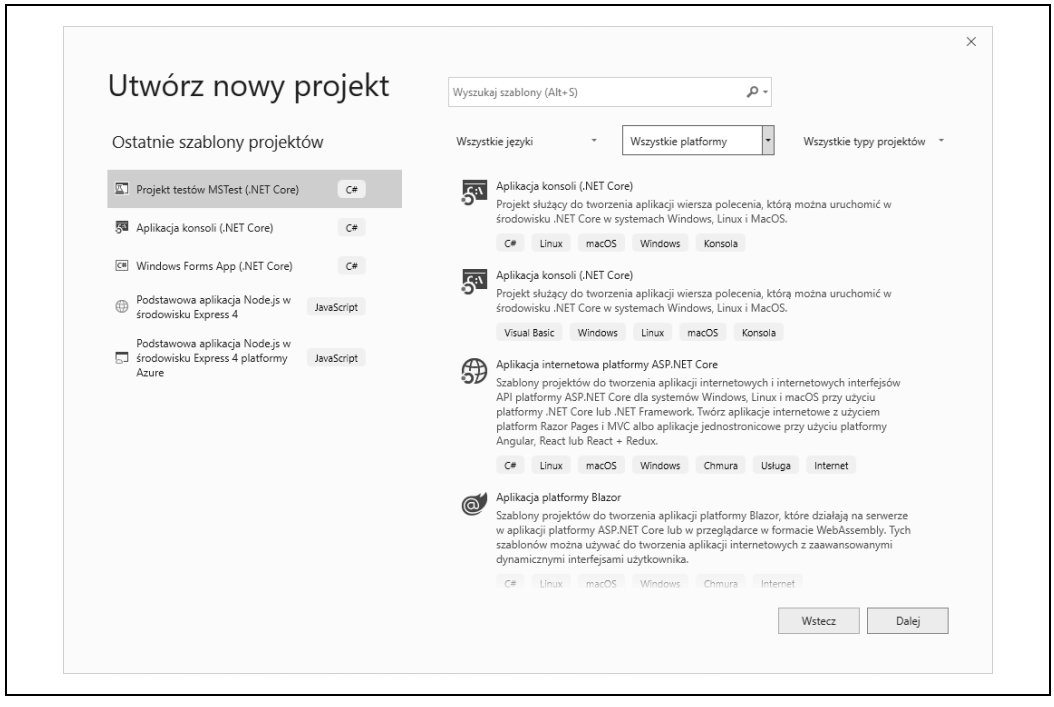


Rysunek 1.2. Powitalne okno dialogowe Visual Studio 2019

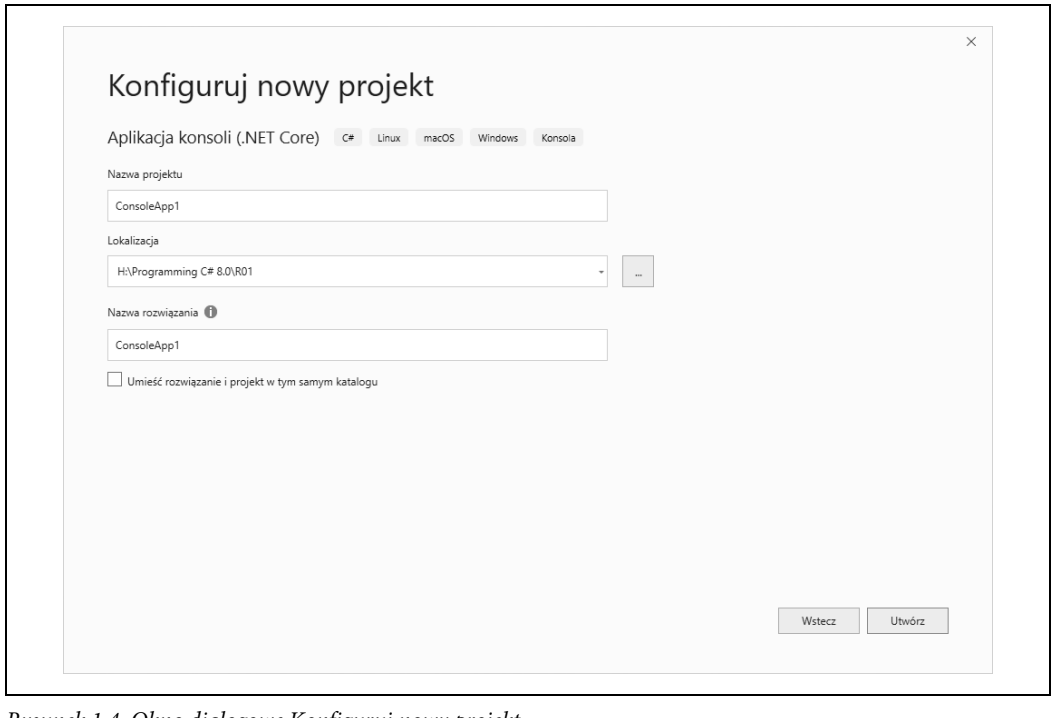
Po kliknięciu przycisku *Utwórz nowy projekt* (*Create new project*) umieszczonego w prawym dolnym rogu zostanie wyświetlone kolejne okno dialogowe, służące do tworzenia nowego projektu. Ewentualnie, jeśli Visual Studio już działa (bądź też jeśli ktoś używa starszej wersji Visual Studio, która nie dysponuje powitalnym oknem dialogowym), nowy projekt można utworzyć, wybierając z menu głównego opcje *Plik/Nowy/Projekt* (*File/New/Project*). Jeśli ktoś preferuje korzystanie ze skrótów klawiszowych, to może utworzyć nowy projekt, używając kombinacji *Ctrl+Shift+N*. W każdym z tych przypadków zostanie wyświetlone okno dialogowe *Utwórz nowy projekt* (*Create a new project*), przedstawione na rysunku 1.3.

To okno wyświetla listę dostępnych typów aplikacji. Jej konkretna zawartość będzie zależeć od zainstalowanej wersji Visual Studio oraz od tego, jakie obszary tematyczne zostały wybrane podczas jego instalacji. Jeśli został zainstalowany przynajmniej jeden obszar tematyczny zawierający język C#, to na tej liście powinna znaleźć się opcja *Aplikacja konsoli (.NET Core)*. Po jej zaznaczeniu i kliknięciu przycisku *Dalej* zostanie wyświetlone okno dialogowe *Konfiguruj nowy projekt* (*Configure your new project*), przedstawione na rysunku 1.4.

