Koduj jak profesjonalista C#

Jort Rodenburg

.NET/C#

Koduj jak profesjonalista C#

Jort Rodenburg

Znasz już podstawy, zatem przygotuj się na kolejny etap! Kod C# profesjonalnej jakości jest wydajny, czysty i szybki. Niezależnie od tego, czy budujesz aplikacje biznesowe dla użytkowników, czy usługi backendu dla aplikacji Web, oparte na doświadczeniu praktyczne techniki przedstawione w tej książce pozwalają podnieść umiejętności C# na nowy poziom.

Koduj jak profesjonalnista w C# pokazuje, jak pisać czysty kod C#, właściwy dla aplikacji klasy przedsiębiorstwa. Wraz z autorem przejdziesz proces refaktoryzacji odziedziczonej bazy kodu, stosując techniki nowoczesnego C#. Po drodze poznasz takie narzędzia, jak Entity Framework Core, techniki projektowe, takie jak wstrzykiwanie zależności oraz kluczowe praktyki wytwarzania sterowanego testami. Jest to doskonały sposób na zwiększenie już posiadanych umiejętności lub przejścia z innego języka obiektowego do C# i ekosystemu .NET.

Podstawowe zagadnienia:

- Testy jednostkowe i wytwarzanie sterowane testami
- Refaktoryzacja odziedziczonego kodu .NET
- Zasady czystego kodu
- Odpytywanie i manipulowanie bazami danych przy użyciu LINQ i Entity Framework Core

Książka przeznaczona jest dla programistów z doświadczeniem w programowaniu obiektowym. Znajomość C# nie jest wymagana.

Jort Rodenburg jest inżynierem oprogramowania z bogatym doświadczeniem praktycznym, a także wykładowcą i autorem kursów na temat przyśpieszania i usprawniania pracy w C# i .NET.

„Warto przeczytać kilka razy”.
Prabhuti Prakash
Synechron Technologies

„Pełna wskazówek i spostrzeżeń pozwalających szybko wejść w rytm. Zdecydowanie polecam!”
Edin Kapic, isolutions

„Ta książka naprawdę pomogła mi przejść na następny poziom”.
Daniel Vásquez Estupiñan
Tokiota

„Takiej właśnie książki szukałem, gdy chciałem nauczyć się pisania lepszego kodu w C#”.
Gustavo Filipe Ramos Gomes
Troido

„Uczy świetnych technik i najlpeszych praktyk nowoczesnego programowania w C#”.
Foster Haines, J2 Interactive

ISBN 978-83-7541-461-5

Cena z VAT 89,99 zł

CENTRUM EDUKACYJNE
PROMISE
GRUPA APN PROMISE

MANNING

Koduj jak profesjonalista C#

Jort Rodenburg

przełożył
Marek Włodarz

APN Promise
Warszawa 2021

Koduj jak profesjonalista C#

Authorized Polish translation of the English language edition entitled:
Code Like a Pro in C#, by Jort Rodenburg, published by Manning Publications, Co., ISBN 9781617298028.

APN PROMISE SA, ul. Domaniewska 44a, 02-672 Warszawa, tel. +48 22 35 51 600, fax +48 22 35 51 699, e-mail: mspress@promise.pl

ISBN: 978-83-7541-461-5 (druk), 978-83-7541-459-2 (ebook)

Projekt graficzny okładki: Marija Tudor

Przekład: Marek Włodarz

Korekta: Ewa Swędrowska

Skład i łamanie: MAWart Marek Włodarz

Spis treści

przedmowa

Moje pierwsze spotkanie z C# miało miejsce w roku 2016, gdy dołączyłem do zespołu Fujifilm Medical Systems. Miałem wcześniejsze doświadczenia w programowaniu w Javie i Pythonie, ale gdy odkryłem C#, nie oglądałem się wstecz. Spodobała mi się łatwość, z jaką można w nim tworzyć od razu działające moduły oraz (początkowo nieznośnie irytującą) koncentrację na jawnym określaniu typów. W czasie mojej pracy w tej firmie zamęczałem kolegów pytaniami o C# i jak go najlepiej używać. Początki były łatwe, ale osiągnięcie biegłości to inna sprawa. Każdy może napisać aplikację „Hello, World" w 10 minut, bez względu na swoje przygotowanie, ale wykorzystanie całej mocy języka z jednoczesną świadomością, dlaczego określone rzeczy są implementowane właśnie tak, a nie inaczej, po prostu wymaga czasu. Po pewnym czasie poczułem, że przestałem się rozwijać, jeśli chodzi o moją znajomość C# i zacząłem poszukiwać źródeł, które pozwoliłyby mi wejść na wyższy poziom. Szybko zrozumiałem, że istnieją trzy główne rodzaje książek zajmujących się .NET oraz C#: książki o tematyce wykraczającej poza język (czysty kod, architektura, infrastruktura i temu podobne), których autorzy przypadkiem używali C#, następnie książki o tym, jak zacząć programować w C#, oraz książki tak zaawansowane, że po ich przeczytaniu będziemy w stanie zostać dyrektorem do spraw technicznych w Microsofcie. Ja jednak chciałem znaleźć książkę, która znalazłaby się pośrodku tych trzech kategorii: książki, która zajmuje się czystym kodem i tworzy pomost pomiędzy zagadnieniami początkowymi a zaawansowanymi. Taka książka jednak nie istniała, zatem ją napisałem. To właśnie ta książka.

Jeśli Czytelnik jest inżynierem oprogramowania (albo programistą, koderem czy jakkolwiek określa swoją pozycję) z wcześniejszymi doświadczeniami w programowaniu (preferowane są języki obiektowe), który chciałby wejść w świat C#, ta książka jest dla niego. Nie zamierzam tu pokazywać, jak napisać instrukcję `if`, ani nie będę wyjaśniać, czym jest obiekt. Tym, co Czytelnik znajdzie w tej książce, to umiejętności i zagadnienia, które przygotowują do pogłębionych studiów nad językiem i platformą .NET. Oczywiście nie mogę obiecać, że omówię wszystkie trudniejsze zagadnienia, ale naprawdę się starałem, w ramach limitowanej objętości książki. Mam szczerą nadzieję, że lektura tej książki sprawi przyjemność i że w pamięci pozostanie po niej jedna lub kilka nowych, wartościowych rzeczy. Nie twierdzę, że ta książka jest arcydziełem, ale mam nadzieję, że okaże się przynajmniej użyteczna. A jeśli nawet nie, no cóż, nigdy nie zaszkodzi, gdy ponownie odświeżymy sobie już posiadaną wiedzę.

podziękowania

Gdy po raz pierwszy rozmawiałem z wydawnictwem Manning o napisaniu tej książki, miałem niewielkie pojęcie, że pochłonie mi ona cały rok (a nawet nieco więcej). Aby być uczciwym, byłem ostrzegany wielokrotnie, że autorzy mają skłonność niedoszacowywania czasu niezbędnego do napisania książki. Ja, będąc upartym, sądziłem, że będę wyjątkiem od tej reguły. Nie byłem. Od grudnia 2019 do marca 2021 poświęciłem tej książce wiele godzin. W wielu momentach byłem przekonany, że „to już na pewno będzie koniec". Za każdym razem (poza jednym, co oczywiste) tak się nie działo. Szczęśliwie mam bardzo cierpliwą żonę i mnóstwo czasu do zabicia.

Mając to na uwadze, chciałbym przede wszystkim podziękować właśnie jej za trwanie przy mnie w całej tej podróży oraz przeprosić ją za to, że na przeszło rok niemal zniknąłem z jej życia. Nie zdołałbym napisać tej książki bez jej niezachwianego wsparcia. To ona jest kamieniem węgielnym, na którym zbudowałem tę książkę. Chciałbym też podziękować mojej rodzinie, która zawsze z ciekawością słuchała o nowych pomysłach i uzupełnieniach. Pozwoliłem sobie nadać prezesowi fikcyjnej firmy przedstawionej w książce imię mojego dziadka ze strony matki (Aljen) i nazwisko babci ze strony ojca (van der Meulen).

Chcę także podziękować wspaniałemu zespołowi z wydawnictwa Manning. W szczególności podziękowania należą się Marinie Michaels. Jako mój redaktor, przekształciła tę książkę w coś więcej, niż losowa kolekcja niespójnych przemyśleń. Dzięki Marinie wypracowałem zdrowy lęk przed używaniem słowa *będzie* w swojej pisaninie. Miałem również bardzo cenny zespół pomocników: to Jean-François Morin, Tanya Wilke, Eric Lippert, Rich Ward, Enrico Buonanno oraz Katie Tennant. Ten międzynarodowy (a właściwie międzykontynentalny) zespół superbohaterów/ninjów/gwiazd rocka zaowocował świetnymi uwagami i wyłapał ogromną liczbę (często bardzo zawstydzających) błędów technicznych. Chciałbym również podziękować wszystkim czytelnikom wersji wstępnych, którzy poświęcili czas na czytanie rękopisu przed publikacją i wysunęli mnóstwo przydatnych uwag.

Lista recenzentów, którym należą się podziękowania, jest dość długa (w kolejności alfabetycznej): Arnaud Bailly, Christian Thoudahl, Daniel Vásquez Estupiñan, Edin Kapic, Foster Haines, George Thomas, Goetz Heller, Gustavo Filipe Ramos Gomes, Hilde Van Gysel, Jared Duncan, Jason Hales, Jean-François Morin, Jeff Neumann, Karthikeyarajan Rajendran, Luis Moux, Marc Roulleau, Mario Solomou, Noah Betzen, Oliver Korten, Patrick Regan, Prabhuti Prakash, Raymond Cheung, Reza Zeinali, Richard B. Ward, Richard DeHoff, Sau Fai Fong, Slavomir Furman, Tanya Wilke, Thomas F. Gueth, Víctor M. Pérez oraz Viktor Bek. Wasze uwagi i sugestie pomogły mi sprawić, że ta książka jest lepsza.

Na koniec jest kilka osób, którym chcę podziękować za to, że pomogli mi przy jakimś fragmencie tej książki albo w ogóle w moim rozwoju. Na początek David Lavielle i Duncan Henderson: dziękuję za danie mi szansy i pierwszą pracę przy tworzeniu oprogramowania. Jerry Finegan: za wprowadzenie mnie do świata C# i cierpliwe wysłuchiwanie głupich pytań,

jednego po drugim. Twoja cierpliwość i pomoc są nieocenione. Michael Breecher: za udział w ukształtowaniu fragmentu o kongruencji (wymuszone nocnymi pytaniami o notację matematyczną), sądzę, że książka jest lepsza dzięki temu. Szymon Zuberek: pierwszy szkic rozdziału 2 napisałem w twoim nowojorskim mieszkaniu. Dzięki za miejsce na kanapie za każdym razem, gdy chcieliśmy wpaść do Miasta, a także za jakże pasjonujące rozmowy. No i dziękuję wspaniałym ludziom z Acronis i Workiva, którzy musieli wysłuchiwać mojego paplania o „tej książce, którą piszę". Byliście bardzo cierpliwi (zazwyczaj).

o tej książce

Książka ta ma na celu rozwinięcie istniejących umiejętności programistycznych, aby pozwolić czytelnikowi gładko usprawnić swoją praktykę kodowania albo przejść do programowania w C# z Javy lub innego języka obiektowego. Nauczymy się, jak pisać idiomatyczny kod C# – coś, co jest kluczowe w tworzeniu oprogramowania dla przedsiębiorstw. Omawiam tu niezbędne umiejętności backendowe i pokazuję ich stosowanie w praktyce poprzez typowe wyzwanie, z jakim każdy programista musi się zmierzyć prędzej albo później: refaktoryzację odziedziczonej bazy kodu, aby sprawić, że będzie on bezpieczny, czysty i czytelny. Gdy dojdziemy do końca, czytelnik uzyska profesjonalny poziom rozumienia C# i będzie gotowy na specjalizowanie się w zagadnieniach poziomu zaawansowanego.

Nie znajdziemy tu „Hello, World" ani tematów z lekcji informatyki – będziemy się uczyć, refaktoryzując przestarzały, odziedziczony kod przy użyciu nowych technik, narzędzi i najlepszych praktyk, aby podnieść go do poziomu standardów nowoczesnego C#. W tej książce weźmiemy istniejącą bazę kodu (napisaną w .NET Framework) i zrefaktoryzujemy ją, wraz z uproszczonym API, do standardu .NET 5.

Kto powinien przeczytać tę książkę

Dla kogoś, kto jest sprawnym programistą posługującym się językiem obiektowym, niech to będzie Java, Dart, C++ lub cokolwiek innego, książka ta może być pomocą w szybkim przejściu do C# i .NET bez konieczności zaczynania od początku. Znacząca część posiadanej wiedzy zachowuje ważność, zatem po co miałbym kogoś uczyć po raz pięćsetny, jak napisać instrukcję `if`?

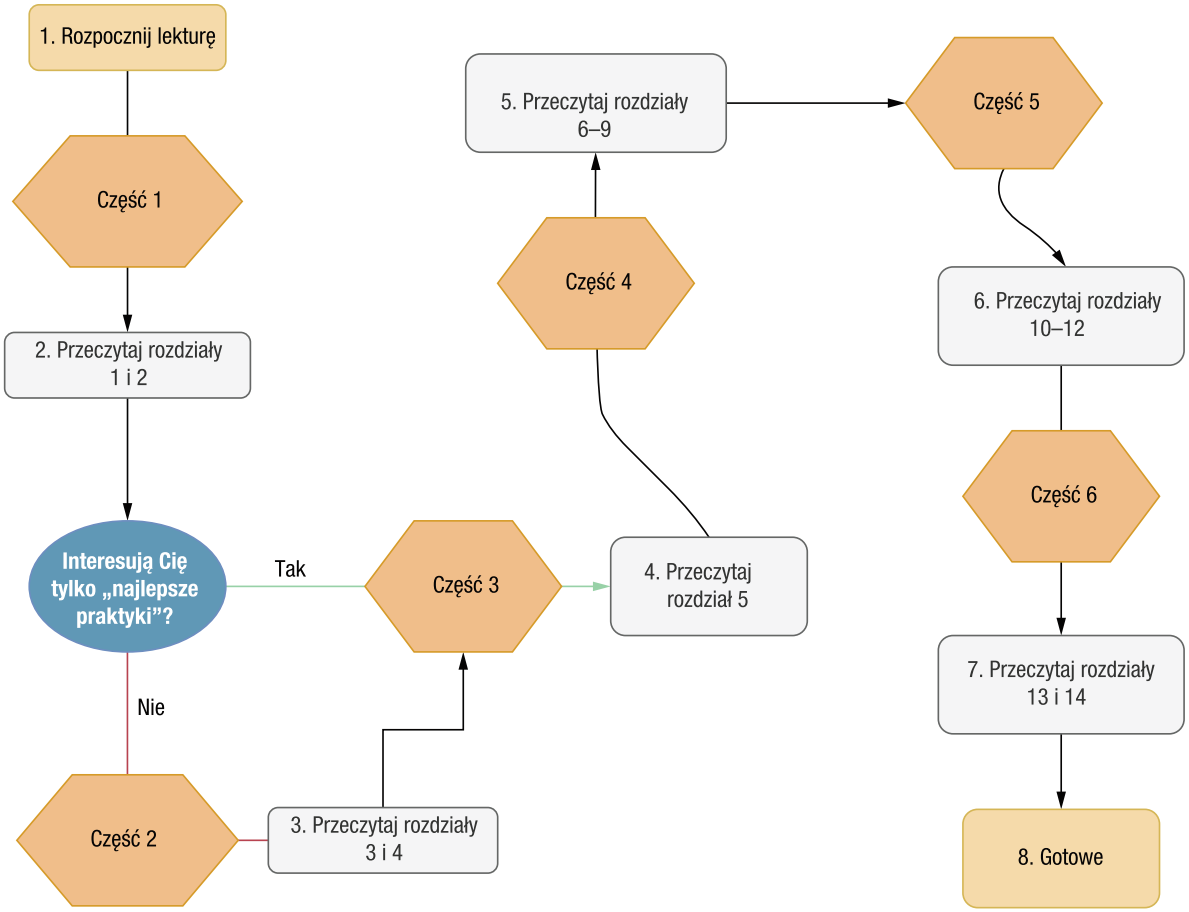
Analogicznie, jeśli ktoś ma doświadczenie w języku programowania, takim jak Go, C, JavaScript, Python lub dowolny inny język głównego nurtu, po przeczytaniu tej książki będzie umiał pisać czysty, idiomatyczny C#. Zapewne będzie trzeba poczytać trochę o różnych zasadach programowania obiektowego, ale nie powinno to stanowić istotnej przeszkody (OK, jeśli ktoś przychodzi z języka Go, powinien zwrócić więcej uwagi na to, jak i kiedy używamy interfejsów; nie działają one tak samo).