To okno prezentuje listę dostępnych typów aplikacji. Konkretny zestaw dostępnych opcji będzie zależeć od zainstalowanej wersji Visual Studio oraz pakietu roboczego wybranego podczas instalacji. Jeśli zainstalowaliśmy jeden z pakietów roboczych zawierających język C#, to na liście powinna być dostępna opcja utworzenia aplikacji konsolowej (*Aplikacja konsolowa (.NET Core)*). Jeśli zaznaczymy ją i klikniemy przycisk *Dalej* (*Next*), na ekranie zostanie wyświetlone okno dialogowe *Konfiguruj nowy projekt*, przedstawione na rysunku 1.4.



Rysunek 1.3. Okno dialogowe Utwórz nowy projekt



Rysunek 1.4. Okno dialogowe Konfiguruj nowy projekt

W tym oknie dostępne jest pole wyboru *Umieść rozwiązanie i projekt w tym samym katalogu* (*Place solution and project in the same directory*), które pozwala określić, w jaki sposób zostanie utworzone rozwiązanie skojarzone z projektem. Jeśli zaznaczymy to pole, to projekt i rozwiązanie będą mieć tę samą nazwę i zostaną umieszczone w tym samym katalogu. Jeśli jednak planujemy dodać do rozwiązania wiele projektów, to zazwyczaj preferowane będzie umieszczenie rozwiązania w odrębnym katalogu, w którym z kolei będą umieszczane podkatalogi poszczególnych projektów. Jeśli nie zaznaczymy tego pola wyboru, Visual Studio skonfiguruje wszystko właśnie w taki sposób i jednocześnie uaktywni pole tekstowe *Nazwa rozwiązania* (*Soluton name*), pozwalając w razie takiej potrzeby nadać rozwiązaniu inną nazwę niż tworzonemu projektowi. W tym przykładzie planuję dodać do rozwiązania drugi projekt, z testami jednostkowymi, dlatego też pozostawię to pole wyboru niezaznaczone. Tworzonemu projektowi nadałem nazwę *HelloWorld*, a Visual Studio nadało ją także tworzonemu rozwiązaniu, co mi w zupełności odpowiada. Po kliknięciu przycisku *Utwórz* (*Create*) zostanie utworzony nowy projekt aplikacji konsolowej pisanej w języku C#. Oznacza to, że dysponujemy już rozwiązaniem zawierającym jeden projekt.

Dodawanie projektu do istniejącego rozwiązania

Aby dodać do rozwiązania projekt testów jednostkowych, należy przejść do panelu *Eksplorator rozwiązań*, kliknąć węzeł rozwiązania (ten najwyższy) prawym przyciskiem myszy i z wyświetlonego menu kontekstowego wybrać opcje *Dodaj/Nowy projekt* (*Add/New Project*). Ewentualnie można także wyświetlić okno dialogowe *Nowy projekt* (*New project*). Jeśli to zrobimy, wyświetlone zostanie okno dialogowe niemal identyczne z tym przedstawionym na rysunku 1.3, jednak zatytułowane *Dodawanie nowego projektu* (*Add a new project*). Chciałbym dodać do rozwiązania projekt testów jednostkowych. Mógłbym w tym celu przejrzeć całą listę dostępnych typów projektów, jest jednak szybszy sposób: wystarczy wpisać słowo „test" w polu wyszukiwania u góry okna dialogowego. Mógłbym także rozwinąć listę *Wszystkie typy projektów* (*All Project Types*; umieszczoną w prawym górnym rogu okna dialogowego) i wybrać z niej opcję *Test*. W każdym z tych przypadków zostanie wyświetlona lista kilku dostępnych typów projektów testowych. Jeśli będą na niej widoczne projekty używające języków innych niż C#, to można rozwinąć listę *Wszystkie języki* (*All Languages*) i wybrać z niej opcję *C#*. Jednak nawet wtedy na liście będzie widocznych kilka typów projektów, gdyż Visual Studio obsługuje kilka różnych frameworków testowych. Ja wybiorę projekt typu *Projekt testów MSTest (.NET Core)*.

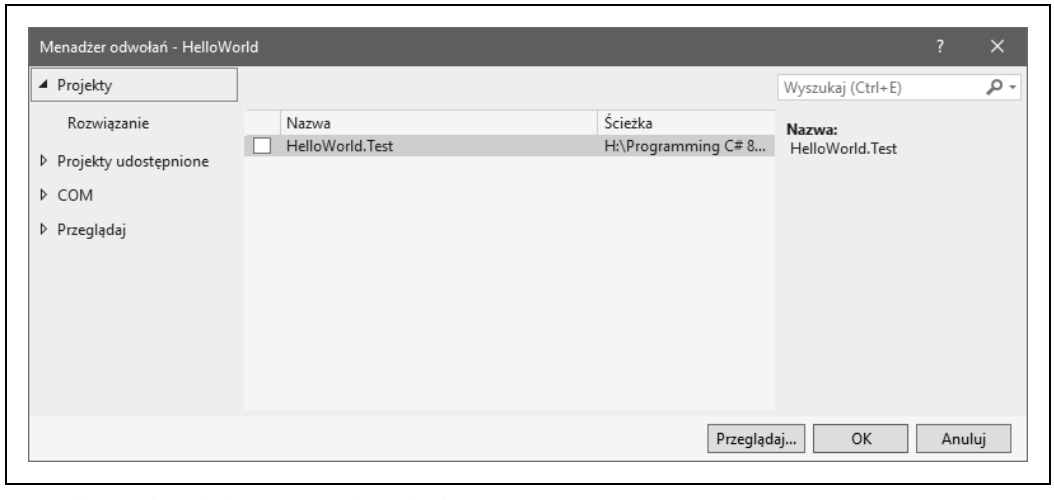
Kliknięcie przycisku *Dalej* spowoduje ponowne wyświetlenie okna dialogowego *Konfiguruj nowy projekt*. Ten nowy projekt będzie zawierał testy dla projektu *HelloWorld*, dlatego też nadam mu nazwę *HelloWorld.Tests*. (A swoją drogą nie ma wymogu stosowania takiej konwencji nazewniczej — mogłem nadać temu projektowi dowolną inną nazwę). Po kliknięciu przycisku *Utwórz* Visual Studio utworzy drugi projekt i oba będą widoczne w panelu *Eksplorator rozwiązań*, który teraz będzie wyglądał podobnie do tego z rysunku 1.1.

Ten dodatkowy projekt testowy będzie miał za zadanie zapewnić, że nasz główny projekt będzie działał zgodnie z naszymi oczekiwaniami. Osobiście preferuję styl programowania, w którym testy pisze się przed testowanym kodem, dlatego też zaczniemy właśnie od tego projektu. Aby projekt testowy mógł robić to, czego od niego oczekujemy, musi mieć dostęp do kodu umieszczonego w projekcie *HelloWorld*. Visual Studio nie próbuje odgadywać, jakie są zależności pomiędzy

poszczególnymi projektami w rozwiązaniu. Choć w naszym przykładzie są tylko dwa projekty, to gdyby Visual Studio miało samo odgadnąć, który z nich jest zależny od drugiego, zapewne odgadłoby źle, gdyż projekt *HelloWorld* generuje plik *.exe*, a projekt testowy — bibliotekę *.dll*. Najbardziej oczywiste byłoby zatem przypuszczenie, że to program *.exe* jest zależny od biblioteki *.dll*; jednak w naszym przykładzie występuje dosyć niecodzienna sytuacja, gdyż to właśnie biblioteka (czyli projekt testowy) jest zależna od kodu aplikacji.