Na koniec, jeśli ktoś jest deweloperem, który już od jakiegoś czasu używa C# i szuka sposobu „podniesienia" swojej wiedzy na wyższy poziom: ta książka jest właśnie dla niego. Wiele zaawansowanych źródeł dotyczących C# zakłada posiadanie wiedzy i umiejętności, które nie są omawiane w podręcznikach dla początkujących. Ta książka próbuje załatać tę lukę.

Organizacja książki: mapa drogowa

Książka ta wykazuje trochę niekonwencjonalne podejście do struktury, jeśli porównamy ją z typowymi książkami technicznymi. Większość takich książek do pozycje źródłowe (referencyjne), które można czytać w dowolnej kolejności. Ta książka nie jest jednak źródłem referencyjnym i konieczne jest przeczytanie rozdziałów po kolei, aby wydobyć z niej jak najwięcej. Podzieliłem książkę na sześć części, zgodnie ze schematem pokazanym na rysunku 1:

1. „Używanie C# i .NET" – w rozdziale 1 wyjaśnię, o czym jest ta książka, czego ma nauczyć, a także czego nie zamierza uczyć. Rozdział 2 to krótka wycieczka po języku C# i ekosystemie .NET, skupiona na tym, co odróżnia .NET od innych platform oraz na przebiegu kompilacji kodu C#.
2. „Istniejąca baza kodu" – w tej części zaproszę czytelników do eksploracji odziedziczonej bazy kodu. Jest to szczegółowa analiza istniejącej bazy kodu, wraz z omówieniem potencjalnych ulepszeń i istniejących wad projektowych.
3. „Warstwa dostępu do bazy danych" – Po części drugiej rozpoczniemy faktyczne pisanie całej usługi od początku. W części 3 skupimy się na tworzeniu nowego projektu .NET Core i dowiemy się, jak możemy użyć Entity Framework Core do połączenia się z chmurową (lub lokalną) bazą danych. Inne omawiane zagadnienia obejmują wzorzec repozytorium/usługa, metody i właściwości wirtualne oraz klasy zapieczętowane.
4. „Warstwa repozytorium" – W części 4 wkroczymy w świat wzorca repozytorium/usługa i zaimplementujemy pięć klas repozytorium. Poznamy tu również zagadnienia wstrzykiwania zależności, wielowątkowość (w tym blokady, muteksy i semafory), niestandardowe porównania równościowe, wytwarzanie sterowane testami, typy ogólne, metody rozszerzające oraz LINQ.
5. „Warstwa usługi" – Kolejny krok to implementacja klas warstwy usługi. W części 5 od podstaw napiszemy cztery klasy usługowe i porozmawiamy o refleksji, imitacjach, sprzężeniach, asercjach i sprawdzaniu typów w czasie działania, obsłudze błędów, strukturach i `yield return`.
6. „Warstwa kontrolera" – Część 6 to finalny krok w naszej refaktoryzacji usługi, którą odziedziczyliśmy w części 2. W tej części napiszemy dwie klasy kontrolerów i wykonamy testy akceptacyjne. Oprócz tych tematów, poruszymy też zagadnienia oprogramowania pośredniego (middleware) ASP.NET Core, routingu HTTP, niestandardowego wiązania danych, serializacji i deserializacji oraz generowania specyfikacji OpenAPI podczas działania programu.



Rysunek 1 Schemat blokowy sugerowanych dróg podczas czytania książki. Schemat ten wynika z inspiracji podobnymi wykresami struktury zamieszczanym w książkach z serii The Art of Computer Programming Donalda Knutha.

Wiele rozdziałów tej książki (a niekiedy podrozdziałów) kończy się ćwiczeniami, które mają na celu sprawdzenie opanowania materiału przez Czytelnika. Zachęcam do próby ich rozwiązywania w miarę napotykania w lekturze i powrotu do odpowiednich części, jeśli okaże się, że coś zostało pominięte albo źle zrozumiane.

Kod dołączony do książki

W chwili pisania tych słów krajobraz .NET można było podzielić na trzy główne części: .NET Framework 4.x, .NET Core 3.x oraz .NET 5. W całej książce używamy .NET 5, z wyjątkiem rozdziałów 3 oraz 4 (z powodów, które staną się zrozumiałe po przeczytaniu tych rozdziałów).

Wersje języka C# używane w kodzie to C# 3 oraz C# 9 (choć nie używamy żadnej funkcji specyficznej dla C# 9, zatem równie dobrze sprawdzi się instalacja C# 8). Jako że poszczególne wersje języka C# zapewniają wsteczną zgodność, trzeba zainstalować tylko najnowszą wersję (w czasie pisania tych słów jest to albo C# 8, albo wstępna wersja C# 9). Rozdziały, do których dostarczony jest kod źródłowy, to 2, 3 i 4 (łącznie), 5, 6, 7, 8, 9, 10, 11, 12, 13 oraz 14.

Aby móc uruchomić kod, trzeba zainstalować wersję .NET Framework wyższą niż 3.5 (jeśli ktoś chce uruchamiać kod z rozdziałów 3 i 4) oraz .NET 5. Jeśli ktoś chce lokalnie uruchomić bazę danych używaną w książce albo ma trudności z instalowaniem potrzebnych komponentów, instrukcje instalacyjne zamieszczone są w dodatku C („Wskazówki instalacyjne"). W książce używam głównie Visual Studio jako IDE, ale można użyć dowolnego IDE wspierającego C# (albo w ogóle żadnego), jeśli ktoś ma ochotę. Visual Studio 2019 ma bezpłatną wersję o nazwie Visual Studio 2019 Community. Gdy wystąpią rzeczy, które wymagają użycia Visual Studio, zostanie to odnotowane w treści książki. Kod oraz .NET 5 powinny działać poprawnie (i tak samo) w systemach Windows, macOS i Linux. W książce używam wiersza poleceń (lub terminala w przypadku macOS i Linuksa), ilekroć jest to możliwe, aby unikać uzależnienia od jakiegoś konkretnego IDE lub systemu operacyjnego.

Książka zawiera wiele przykładów kodu źródłowego, zarówno w postaci numerowanych listingów, jak i włączonych do normalnego tekstu. W obu przypadkach kod jest formatowany przy użyciu `fontu stałopozycyjnego`, jak tutaj, aby odróżnić go od zwykłego tekstu. Niekiedy kod jest również **`wytłuszczony`**, aby wyróżnić fragmenty zmienione od poprzednich kroków w tym samym rozdziale, jak w przypadku, gdy nowa funkcja jest dodawana do istniejących wierszy kodu.

W wielu przypadkach oryginalny kod źródłowy został przeformatowany; dodane zostały podziały wierszy i zmienione wcięcia, aby zmieścić zbyt długie wiersze na ograniczonej przestrzeni strony. W niektórych przypadkach nawet to nie wystarczyło i listingi zawierają wówczas znaczniki kontynuacji wiersza (➥). Wiele listingów opatrzonych jest adnotacjami wyjaśniającymi i wyróżniającymi ważne koncepcje. Można też zauważyć, że nawiasy klamrowe są typowo umieszczane w wierszach poprzedzających nowe bloki kodu. To nie jest właściwa konwencja C# stosowana w praktyce, ale użyliśmy jej w celu zaoszczędzenia miejsca. Sam kod źródłowy nie używa tej konwencji.

o autorze

Jort Rodenburg jest inżynierem oprogramowania, autorem i wykładowcą. Specjalizuje się w C# i pracował przy oprogramowaniu dla wielu obszarów zastosowań, takich jak zgodność finansowa i raportowanie, obsługa drukarek atramentowych, obrazowanie medyczne, systemy rozproszone i cyberbezpieczeństwo. Jort kształci inżynierów biegłych w innych językach programowania w przechodzeniu do C# i .NET. Prowadzi również wykłady i prezentacje na konferencjach i meetupach poświęconych C#, .NET i programowaniu.

o ilustracji na okładce

Obraz zamieszczony na okładce *Koduj jak profesjonalista C#* jest podpisany „Homme Samojede", czyli „Mężczyzna samojedzki". Ilustracja pochodzi z kolekcji strojów z różnych krajów autorstwa Jacques'a Grasset de Saint-Sauveur (1757–1810), zatytułowanej *Costumes de Différents Pays*, opublikowanej we Francji w roku 1797. Każda ilustracja jest starannie narysowana i ręcznie kolorowana. Wielka różnorodność kolekcji Grasset de Saint-Sauveura przypomina nam żywo, jak bardzo zróżnicowane kulturowo były regiony świata zaledwie 200 lat temu. Odizolowani od siebie, ludzie posługiwali się różnymi dialektami i językami. Czy na ulicach miast, czy w obszarach wiejskich łatwo można było rozpoznać, gdzie żyją i jaka jest rola lub pozycja w życiu, jedynie na podstawie strojów.

To, jak się ubieramy, zmieniło się od tamtych czasów i zróżnicowanie regionalne, tak bogate jeszcze niedawno, stopniowo zanikło. Dziś trudno byłoby rozróżnić mieszkańców różnych kontynentów, a co dopiero różnych miast, regionów czy krajów. Być może zamieniliśmy zróżnicowanie kulturowe na bardziej urozmaicone życie osobiste w ciągle przyśpieszającym świecie.

W czasach, gdy trudno jest odróżnić jedną książkę informatyczną od innej, Manning celebruje pomysłowość i inicjatywę w branży komputerowej okładkami książek opartymi na bogactwie różnorodności życia regionalnego sprzed dwóch stuleci, przywróconej do życia dzięki obrazom Grasset de Saint-Sauveura.

Część 1

Używanie C# i .NET

W pierwszej części tej książki odbędziemy krótką wycieczkę po języku C# i porozmawiamy o niektórych jego cechach. Rozdział 1 wyjaśnia, czym są C# oraz .NET i dlaczego mielibyśmy ich używać (albo nie) w swoich projektach. Rozdział 2 wchodzi głębiej w rozmaite iteracje .NET oraz omawia proces kompilacji metod języka C#, zatrzymując się przy każdym istotnym kroku.

Choć część ta jest w istocie wprowadzeniem do książki, nadal zapewnia nieocenione informacje również dla kogoś, kto dobrze zna C#. Część wiedzy przedstawianej w tych pierwszych rozdziałach jest tym, co trzeba poznać, zanim przejdziemy do bardziej zaawansowanych tematów.

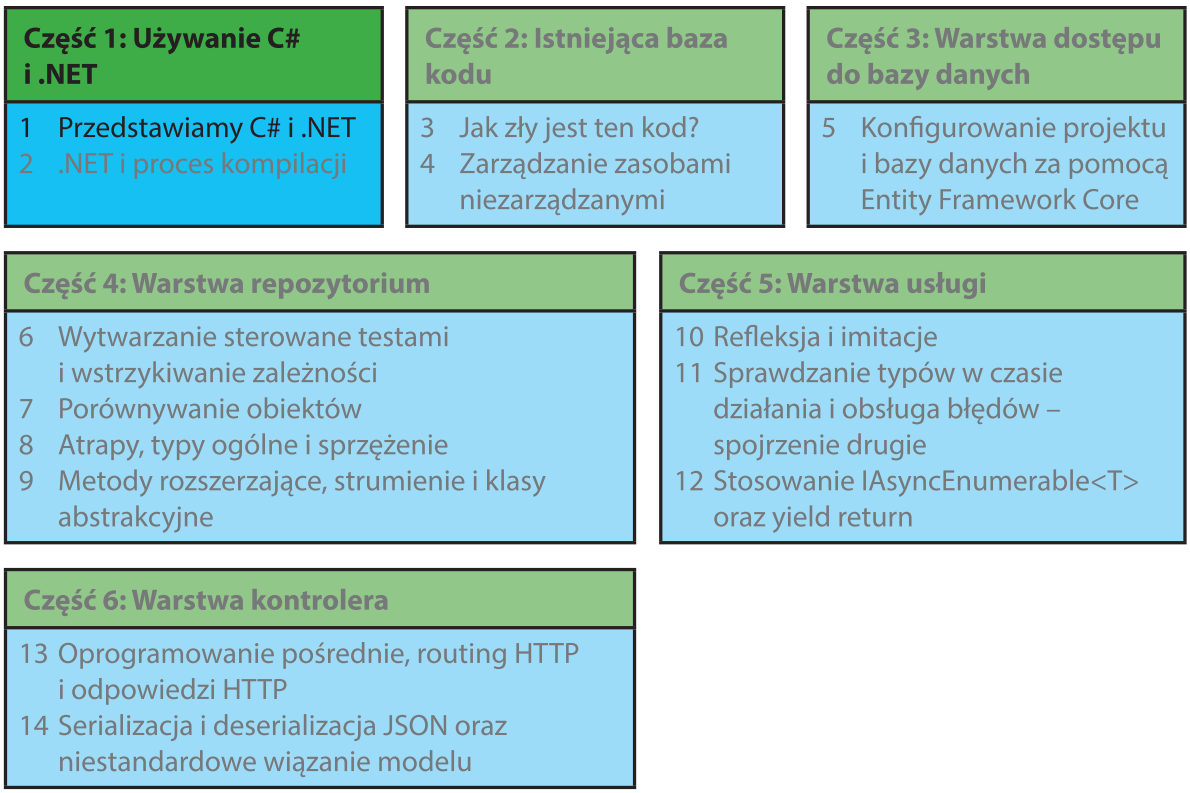
1 Przedstawiamy C# i .NET

W tym rozdziale

- Czym są C# i .NET
- Dlaczego powinniśmy używać C# w swoich projektach (i dlaczego nie powinniśmy)
- Przełączanie się na C# i jak rozpocząć

O nie, kolejna książka o C#? Tak, jeszcze jedna. Mnóstwo książek powstało wokół tematyki C# i .NET, ale tę cechuje jedna fundamentalna różnica: napisałem ją, aby pomóc w tworzeniu czystego, idiomatycznego kodu C# w swojej codziennej pracy. Nie jest to podręcznik źródłowy, ale raczej praktyczny przewodnik. Nie wyjaśniam w niej takich rzeczy, jak napisać instrukcję `if`, czym jest sygnatura metody ani czym są obiekty. Nie zajmujemy się tu składnią – zakładam, że ten poziom opanowania języka Czytelnik ma już za sobą – ale skupiamy się na koncepcjach i ideach. Istnieje znacząca różnica pomiędzy znajomością składni języka a zdolnością do pisania przejrzystego, idiomatycznego kodu. Po przejściu całej książki To właśnie powinien wynieść Czytelnik z lektury tej książki. Bez względu, jakie Czytelnik ma przygotowanie i jakie zna języki programowania, o ile rozumie koncepcje programowania obiektowego, książka ta pozwoli mu skutecznie przejść do ekosystemu C# i .NET, co pokazuje rysunek 1.1.