Odwołania do innych projektów

Aby przekazać Visual Studio informacje o zależnościach pomiędzy naszymi dwoma projektami, trzeba kliknąć prawym przyciskiem myszy węzeł *Zależności* (*Dependencies*) projektu *HelloWorld.Test* w panelu *Eksplorator rozwiązań* i wybrać opcję *Dodaj odwołanie* (*Add Reference*). Gdy to zrobimy, na ekranie zostanie wyświetlone okno dialogowe *Menedżer odwołań* (*Reference Manager*) przedstawione na rysunku 1.5. Po lewej stronie okna wybierany jest rodzaj odwołania, które chcemy utworzyć — w tym przypadku interesuje nas odwołanie do innego projektu należącego do tego samego rozwiązania. Dlatego należy rozwinąć sekcję *Projekty* (*Project*) i zaznaczyć opcję *Rozwiązanie* (*Solution*). W efekcie w środkowej części okna dialogowego zostanie wyświetlona lista dostępnych projektów. W naszym przykładzie pojawi się tylko jeden projekt, należy go zatem zaznaczyć i kliknąć przycisk *OK*.

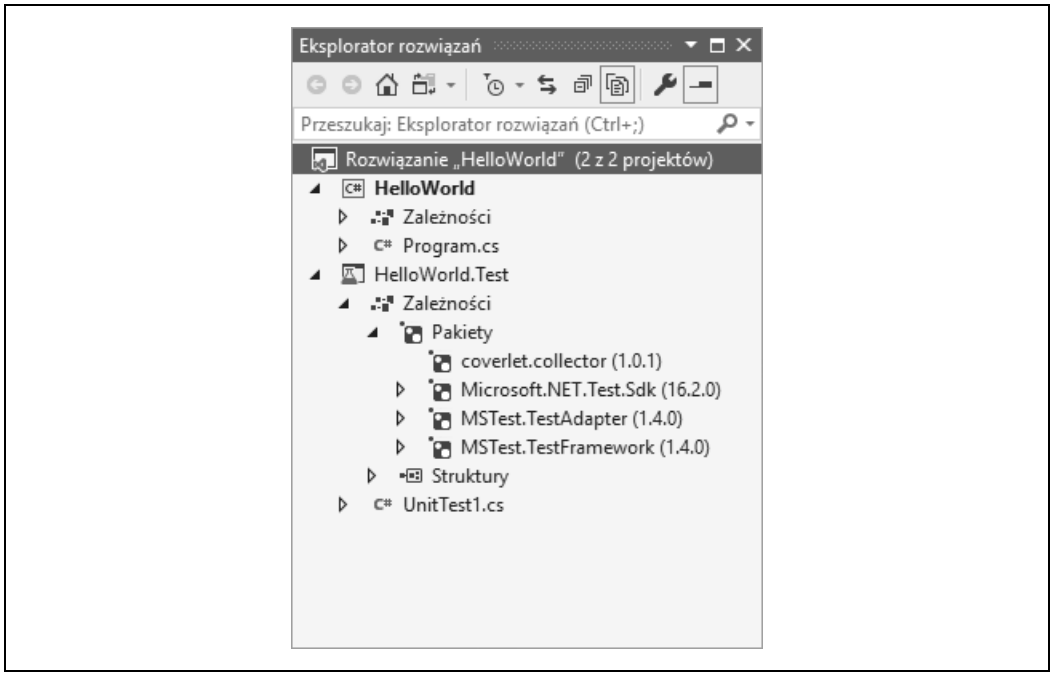


Rysunek 1.5. Okno dialogowe Menedżer odwołań

Odwołania do bibliotek zewnętrznych

Choć biblioteka klas .NET niewątpliwie jest bardzo rozbudowana, to jednak nie zawiera klas na wszelkie możliwe sytuacje. Istnieją tysiące bardzo użytecznych bibliotek przeznaczonych dla platformy .NET, a wiele z nich jest dostępnych za darmo. Microsoft udostępnia coraz to więcej bibliotek, które nie wchodzą w skład biblioteki klas .NET. Visual Studio obsługuje dodawanie odwołań do tych bibliotek, używając przy tym serwisu NuGet, o którym już wcześniej wspominałem. Okazuje się, że nasz projekt już korzysta z tej możliwości — choć wybraliśmy framework

testowy MSTest stworzony przez Microsoft, to okazuje się, że nie jest on wbudowanym elementem platformy .NET. (Z usług testowych zazwyczaj nie korzysta się podczas wykonywania programów, dlatego też nie ma potrzeby, by były one wbudowane w bibliotekę klas dostarczaną wraz z platformą). Jeśli rozwiniemy węzeł *Zależności* dla projektu *HelloWorld.Test*, poniżej niego zostaną wyświetlone różne pakiety NuGet, które przedstawiłem na rysunku 1.6. (Używane numery wersji poszczególnych pakietów mogą być nieco wyższe, gdyż biblioteki te są cały czas rozwijane).