Co wspólnego ze sobą mają takie organizacje, jak Microsoft, Google i rząd Stanów Zjednoczonych? Wszystkie one używają C# – i mają do tego dobry powód. Ale dlaczego?



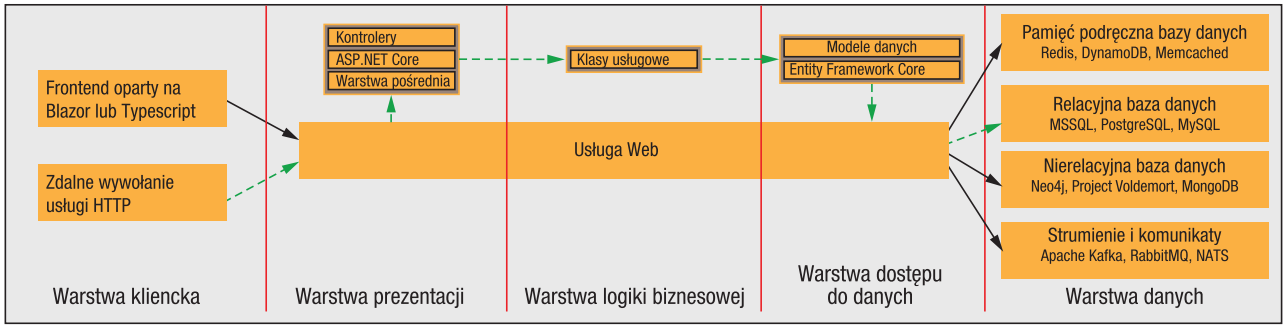
Rysunek 1.1 Każdy rozdział zawiera diagram postępów, który pozwala szybko ustalić miejsce, w którym się znajdujemy.

C# to przecież tylko „kolejny język" programowania. Wykazuje podobieństwo do Javy i C++, pozwalając zarówno na programowanie obiektowe, jak i funkcjonalne, a także korzysta z wielkiego wsparcia ogromnej społeczności open source. Wspaniale. Ale dlaczego mielibyśmy się tym przejmować? W tym rozdziale szczegółowo zajmę się tym zagadnieniem, ale na początek kilka spoilerów: C# jest doskonały, jeśli chodzi o tworzenie skalowalnego oprogramowania. Wszystko, czego potrzebujemy, aby zacząć pisać kod C#, to dowolnie wybrane .NET SDK (więcej na ten temat w rozdziale 2) i – zapewne – jakieś IDE. Zarówno język, jak i runtime (środowisko wykonawcze) należą do domeny open source.

Za każdym razem, gdy szukamy online jakiś informacji o C#, niemal na pewno natrafimy na termin .NET Framework. O .NET Framework można myśleć jak o ciepłym kocyku, trzaskającym ogniu w kominku i kubku gorącej czekolady w zimowy dzień. Dostarcza wszystkiego, czego potrzebujemy: bibliotek opakowujących niskopoziomowe API Windows, eksponuje powszechnie używane struktury danych i zapewnia wrappery dla złożonych algorytmów. Codzienne tworzenie oprogramowania w C# niemal na pewno będzie wiązało się z używaniem .NET Framework, .NET Core albo .NET 5, zatem zbadamy te frameworki i to, gdzie są odpowiednie.

Rysunek 1.2 pokazuje, jak tematyka tej książki wpasowuje się w ogólną architekturę aplikacji webowej .NET. Przedstawia również architekturę, której użyjemy do całkowitego przekonstruowania (przepisania) istniejącej aplikacji, co rozpoczniemy w rozdziale 5 (zielone/przerywane strzałki wskazują tę ścieżkę).

Typowa architektura Web przy użyciu stosu Microsoft



Rysunek 1.2 Przykład architektury typowej usługi webowej opartej na stosie Microsoft. W książce tej będziemy realizować podejście pokazane zielonymi/przerywanymi strzałkami. Omówimy warstwy prezentacji, logiki biznesowej oraz dostępu do danych.

Dla tych Czytelników, którzy mają już wcześniejsze doświadczenia w C#: książkę tę można spozycjonować pomiędzy poziomem początkowym i zaawansowanym. Umiejętności, których można się z niej nauczyć, wypełniają lukę w wiedzy i zapewniają podstawy dla zaawansowanych zadań. Pierwsze dwa rozdziały mogą się wydawać powtórką ze znanych już zagadnień, ale zachęcam, aby ich nie pomijać. Zawsze warto odświeżyć sobie posiadaną wiedzę.

1.1 *Dlaczego warto pracować w C#?*

Ktoś, kto już zna jakiś język programowania inny niż C# i lubi się nim posługiwać, może zapytać, dlaczego miałby w ogóle zacząć używać C#? Dobrym powodem może być zatrudnienie się w firmie, która używa tylko C#. A być może ktoś chciałby po prostu dowiedzieć się, o co w tym wszystkim chodzi.

Obiecuję, że nie będę w kółko powtarzać, że C# jest „silnie typowanym obiektowym językiem programowania, który umożliwia międzyplatformowe wytwarzanie skalowalnego oprogramowania klasy przedsiębiorstwa". Czytelnik zapewne już jest tego świadomy i analizowanie tego stwierdzenia zapewne nie jest najbardziej ekscytującym zadaniem. W tym podrozdziale omówię wszystkie modne słowa zawarte w tej definicji tylko raz – i nie będziemy już do tego wracać. Podejmę ryzyko, że zabrzmi to tak, jakbym był pracownikiem działu marketingu Microsoftu, w pozostałej części tego podrozdziału skupię się na następujących wyróżniających cechach i przypadkach użycia języka C#:

- C# (oraz ekosystem .NET) umożliwia wytwarzanie oprogramowania w sposób ekonomiczny. Efektywne rozwiązania są ważne, gdyż wytwarzanie na potrzeby przedsiębiorstw jest chlebem powszednim pracy w C#.
- C# pozwala ulepszyć stabilność i utrzymywanie kodu dzięki wsparciu dla samo-dokumentowanego kodu, bezpiecznym bibliotekom i łatwości użycia.
- C# jest przyjazny dla dewelopera i łatwy w użyciu. Nie ma nic gorszego, niż odkrycie, że język programowania, którego chcemy użyć, nie zapewnia dobrego wsparcia dla

rzeczy, które naprawdę lubimy (takich jak stabilny menedżer pakietów, dobre wsparcie dla testów jednostkowych i międzyplatformowe środowisko wykonawcze).

Oczywiście pisanie skalowalnego, dającego się utrzymywać i przyjaznego dla deweloperów „czystego kodu" można zrealizować w większości (jeśli nie w każdym) języków programowania. Różnica tkwi w komforcie pracy dewelopera. Niektóre języki są naprawdę dobre w pomaganiu tworzenia czystego kodu, podczas gdy inne takie nie są. C# nie jest doskonały, ale naprawdę stara się pomóc nam w tym względzie.

1.1.1 *Powód 1: C# jest ekonomiczny*

C# jest darmowy – zarówno w tworzeniu, jak i w stosowaniu. Język i platforma w całości należą do świata open source, cała dokumentacja jest bezpłatna, a większość narzędzi oferuje darmowe opcje. Dla przykładu, typowa konfiguracja C# obejmuje instalację C# 8, .NET 5 oraz Visual Studio Community. Wszystkie te komponenty są bezpłatne i będziemy używać ich w tej książce. Środowisko wykonawcze nie wymaga żadnych opłat licencyjnych, a produkt końcowy można wdrożyć w dowolnie wybranym miejscu.

1.1.2 *Powód 2: C# jest łatwy w utrzymaniu*

Ilekroć w tej książce piszę o utrzymywaniu kodu, mam na myśli zdolność do poprawiania błędów, zmian funkcjonalności i rozwiązywania innych problemów bez niezamierzonych efektów ubocznych. Może się to wydawać oczywistym wymaganiem dla dowolnego języka programowania, ale w rzeczywistości jest bardzo trudne do zaimplementowania. C# zawiera funkcjonalności, które poprawiają utrzymywanie (a dzięki temu bezpieczne rozszerzanie) obszernych baz kodu. Dla przykładu rozważmy typy ogólne oraz Language-Integrated Query (LINQ). Obie te rzeczy omówimy szczegółowo w dalszej części książki, ale są to przykłady funkcjonalności platformy, które pomagają w pisaniu lepszego kodu.

Pozornie, z punktu widzenia firmy zdolność utrzymywania kodu nie musi być priorytetem numer jeden, ale jeśli tworzymy kod, który jest utrzymywalny (przez co rozumiem czysty kod, który jest łatwo rozszerzalny i oparty na testach), koszty wytwarzania znacząco maleją. Spadek kosztów wytwarzania dzięki pisaniu utrzymywalnego kodu może się początkowo wydawać czymś sprzecznym z intuicją: zaprojektowanie i napisanie takiego kodu wymaga więcej czasu, co zwiększa początkowe koszty wytwarzania. Wystarczy jednak wyobrazić sobie, co nastąpi nieco później, gdy użytkownicy odkryją błąd albo zapragną dodatkowej funkcjonalności. Jeśli napisany przez nas kod jest utrzymywalny, możemy szybko i łatwo znaleźć ten błąd (i go naprawić). Dodanie nowej funkcji jest prostsze, gdyż baza kodu jest rozszerzalna. Jeśli możemy łatwo rozszerzać i poprawiać bazę kodu, koszty wytwarzania zmaleją.

Zasada otwarte/zamknięte (Open/Closed Principle)

W roku 1988 francuski informatyk Bertrand Meyer (twórca języka programowania Eiffel) opublikował książkę zatytułowaną *Object-Oriented Software Construction* (Prentice Hall, 1988). Wydanie tej książki było punktem zwrotnym w historii obiektowego programowania i projektowania, gdyż wprowadza ona *zasadę otwarte/zamknięte* (Open/Closed Principle – OCP). Zasada OCP skupia się na usprawnianiu utrzymywalności i elastyczności projektów programistycznych. W ujęciu Meyera OCP oznacza, że „byty programistyczne (klasy, moduły, funkcje itp.) powinny być otwarte na rozszerzanie, ale zamknięte na modyfikacje".

Co jednak OCP oznacza w praktyce? Aby to zbadać, zastosujmy OCP do klasy: postrzegamy klasę jako „otwartą" na rozszerzenie i „zamkniętą" na modyfikację, jeśli możemy dodać do niej jakąś funkcjonalność bez zmieniania już istniejącej (a dzięki temu bez psucia innych części naszego kodu). Jeśli będziemy trzymać się tej reguły, ryzyko wprowadzenia zakłóceń (albo nowego błędu) w istniejącym kodzie jest znacznie mniejsze, niż gdybyśmy próbowali wymusić poprawkę albo nową funkcję bez zwracania uwagi na utrzymywanie i rozszerzalność kodu. Gdy pracujemy z kodem, ktory jest bardziej złożony (i sprzężony; omówimy to w rozdziale 8), rośnie prawdopodobieństwo wprowadzania nowych błędów z powodu niezrozumienia efektów ubocznych naszych zmian. Jest to coś, czego wolelibyśmy uniknąć za wszelką cenę.

1.1.3 *Powód 3: C# jest przyjazny dla dewelopera i łatwy w użyciu*

Najczęstsze zastosowania C# to wytwarzanie oprogramowania na potrzeby przedsiębiorstw i zarazem miejsce, w których C# i .NET szczególnie błyszczą. Jak powinna wyglądać idealna baza kodu w środowisku przedsiębiorstwa? Zapewne chcielibyśmy mieć bazę kodu, po której można się łatwo poruszać, opartą na solidnym menedżerze pakietów i wspomaganą testami (jednostkowymi, integracyjnymi i dymnymi). Dodajmy do tego jeszcze doskonałą dokumentację oraz wieloplatformowość.

> **Definicja** Poprzez *samodokumentujący się kod* (*self-documenting code*) rozumiemy kod, który jest napisany wystarczająco jasno, że nie potrzeba komentarzy objaśniających logikę. Kod dokumentuje się sam. Dla przykładu, jeśli mamy metodę o nazwie `DownloadDocument`, inni będą mieli oczywiste skojarzenie, co ona robi. Nie ma potrzeby dodawania komentarza wyjaśniającego, że logika w tej metodzie pobiera (*download*) dokument.

Na domiar złego, zapewne będziemy chcieli mieć dobrą integrację z usługą chmurową, aby zapewnić ciągłą integrację i dostarczanie (continuous integration and delivery – CI/CD). Pragmatyczne podejście mówi nam, że prawdopodobieństwo posiadania takiej bazy kodu nie jest najwyższe. Naturalnie ta lista życzeń jest nierealistyczna w większości scenariuszy. Jednak jeśli chcemy osiągnąć przynajmniej niektóre z tych rzeczy (albo wszystkie, jeśli ktoś lubi przygody), C# przynajmniej nie będzie nam w tym przeszkadzać. Zapewnia istniejące

już przepływy pracy, funkcjonalności i natywne biblioteki pozwalające pokonać 99% drogi do wymarzonego celu.

Deweloperzy przechodzący z takich języków, jak Java, zauważą pewne podobieństwa w strukturze projektu. Choć istnieją także pewne różnice, nie są one zbyt wielkie. Strukturę projektu C# omówimy szczegółowo w całej książce.

.NET zapewnia również wsparcie dla wielu popularnych frameworków testowych. Microsoft udostępnia Visual Studio Unit Testing Framework, który zawiera MSTest (a poprzez uogólnianie niekiedy jest tak nazywany). MSTest jest po prostu środowiskiem wiersza polecenia dla Visual Studio Unit Testing Framework. Inne typowo używane frameworki testowe to xUnit i NUnit. Można również znaleźć wsparcie dla frameworków imitacyjnych (mocking), takich jak Moq (Moq jest analogiem środowiska Mockito w Javie albo GoMock w Go; więcej informacji na temat stosowania Moq w testach jednostkowych zawiera podrozdział 10.3.3.), SpecFlow (wytwarzanie sterowane zachowaniami, podobne do Cucumber), NFluent (biblioteka płynnych asercji), FitNesse i wielu innych.

Na koniec można uruchamiać kod C# na najrozmaitszych platformach, choć z pewnymi ograniczeniami (niektóre starsze platformy mogą korzystać jedynie ze starszych wersji C# i .NET Framework). Przy stosowaniu .NET 5 możemy uruchamiać ten sam kod w Windows 10, Linuksie i macOS. Ta funkcjonalność miała swój początek jako .NET Core, odgałęzienie .NET Framework, które następnie zostało scalone z .NET Framework (i innymi frameworkami), tworząc .NET 5. Kod C# można nawet uruchamiać w systemach iOS i Android za pośrednictwem narzędzia Xamarin, a także na platformach PlayStation, Xbox oraz Nintendo Switch za pomocą Mono.

1.2 *Kiedy lepiej nie pracować w C#?*

C# nie jest najlepszym wyborem dla każdego i każdych okolicznościach. Oczywiste jest, że powinniśmy zawsze wybierać najlepsze narzędzie dla konkretnej roboty. C# działa świetnie w szerokim zakresie zastosowań. Jednak trzeba mieć na uwadze, że istnieje kilka przypadków użycia, w których nie będziemy chcieli (ani nawet rozważali) użycia C# oraz .NET, a mianowicie:

- Tworzenie systemu operacyjnego
- Kod dla systemów czasu rzeczywistego (oprogramowanie wbudowane)
- Przetwarzanie numeryczne

Przyjrzyjmy się w skrócie, dlaczego C# nie jest dobrym wyborem dla tych przypadków użycia.