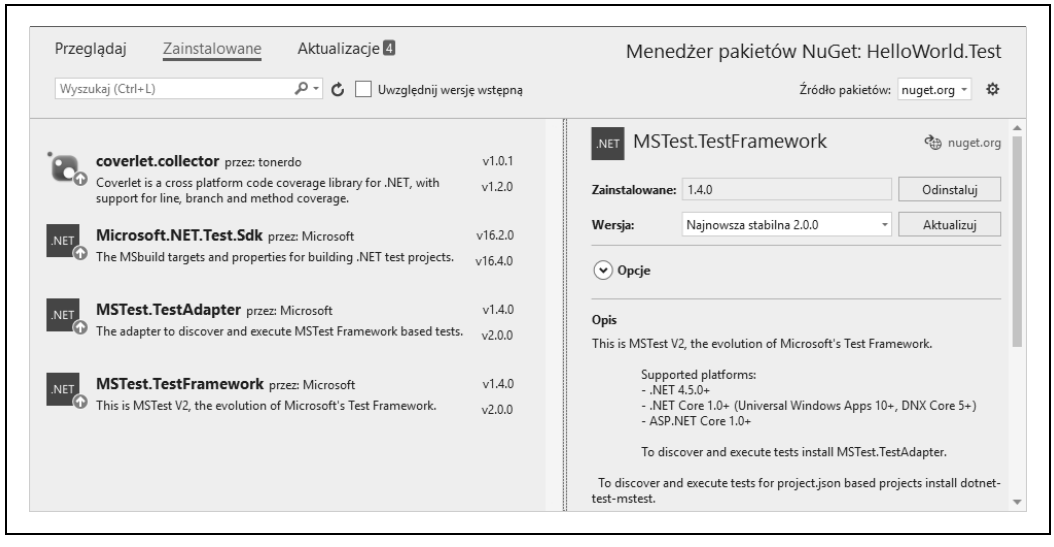


Rysunek 1.6. Odwołania do pakietów NuGet

Można używać czterech pakietów związanych z testowaniem, które są dostępne w szablonie projektu testowego Visual Studio. NuGet jest systemem bazującym na pakietach, dlatego zamiast dodawać referencje do poszczególnych bibliotek DLL, dodajemy referencje do pakietu mogącego zawierać wiele bibliotek oraz wszelkie inne pliki, które mogą być potrzebne do ich działania.

Publiczne repozytorium pakietów prowadzone przez Microsoft w serwisie NuGet (*http://nuget.org/*) zawiera kopie wszystkich bibliotek, których Microsoft nie dołącza do biblioteki klas .NET, a które jednocześnie w pełni wspiera. (Przykładem takiej biblioteki może być framework testowy używany w tym przykładzie, kolejnym może być framework ASP.NET Core). Centralne repozytorium NuGet nie jest przeznaczone do przechowywania pakietów stworzonych wyłącznie przez firmę Microsoft. Każdy może udostępniać w nim swoje pakiety, dlatego też właśnie w tym serwisie znajdziemy ogromną większość bezpłatnie dostępnych bibliotek dla platformy .NET.

Visual Studio potrafi przeszukiwać główne repozytorium NuGet. Jeśli klikniemy prawym przyciskiem myszy na projekcie bądź jego węźle *Zależności* i z wyświetlonego menu wybierzemy opcję *Zarządzaj pakietami NuGet* (*Manage NuGet Packages*), to zostanie wyświetlone okno *Menedżer pakietów NuGet* (*NuGet Package Manager*), przedstawione na rysunku 1.7. Po jego lewej stronie wyświetlana jest lista pakietów pochodzących z repozytorium NuGet. Jeśli u góry okna wybierzemy kartę *Zainstalowane* (*Installed*), to w oknie będą widoczne wyłącznie aktualnie używane pakiety. Jeśli jednak przejdziemy na kartę *Przeglądaj* (*Browse*), to okno początkowo wyświetli listę najpopularniejszych dostępnych pakietów, jednak dzięki umieszczonemu u góry polu tekstowemu będzie także można wyszukiwać konkretne biblioteki.



Rysunek 1.7. Menedżer pakietów NuGet

W repozytorium NuGet można także udostępniać swoje własne repozytoria. Wiele firm używa własnych repozytoriów chronionych firmowymi zaporami sieciowymi, a następnie udostępnia przygotowane pakiety innym programistom, bez publicznego udostępniania ich kodu. Witryna *https://myget.org* specjalizuje się w przechowywaniu i udostępnianiu pakietów w internecie, a możliwość przechowywania prywatnych pakietów zapewniają serwisy Azure DevOps oraz GitHub. Repozytorium pakietów można także przechowywać w lokalnym systemie plików. NuGet można skonfigurować w taki sposób, by oprócz głównego repozytorium publicznego przeszukiwane były także dowolne inne.

Bardzo ważną cechą pakietów NuGet jest możliwość określania zależności od innych pakietów. Na przykład, jeśli przyjrzymy się pakietowi `Microsoft.NET.Test.Sdk` widocznemu na rysunku 1.6, to zauważymy przy jego węźle niewielki trójkącik informujący o tym, że węzeł można rozwinąć. Kiedy go klikniemy, zostanie wyświetlona lista pakietów, od których dany pakiet zależy, w tym przypadku będzie nim na przykład pakiet `Microsoft.CodeCoverage`. Dzięki temu, że pakiety opisują swoje zależności, Visual Studio może automatycznie pobierać wszystkie niezbędne pakiety.

Pisanie testu jednostkowego

Teraz musimy napisać sam test. Aby ułatwić nam rozpoczęcie pracy, Visual Studio wygenerowało klasę testową umieszczoną w pliku *UnitText1.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. Jak można się domyślić na podstawie nazwy naszego projektu, nasz program będzie miał tylko jedno zadanie: po uruchomieniu powinien wyświetlić komunikat „Witaj, świecie!". Dlatego też zmienimy nazwę klasy testowej na *WhenProgramRuns.cs*[5]. Ten test powinien sprawdzić, że po uruchomieniu program wyświetli prawidłowy komunikat. Sam test jest bardzo prosty, niestety jednak znacznie trudniejsze jest dotarcie do punktu, w którym będziemy mogli go wykonać. Pełny kod źródłowy klasy testowej został przedstawiony na listingu 1.1; kod testu jest umieszczony na samym dole i został wyróżniony pogrubieniem.

Listing 1.1. Klasa testu jednostkowego naszego pierwszego programu

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

namespace HelloWorld.Tests
{
    [TestClass]
    public class WhenProgramRuns
    {
        private string _consoleOutput;

        [TestInitialize]
        public void Initialize()
        {
            var w = new System.IO.StringWriter();
            Console.SetOut(w);

            Program.Main(new string[0]);

            _consoleOutput = w.GetStringBuilder().ToString().Trim();
        }

        [TestMethod]
        public void SaysHelloWorld()
        {
            Assert.AreEqual("Witaj, świecie!", _consoleOutput);
        }
    }
}
```

[5] Kiedy program zostanie uruchomiony — *przyp. tłum.*

Każdy z fragmentów powyższego kodu zostanie opisany nieco później, po przedstawieniu samego programu. Jak na razie najbardziej interesującym aspektem tego przykładu jest jego metoda `SaysHelloWorld`, która definiuje zachowanie, które musi realizować nasz program. Test stwierdza, że w wyniku wykonania program powinien wygenerować komunikat „Witaj, świecie!”. Jeśli program tego nie zrobi, test zostanie uznany za nieudany. Sam test jest przyjemnie prosty, natomiast nieco dziwny jest kod, który przygotowuje jego wykonanie. Problem polega na tym, że pierwszy program, który z mocy prawa jest wymagany przez wszystkie książki programistyczne, nie nadaje się najlepiej do przeprowadzania testów jednostkowych pojedynczych klas lub metod, gdyż tak naprawdę nie można testować czegoś mniejszego od całego programu. My chcemy sprawdzić, czy program generuje na konsoli konkretny komunikat. W rzeczywistej aplikacji można by stworzyć jakąś abstrakcję reprezentującą miejsce, do którego kierowane są wyniki, a testy jednostkowe udostępniłyby podrobioną wersję tej abstrakcji, służącą do celów testowych. My jednak chcielibyśmy, by nasza aplikacja (którą kod z listingu 1.1 jedynie testuje) była w pełni zgodna z duchem standardowych programów „Witaj, świecie!”. Aby uniknąć zbytniego komplikowania programu, test został skonstruowany w taki sposób, że przechwytuje wyniki przesyłane na konsolę, dzięki czemu możemy sprawdzić, czy program wyświetlił to, co powinien. (Możliwości, które zostały przy tym użyte, są zdefiniowane w przestrzeni nazw `System.IO` i zostały opisane w rozdziale 15.).

Jest także drugi problem. Przeważnie testy jednostkowe z definicji badają działanie jakiegoś izolowanego i zazwyczaj małego fragmentu programu. Jednak w naszym przykładzie program jest tak prosty, że posiada tylko jedną metodę nadającą się do testowania, a ta jest wykonywana po uruchomieniu programu. Oznacza to, że nasz test będzie musiał wywołać punkt wejścia do programu. Można by to zrobić, uruchamiając program *HelloWorld* w zupełnie nowym procesie, jednak w takim przypadku przechwycenie generowanych przez niego wyników byłoby jeszcze bardziej złożone niż w razie przechwytywania wyników w ramach jednego procesu, które zastosowaliśmy w przykładzie z listingu 1.1. Zamiast tego jawnie wywołujemy punkt wejścia do programu. W programach pisanych w C# punkt wejścia do programu jest zazwyczaj metodą o nazwie `Main`, zdefiniowaną w klasie `Program`. Listing 1.2 przedstawia odpowiedni wiersz kodu z listingu 1.1, w którym wywołujemy tę metodę, przekazując do niej pustą tablicę, symulując przez to uruchomienie programu bez przekazania do niego argumentów z wiersza poleceń.

Listing 1.2. Wywoływanie metody

```
Program.Main(new string[0]);
```

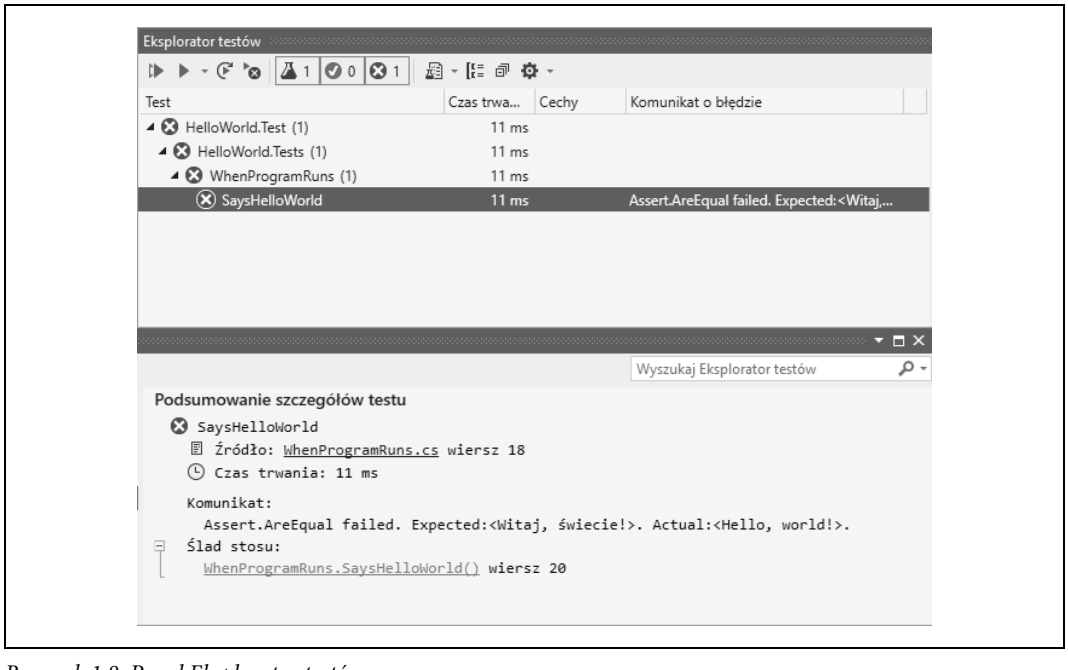
Niestety, takie rozwiązanie stwarza pewien problem. Punkt wejścia do programu jest zazwyczaj dostępny wyłącznie w trakcie jego działania — jest to szczegół implementacyjny programu i zazwyczaj nie ma żadnego powodu, by go udostępniać publicznie. Jednak w tym przykładzie zrobimy wyjątek, gdyż metoda `Main` stanowi jedyne miejsce naszego programu, w którym znajduje się jakiś kod. A zatem aby nasz kod został skompilowany, konieczne będzie wprowadzenie pewnej modyfikacji w programie głównym. Wprowadzimy ją w pliku *Program.cs* projektu *HelloWorld*, w wierszu przedstawionym na listingu 1.3. (Wszystko zostanie wyjaśnione już niebawem).

Listing 1.3. Udostępnienie punktu wejścia do programu

```
public class Program
{
    public static void Main(string[] args)
    {
...
```

Na początku obu wierszy kodu dodaliśmy słowo kluczowe `public`, dzięki czemu kod programu stanie się dostępny dla testów i w końcu będzie można skompilować kod z listingu 1.1. Istnieją także inne sposoby pozwalające uzyskać taki sam efekt. Można pozostawić klasę w niezmienionej postaci, lecz oznaczyć metodę modyfikatorem `internal`, a następnie użyć w programie klasy `InternalsVisibleToAttribute`, by uzyskać dostęp do testu. Niemniej jednak ochrona wewnętrzna oraz korzystanie z atrybutów podzespołów są zagadnieniami opisywanymi w dalszej części książki (odpowiednio w rozdziałach 3. i 14.), dlatego też chciałem, by ten pierwszy przykład był możliwie jak najprostszy. Alternatywne rozwiązanie przedstawiłem w rozdziale 14.

Teraz jesteśmy już gotowi do wykonania testu. W tym celu należy wybrać z menu opcje *Test/Eksplorator testów* (*Test/Test Explorer*), aby wyświetlić panel *Eksplorator testów* (*Test Explorer*). Następnie musimy zbudować projekt, wybierając opcje *Kompiluj/Kompiluj rozwiązanie* (*Build/Build Solution*). Kiedy to zrobimy, w panelu *Eksplorator testów* zostanie wyświetlona lista wszystkich testów zdefiniowanych w rozwiązaniu. Jak widać na rysunku 1.8, nasza metoda testowa `SayHelloWorld` została odnaleziona. Kliknięcie przycisku *Uruchom wszystkie testy* (*Run All Tests*; mającego kształt podwójnej strzałki i umieszczonego w lewym górnym rogu panelu) spowoduje wykonanie testu, co zakończy się niepowodzeniem, gdyż jeszcze nie wprowadziliśmy żadnych zmian w domyślnym kodzie naszego głównego programu. Wyświetlony komunikat o błędzie jest widoczny po prawej stronie rysunku 1.8. Stwierdza on, że oczekiwany był komunikat „Witaj, świecie!”, lecz komunikat, który faktycznie został wygenerowany na konsoli, miał inną postać.



Rysunek 1.8. Panel Eksplorator testów

Nadszedł zatem czas, aby zająć się naszym głównym programem *HelloWorld* i dodać do niego brakujący kod. Kiedy tworzyliśmy projekt, Visual Studio wygenerowało różne pliki, w tym także plik *Program.cs*, zawierający punkt wejścia do programu. Kod tego pliku został przedstawiony

na listingu 1.4, przy czym zawiera on już modyfikacje przedstawione na listingu 1.3. Poniżej listingu opiszę poszczególne elementy tego programu, gdyż będzie to stanowić przydatne wprowadzenie do prezentacji ważnych elementów składni i struktury programów pisanych w języku C#.

Listing 1.4. Program.cs