1.2.1 *Tworzenie systemu operacyjnego*

Budowanie systemów operacyjnych (OS) jest niezwykle ważnym elementem inżynierii oprogramowania (wręcz kamieniem węgielnym), jednak tylko nieliczni programiści faktycznie tworzą systemy. Wytworzenie OS zajmuje mnóstwo czasu i starań, a odpowiednie

bazy kodu rutynowo obejmują miliony wierszy kodu, tworzonego i utrzymywanego przez wiele lat, a czasami dziesięcioleci.

Główny powód, dla którego C# nie nadaje się do tworzenia OS, sprowadza się do niezgodności pomiędzy ręcznym zarządzaniem pamięcią (kod niezarządzany) a procesem kompilacji C#. Wprawdzie C# pozwala na używanie wskaźników przy korzystaniu z trybu „unsafe", nie może równać się z takim językiem programowania, jak C, jeśli chodzi o łatwość ręcznego zarządzania pamięcią.

Inny problem związany z próbą użycia C# do tworzenia OS jest jego częściowa zależność od kompilatora just-in-time (JIT) (więcej informacji na ten temat zawiera rozdział 2). Wyobraźmy sobie, że musimy uruchomić nasz system operacyjny za pomocą maszyny wirtualnej. Natychmiast ujawni się problem wydajnościowy, gdyż maszyna wirtualna będzie musiała ciągle czekać i nadganiać, aby wykonać kod skompilowany na bieżąco, analogicznie do tego, co się dzieje, gdy kod .NET działa w naszym komputerze. To spostrzeżenie oznacza, że w pełni statycznie kompilowany język jest lepiej dopasowany do potrzeb tworzenia OS.

Tym niemniej istnieją systemy operacyjne napisane w językach wyższego poziomu. Przykładem takim jest Pilot-OS (utworzony przez Xerox PARC w roku 1977), napisany w języku Mesa[1], poprzedniku Javy.

Jeśli ktoś chciałby dowiedzieć się więcej na temat tworzenia systemów operacyjnych, doskonałym źródłem może być społeczność osdev.org (*wiki.osdev.org*). Można tam znaleźć wskazówki, tutoriale i sugerowane lektury. Ze źródeł dotyczących nauki języka C warto wymienić książę Jensa Gustedta *Modern C* (Manning, 2019) oraz klasyczne dzieło *The C Programming Language* Briana Kernighana i Dennisa Ritchie'go (Prentice Hall, 1988)[2].

1.2.2 *Tworzenie wbudowanych systemów czasu rzeczywistego w C#*

Analogicznie jak w przypadku budowania systemu operacyjnego (podrozdział 1.2.1), kod dla systemów czasu rzeczywistego (real-time operating system – RTOS), który najczęściej spotykamy w tzw. systemach wbudowanych, napotyka znaczące problemy wydajnościowe, gdy jest wykonywany za pośrednictwem maszyny wirtualnej. RTOS skanuje kod liniowo, w czasie rzeczywistym i wykonuje instrukcje w konfigurowalnych interwałach, zmieniających się od jednej operacji na sekundę do wielu działań na mikrosekundę, zależnie od życzeń dewelopera i możliwości mikrokontrolera lub sterownika programowalnego (programmable logic controller – PLC), w którym działa ten kod. Maszyna wirtualna w pewnym stopniu wyklucza prawdziwe działanie w czasie rzeczywistym ze względu na opóźnienia i nadmiarowość wprowadzaną w czasie działania.

Osoby zainteresowane kodem dla RTOS i wytwarzaniem oprogramowania wbudowanego mogą zainteresować się wieloma cenionymi książkami, takimi jak *An Embedded Software*

1 Nazwa „Mesa" (jedno ze znaczeń tego wyrazu to „góra stołowa") jest żartem, odnoszącym się do tego, że język programowania jest tak wysokiego poziomu, że wygląda jak izolowane, wyniosłe wzgórze z płaskim szczytem.

2 Pełne zestawienie polecanych lektur znajduje się w dodatku E. Tamże zostały podane polskie wydania wymienionych książek, o ile istnieją (przyp. tłum.).

Primer Davida E. Simona (Addison-Wesley Professional, 1999) albo *Making Embedded Systems: Design Patterns for Great Software* autorstwa Elecii White (O'Reilly Media, 2011).

1.2.3 *Przetwarzanie numeryczne a C#*

Przetwarzanie numeryczne (nazywane też *analizą numeryczną*) dotyczy badania, rozwijania i analizowania algorytmów. Osoby pracujące przy przetwarzaniu numerycznym (zazwyczaj informatycy lub matematycy) używają przybliżeń numerycznych do rozwiązywania problemów w niemal każdej dziedzinie nauki i inżynierii. Z perspektywy języka programowania stwarza to unikatowe wyzwania i uwarunkowania. Każdy język programowania może obliczać matematyczne wzory i formuły, jednak niektóre zostały zbudowane specjalnie do tego celu.

Rozważmy tworzenie wykresów. C# naturalnie jest w stanie obsłużyć kreślenie, ale jaką wydajność i wygodę może zaoferować, gdy porównamy go z czymś takim, jak MATLAB? (MATLAB jest zarówno środowiskiem obliczeniowym, jak i językiem programowania stworzonym przez MathWorks). Krótka odpowiedź brzmi – to w ogóle nie jest porównywalne. Programowanie graficzne w C# oznacza pracę w czymś podobnym do WPF (który wykorzystuje Direct3D), OpenGL, DirectX albo jeszcze innej bibliotece graficznej innego dostawcy (zazwyczaj nakierowanej na gry wideo). W przypadku MATLAB mamy język, który idealnie pasuje do środowiska zbudowanego w celu renderowania złożonych wykresów dwu- i trójwymiarowych. Musimy jedynie wywołać `plot(x, y)`, a MATLAB samodzielnie sporządzi potrzebny wykres.

Tak więc, C# potrafi wykonywać obliczenia numeryczne, ale nie oferuje takiej prostoty użycia, jak język wyposażony w biblioteki wysokiego poziomu oraz abstrakcje zajmujące się kreśleniem wykresów, jak MATLAB. Jeśli ktoś chciałby dowiedzieć się więcej na temat MATLAB lub przetwarzania numerycznego, wśród licznych dostępnych źródeł warto wymienić *Numerical Methods for Scientists and Engineers* Richarda Hamminga (Dover Publications, 1987), *MATLAB: An Introduction with Applications* Amosa Gilata (Wiley, 2016) oraz tutorial Cody dla samego MATLAB-a (*https://www.mathworks.com/matlabcentral/cody*).

1.3 *Przełączanie się na C#*

Ze względu na podobieństwo pomiędzy tymi językami deweloperzy z dobrą znajomością języków opartych na składni maszyny wirtualnej Java (JVM) (najważniejsze z nich to naturalnie Java, a ponadto Scala i Kotlin) albo C++ mogą mieć łatwiej podczas lektury tej książki, niż ktoś, kto przychodzi od języka o stylu innym niż C, nie opartego na maszynie wirtualnej lub języków skoncentrowanych na działaniach chmurowych, takich jak Dart, Ruby lub Go. Doświadczenie w językach „nie-C" nie oznacza jednak, że C# będzie niemożliwy do zrozumienia. Być może okaże się, że niektóre fragmenty trzeba będzie przeczytać dwukrotnie, ale ostatecznie wszystko powinno się udać.