```
using System;

namespace HelloWorld
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Witaj, świecie!");
        }
    }
}
```

Plik rozpoczyna się od *dyrektywy* `using`. Jest ona co prawda opcjonalna, jednak praktycznie każdy plik źródłowy zawiera jedną lub kilka takich dyrektyw, które informują kompilator o tym, jakich **przestrzeni nazw** chcemy używać. To prowadzi nas do oczywistego pytania: Czym jest **przestrzeń nazw** (ang. *namespace*)?

Przestrzenie nazw

Przestrzenie nazw wnoszą porządek i strukturę do świata, który bez nich byłby jednym wielkim chaosem. Biblioteka klas .NET zawiera bardzo wiele klas, a wiele innych istniejących bibliotek udostępnia kolejne klasy, nie wspominając w ogóle o tych, które sam napiszesz. W przypadku tak wielkiej liczby elementów posiadających własne nazwy pojawiają się dwa podstawowe problemy. Przede wszystkim bardzo trudno jest zagwarantować niepowtarzalność nazw, chyba że są one bardzo długie lub w ich skład wchodzi losowy ciąg znaków. 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. Typy opracowane przez firmę Microsoft znalazły się w odrębnych przestrzeniach. Jeśli typy są elementem .NET, to należą do przestrzeni nazw rozpoczynających się od słowa `System`, jeśli natomiast są elementami technologii firmy Microsoft — do przestrzeni nazw rozpoczynających się od `Microsoft`. Biblioteki stworzone i dostarczane przez inne firmy także zazwyczaj mają nazwy rozpoczynające się od nazwy firmy; natomiast nazwy bibliotek tworzonych w ramach projektów otwartych rozpoczynają się zazwyczaj 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 klas .NET, który zostanie opublikowany jako prośba wprowadzenia zmian (ang. *pull request*) do repozytorium *https://github.com/dotnet/corefx*, 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.

Przestrzenie nazw zazwyczaj stanowią także pewną podpowiedź dotyczącą przeznaczenia typu. Na przykład wszystkie typy związane z obsługą plików należą do przestrzeni nazw `System.IO`, natomiast te związane z komunikacją sieciową — do przestrzeni `System.Net`. Przestrzenie nazw mogą także tworzyć hierarchie. Dlatego też przestrzeń nazw `System` .NET Framework nie zawiera żadnych typów. Zawiera jednak 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.RegularExpressions` — 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 równie męczącego i tu właśnie przydaje się dyrektywa `using` umieszczona na początku kodu z listingu 1.4. Choć nasz prosty przykład używa tylko jednej takiej dyrektywy, to jednak zazwyczaj będzie ich więcej. Dyrektywa `using` określa przestrzeń nazw, w jakiej zostały zdefiniowane typy, których chcemy używać w konkretnym pliku źródłowym. Zazwyczaj będziemy edytować tę listę, dostosowując ją do wymagań kodu w konkretnym pliku. W naszym przypadku Visual Studio dostało dyrektywę `using System` już podczas generowania projektu. W zależności od kontekstu Visual Studio dobiera różne zestawy 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.

Dzięki umieszczeniu w pliku deklaracji `using` będzie w nim można stosować skrócone, a nie pełne nazwy klas. Wiersz, który sprawia, że nasza klasa realizuje swoje zadanie, korzysta z klasy `System.Console`, jednak dzięki wcześniejszemu użyciu dyrektywy `using` będzie się do niej odwoływał, używając skróconej nazwy — `Console`. W rzeczywistości jest to jedyna klasa używana w naszym programie, więc nie ma sensu dodawać do niego żadnych innych dyrektyw `using`.

Wcześniej w tym rozdziale można się było przekonać, że opcja *Zależności* panelu *Eksplorator rozwiązań* opisuje wszystkie biblioteki używane przez program. 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 klas .NET Framework zawiera plik *System.Core.dll*, jednak nie ma przestrzeni nazw `System.Core`. Dlatego też konieczne jest przekazanie Visual Studio informacji o tym, których bibliotek potrzebuje nasz program, jak również określenie, które przestrzenie nazw będą używane w poszczególnych plikach źródłowych. 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 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 1.5 korzysta z nazw zastępczych, by rozwiązać konflikt, z którym spotykałem się już kilka razy: Windows Presentation Foundation (WPF) — platforma do obsługi interfejsu użytkownika wchodząca w skład .NET Framework — 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 graficzny sposób 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 1.5, dla każdej z tych klas można zdefiniować unikatową nazwę zastępczą.

Listing 1.5. 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.

Wróćmy jednak do naszego przykładowego programu *HelloWorld*. Bezpośrednio za dyrektywami `using` została umieszczona **deklaracja przestrzeni nazw**. Dyrektywy `using` deklarują, które przestrzenie nazw będą używane w kodzie, natomiast deklaracja przestrzeni nazw określa, do jakiej przestrzeni nazw należy nasz kod. Odpowiedni fragment kodu z listingu 1.4 został przedstawiony na listingu 1.6. Bezpośrednio za deklaracją przestrzeni nazw jest umieszczany otwierający nawias klamrowy ({). Cały kod znajdujący się pomiędzy tym nawiasem oraz odpowiadającym mu zamykającym nawiasem klamrowym umieszczonym na końcu pliku będzie należał do przestrzeni nazw `HelloWorld`.

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 testu przedstawionego na listingu 1.1 nie używał dyrektywy `using HelloWorld;` — niejawnie miał on bowiem dostęp do tej przestrzeni nazw, gdyż był umieszczony wewnątrz deklaracji `namespace HelloWorld.Tests`.

Listing 1.6. Deklaracja przestrzeni nazw

```
namespace HelloWorld
{
```

Podczas tworzenia projektu Visual Studio generuje deklaracje przestrzeni nazw odpowiadające nazwie projektu i używa ich w tworzonych plikach źródłowych. Można to jednak zmienić — projekt może zawierać dowolną kombinację przestrzeni nazw, a same nazwy można dowolnie zmieniać. Jeśli jednak zdecydujemy, że przestrzeń nazw ma się różnić od nazwy projektu, i będziemy chcieli jej używać konsekwentnie, to warto o tym poinformować Visual Studio, gdyż jej deklaracja nie pojawi się wyłącznie w pierwszym wygenerowanym pliku — *Program.cs*. Domyślnie Visual Studio dodaje deklarację przestrzeni nazw odpowiadającej nazwie projektu do każdego nowego pliku. Jednak nazwę tę możemy zmienić w ustawieniach projektu. W tym celu należy kliknąć prawym przyciskiem myszy węzeł projektu w panelu *Eksplorator rozwiązań* i wybrać opcję *Właściwości* (*Properties*) z menu podręcznego, aby wyświetlić okno dialogowe właściwości projektu. Następnie należy przejść na kartę *Aplikacja* (*Application*) i zmienić wartość podaną w polu tekstowym *Domyślna przestrzeń nazw* (*Default namespace*). Łańcuch znaków wpisany w tym polu będzie używany w deklaracjach przestrzeni nazw umieszczanych w każdym nowym pliku dodawanym do projektu. (Zmiana tej nazwy nie spowoduje jednak aktualizacji wszystkich plików już należących do projektu). Operacja ta dodaje do pliku *.csproj* właściwość `<RootNamespace>`.

Zagnieżdżone przestrzenie nazw

Jak się już przekonałeś, biblioteka klas .NET korzysta z zagnieżdżonych przestrzeni nazw i to korzysta powszechnie. Zapewne będziesz zagnieżdżać także swoje własne przestrzenie nazw, chyba że tworzysz naprawdę wyjątkowo prosty przykład. Można to zrobić na dwa sposoby. Pierwszym z nich jest zagnieżdżanie deklaracji przestrzeni nazw, przedstawione na listingu 1.7.

Listing 1.7. Zagnieżdżanie deklaracji przestrzeni nazw

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

Alternatywnym rozwiązaniem jest podanie pełnej nazwy przestrzeni w jednej deklaracji, jak pokazałem na listingu 1.8. To rozwiązanie jest stosowane znacznie częściej.

Listing 1.8. 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 1.7 i 1.8 nie wymagałby podawania pełnych nazw ani stosowania dyrektywy `using` w odwołaniach do typów należących do przestrzeni `MyApp.Storage` oraz `MyApp`.

W przypadku stosowania zagnieżdżonych przestrzeni nazw używana jest konwencja polegająca na tworzeniu struktury katalogów odpowiadającej hierarchii przestrzeni. Jeśli nasz projekt nosi nazwę *MyApp*, to wszystkie nowe klasy dodawane do projektu Visual Studio będzie domyślnie umieszczać w przestrzeni nazw `MyApp`. Jeśli w takim projekcie utworzymy nowy katalog (co można zrobić, korzystając z panelu *Eksplorator rozwiązań*), 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 Visual Studio stosuje taką konwencję, to ułatwimy sobie życie, jeśli i my z niej skorzystamy.

Klasy

W naszym pliku *Program.cs*, wewnątrz deklaracji przestrzeni nazw, została umieszczona definicja **klasy**. Ten fragment pliku (wraz ze zmienionym wcześniej słowem kluczowym `public`) przedstawia listing 1.9. Po słowie kluczowym `class` umieszczana jest nazwa klasy. Ponieważ kod klasy znajduje się wewnątrz deklaracji przestrzeni nazw, zatem pełna nazwa naszego typu ma postać `HelloWorld.Program`. Jak widać, w języku C# do ograniczania wszelkiego rodzaju bloków kodu używane są nawiasy klamrowe ({}) — widzieliśmy je już przy okazji deklaracji przestrzeni nazw, a w tym przykładzie służą do określenia zasięgu klasy oraz umieszczonej wewnątrz niej metody.

Listing 1.9. Klasa wraz z metodą

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

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. Dlatego jedyna klasa naszego przykładowego programu nie jest szczególnie interesująca — jej jedynym zadaniem jest pełnienie roli pojemnika zawierającego punkt wejścia do programu. Znacznie bardziej interesujące zastosowania klas zostaną przedstawione w rozdziale 3.

Punkt wejścia do programu

Domyślnie kompilator C# poszukuje metody o nazwie `Main` i jeśli uda mu się ją znaleźć, to automatycznie użyje jej jako punktu wejścia do programu. Jeśli naprawdę będzie nam na tym zależeć, to możemy poinstruować kompilator, by wykorzystał w tym celu inną metodę, choć znaczna część programów trzyma się tej konwencji. Niezależnie od tego, czy punkt wejścia do programu zostanie określony na mocy konwencji, czy też to jawnie wskażemy, metoda ta musi spełniać określone wymagania, które bardzo wyraźnie widać na listingu 1.9.

Punkt wejścia do programu musi być **metodą statyczną**, co oznacza, że w celu jej wywołania nie trzeba będzie tworzyć instancji klasy, w której metoda ta została zdefiniowana (w naszym przypadku jest to klasa `Program`). Metoda ta nie musi zwracać żadnych wyników, co wyraźnie sugeruje użycie słowa kluczowego `void`; choć może także zwracać wartość typu `int`, co pozwala przekazywać programom kod wyjściowy, który system operacyjny prezentuje po zakończeniu programu. (Metoda ta może także zwracać wartość typu `Tast` lub `Task<int>`, co daje możliwość przekształcenia jej w metodę asynchroniczną; zagadnienia związane z metodami asynchronicznymi zostały opisane w rozdziale 17.). Dodatkowo metoda ta nie może pobierać żadnych argumentów (co jest zaznaczone poprzez umieszczenie za jej nazwą pustej pary nawiasów) bądź może pobierać jeden argument — tablicę łańcuchów znaków zawierającą argumenty podane w wierszu wywołania programu (tak właśnie dzieje się w kodzie z listingu 1.9).

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`).

Za deklaracją metody umieszczone jest jej ciało. Początkowo metoda jest pusta. W ten sposób poznaliśmy już cały kod tego pliku wygenerowany przez Visual Studio. Nie pozostało nam zatem nic innego, jak dodać jakiś własny kod pomiędzy nawiasami klamrowymi wyznaczającymi ciało metody. Pamiętajmy, że nasz test nie powiódł się, ponieważ program nie spełnił zakładanego warunku: nie wyświetlił w oknie konsoli odpowiedniego komunikatu. Spełnienie tego warunku wymaga zmodyfikowania jednego wiersza kodu (przedstawionego na listingu 1.10).

Listing 1.10. Wyświetlanie komunikatu

```
Console.WriteLine("Witaj, świecie!");
```

Jeśli dodamy do programu powyższy wiersz kodu i ponownie wykonamy test jednostkowy, to w panelu *Eksplorator testów* obok naszego testu pojawi się znacznik, a poniżej komunikat informujący, że test zakończył się powodzeniem. A zatem wszystko wskazuje na to, że nasz kod działa. Możemy to potwierdzić, uruchamiając go. Możemy to zrobić za pomocą opcji dostępnych w menu *Debugowanie* (*Debug*) Visual Studio. Wybranie opcji *Rozpocznij debugowanie* (*Start Debugging*) spowoduje uruchomienie programu w debugerze. Jeśli uruchomimy program w taki sposób (co możemy także zrobić, naciskając kombinację klawiszy *Ctrl+F5*), przekonamy się, że wyświetla on w oknie konsoli tradycyjny komunikat.

Testy jednostkowe

Skoro nasz program już działa, chciałbym powrócić do pierwszego napisanego w tym rozdziale fragmentu kodu, czyli do testu jednostkowego, gdyż ilustruje on pewne cechy C#, których nie można przedstawić na przykładzie programu głównego. Jeśli ponownie spojrzymy na listing 1.1, zauważymy, że rozpoczyna się on podobnie jak program główny, czyli od grupy dyrektyw `using` oraz deklaracji przestrzeni nazw, której nazwa — `HelloWorld.Tests` — odpowiada nazwie projektu zawierającego test. Jednak sama klasa wygląda nieco inaczej. Zamieszczony poniżej listing 1.11 przedstawia interesujący nas aktualnie fragment kodu z listingu 1.1.

Listing 1.11. Klasa testu jednostkowego z atrybutem

```
[TestClass]
public class WhenProgramRuns
{
```

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 1.1 zostały przedstawione na listingu 1.12. Mechanizm wykonujący testy odnajdzie wszystkie metody oznaczone atrybutem `[TestInitialize]` i dla każdego testu zdefiniowanego w danej klasie jeden raz wywoła każdą z tych metod, przy czym nastąpi to przed wykonaniem samych tekstów. Jeśli natomiast chodzi o atrybut `[TestMethod]`, to jak się zapewne domyślasz, informuje on, które metody reprezentują testy.

Listing 1.12. Metody z atrybutami

```
[TestInitialize]
public void Initialize()
...

[TestMethod]
public void SaysHelloWorld()
...
```

Warto zwrócić uwagę na jeszcze jeden fragment kodu z listingu 1.1: zawartość klasy rozpoczyna się od pola, które przedstawiłem ponownie na listingu 1.13. Pola służą do przechowywania wartości. W tym przypadku metoda `Initialize` zapisuje w polu `_consoleOutput` przechwycone wyniki, które testowany program wyświetla w oknie konsoli. Dzięki temu nasz test może je następnie sprawdzić.

W naszym przykładzie pole zostało oznaczone jako `private`, co oznacza, że jest przeznaczone do wyłącznego użytku w danej klasie. Kompilator C# zapewni, że jedynie kod umieszczony w tej samej klasie będzie miał dostęp do tego pola.

Listing 1.13. Pole

```
private string _consoleOutput;
```

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 Visual Studio zawierające dwa projekty — projekt testowy oraz projekt samego programu. Przedstawiony w rozdziale przykład był bardzo prosty, dlatego każdy projekt składał się wyłącznie z jednego pliku źródłowego, którym mogliśmy się zainteresować. Oba miały podobną strukturę. Każdy z nich rozpoczynał się od grupy dyrektyw `using`, określających, jakie typy będą w nich używane. Deklaracje przestrzeni nazw określały, do jakich przestrzenia będzie należał kod umieszczony w plikach. Z kolei kod każdego z plików zawierał klasę oraz umieszczone w niej metody i inne składowe, takie 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.

Dla profesjonalistów najlepszy jest C#!

Język C# wciąż cechuje prostota, przy czym jego możliwości rosną z każdą wersją. Od początku jest rozwijany z konsekwencją, a każda nowa funkcjonalność idealnie integruje się z resztą języka. W efekcie C# jest dojrzały, nowoczesny, wszechstronny i bezpieczny. Stanowi integralną część platformy Microsoft .NET Framework. Profesjonalni programiści, którzy dbają o wysoką jakość tworzonego kodu, uważają C# i .NET za swoje ulubione narzędzie pracy. Wersja 8.0 tego języka sprawia, że programowanie staje się jeszcze bardziej efektywne i satysfakcjonujące. Pełne wykorzystanie tych imponujących możliwości wymaga jednak mistrzowskiego opanowania subtelności poszczególnych narzędzi i niuansów samego języka.

Ta książka została napisana z myślą o doświadczonych programistach. Podstawowe zagadnienia, takie jak klasy, polimorfizm i kolekcje, znalazły się w kilku pierwszych rozdziałach, jednak zrozumienie treści całej publikacji wymaga umiejętności technicznych. Została poświęcona ważnym koncepcjom C# i tajnikom tego języka, które rzadko kiedy są opisywane w literaturze. Dokładnie omówiono tu typy ogólne, LINQ oraz techniki programowania asynchronicznego. Przedstawiono najnowsze możliwości platformy .NET Core i języka C# 8.0, takie jak strumienie asynchroniczne, referencje akceptujące wartości puste, dopasowywanie wzorców, domyślne implementacje interfejsów, zakresy, a także nową składnię indeksowania oraz zmiany w narzędziach platformy .NET. Liczne rozbudowane przykłady stanowią świetne uzupełnienie prezentowanych treści.

W książce między innymi:

- możliwości języka C#: klasy, typy niestandardowe, kolekcje, obsługa błędów
- optymalizacja kodu pod kątem wykorzystania pamięci
- praca na strumieniach danych za pomocą technologii LINQ
- platforma .NET i programowanie wielowątkowe
- programowanie asynchroniczne a skalowalność aplikacji

Ian Griffiths jest uznanym autorytetem w dziedzinie C# i znakomitym nauczycielem tego języka. Pracuje w firmie konsultingowej endjin jako doradca do spraw technicznych. Jest współautorem kilku cenionych książek o programowaniu. Chętnie udziela się na kilku specjalistycznych forach programistycznych — jest znany z wyczerpujących, fachowych odpowiedzi. 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 978-83-283-6739-5
9 788328 367395
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 149,00 zł