Czytelnicy, którzy przychodzą ze środowiska języka interpretowanego, takiego jak Python, mogą początkowo postrzegać proces kompilacji .NET za coś dziwacznego. Języki leżące w granicach .NET używają dwustopniowego procesu kompilacji. Najpierw kod jest

kompilowany statycznie do języka niższego poziomu, nazywanego Common Intermediate Language (wspólny język pośredni, w skrócie CIL, IL albo MSIL, MS oznacza tu Microsoft – programiści Javy zauważą, że jest to coś analogicznego do kodu bajtowego Javy), który następnie jest kompilowany na żądanie (just-in-time – JIT) na natywny kod maszynowy, gdy środowisko wykonawcze .NET wykonuje kod na docelowym hoście. Może się to wydawać zbyt dużo do strawienia na raz, ale po kilku kolejnych rozdziałach Czytelnicy wszystko zrozumieją.

Dla osób przychodzących z języka skryptowego, takiego jak JavaScript, statyczne typowanie może się wydawać ograniczeniem i powodować frustrację. Jednak gdy już przyzwyczają się do pamiętania cały czas, jaki typ jest używany, zapewne docenią to i polubią.

Natomiast tych, którzy dotychczas posługiwali się takim językiem, jak Go lub Dart, dla których niekiedy trudno jest znaleźć natywne biblioteki, .NET 5 może zaskoczyć bogactwem dostępnych bibliotek. Zapewniając funkcje dla większości rzeczy, o których można pomyśleć, biblioteki .NET są podstawowym źródłem funkcjonalności. Wiele aplikacji napisanych w .NET nigdy nie używa żadnych bibliotek innych firm.

Dla zachowania porządku zajmijmy się teraz narzędziami. W tym rozdziale nie będziemy zajmować się instalowaniem IDE ani .NET SDK. Jeśli ktoś jeszcze nie wykonał tych instalacji i potrzebuje nieco pomocy, dodatek C zawiera kilka szybkich poradników. Aby móc podążać za przykładami przedstawionymi w książce, konieczne jest zainstalowanie najnowszych wersji .NET Framework oraz .NET 5. W tej książce zaczynamy od starej bazy kodu wykorzystującej .NET Framework. Z tego względu będziemy używać .NET Framework do uruchamiania tego starego kodu w miarę migrowania go do .NET 5.

Jak wspomniałem wcześniej, C# jest językiem open source i jest utrzymywany przez społeczność – przy znaczącej pomocy ze strony Microsoftu. Nie trzeba płacić za licencje na środowisko wykonawcze, SDK ani IDE. Jeśli chodzi o IDE, Visual Studio (to środowisko edycyjne jest używane w przykładach przedstawionych w książce) oferuje bezpłatną edycję Community, której można używać do tworzenia osobistych projektów i oprogramowania open source. Jeśli ktoś preferuje aktualnie używane IDE, są spore szanse, że można znaleźć do niego wtyczkę obsługującą C#. Można również posługiwać się trybem wiersza polecenia do kompilowania, uruchamiania i testowania projektów C#, choć osobiście zdecydowanie zachęcam, aby dać szansę narzędziom dedykowanym dla C# (Visual Studio), gdyż zapewnia to najwygodniejszą pracę i jest najprostszą drogą do pisania idiomatycznego kodu C#.

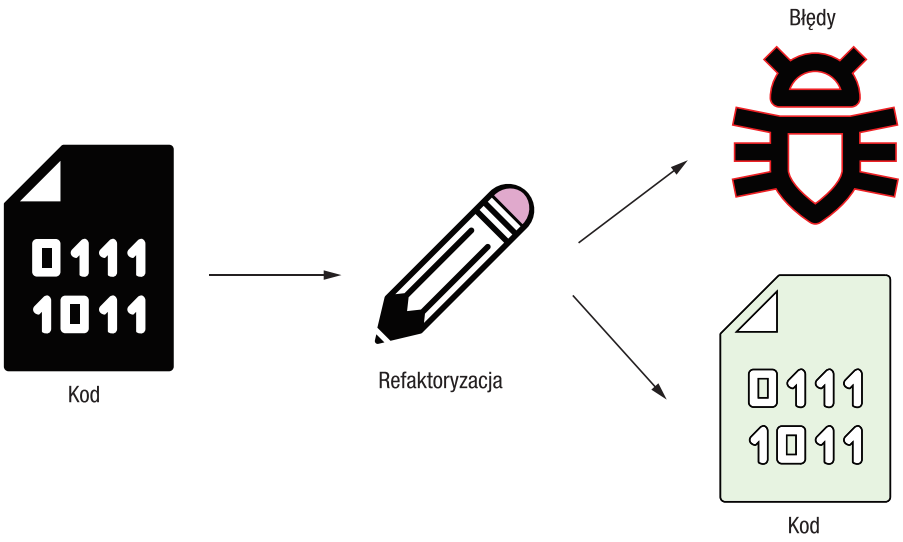
Wiele koncepcji i technik, które znamy z innych miejsc, można przenieść do C#, ale nie wszystkie. C# osiągnął większą dojrzałość w mechanizmach backendu, niż frontendu, gdyż tradycyjnie używany jest głównie w takich celach. Historyczna koncentracja na tworzeniu backendów w C# nie oznacza, że możliwości dotyczące frontendu robią mniejsze wrażenie. Można napisać kompletną aplikację jedynie w C#, bez konieczności dotykania JavaScript. Wprawdzie ta książka również skupia się na tworzeniu backendu, wiele pokazanych tu koncepcji może być pomocnych również przy projektowaniu frontendu.

Wyobraźmy sobie teraz metodę-potwora, z pięcioma zagnieżdżonymi pętlami `for`, mnóstwem zafiksowanych liczb („magiczne cyfry") i większą liczbą komentarzy, niż kodu. Wyobraźmy sobie teraz, że jesteśmy nowym członkiem zespołu, który właśnie do niego dołączył. Jak się poczujemy, gdy uruchomimy IDE, ściągniemy kod źródłowy i zobaczymy taką metodę? Rozpacz to za mało powiedziane. A teraz wyobraźmy sobie, że przenieśliśmy wszystkie indywidualne działania tego potwora w ich własnych, mniejszych metodach

(zapewne krótszych niż 5 do 10 wierszy kodu). Jak teraz wygląda nasz potwór? Zamiast być gąszczem trudnych do prześledzenia zależności i przypisań, bez widocznego sensu, o ile nie dysponujemy bardzo specjalistyczną wiedzą dziedzinową, kod daje się czytać, jak zwykła proza. Jeśli do tego odpowiednio ponazywamy nasze metody, metoda główna powinna teraz wyglądać jak przepis na potrawę, którą potrafi ugotować nawet najgorszy kucharz.

Kiedy piszę o „czystym kodzie", mam na myśli praktyki kodowania zalecane przez Roberta C. Martina w jego tutorialach wideo (*https://cleancoders.com/videos*) oraz książkach *Clean Code* (Prentice Hall, 2008), *Clean Architecture* (Prentice Hall, 2017) oraz, wspólnie z Micah Martin, *Agile Principles, Patterns, and Practices in C#* (Pearson, 2006). Nie wolno też zapominać o ukutych przez niego zasadach „SOLID": Single Responsibility Principle (zasada jednej odpowiedzialność), Open/Closed Principle (zasada otwarte/zamknięte), Liskov Substitution Principle (zasada podstawienia Liskov), Interface Segregation Principle (zasada segregacji interfejsów) oraz Dependency Inversion Principle (zasada odwrócenia zależności). Zasady czystego kodu wyjaśnię dogłębnie, gdy przyjdzie na to czas, wraz z praktycznymi informacjami, jak ich faktycznie używać.

Dlaczego jednak w ogóle mamy trudzić się pisaniem czystego kodu? Czysty kod działa jak pralka wobec błędów i niewłaściwej funkcjonalności. Jeśli włożymy naszą bazę kodu do pralki czystego kodu, jak na rysunku 1.3, zobaczymy, że gdy zrefaktoryzujemy coś, aby było „czyste", błędy wychodzą na wierzch, a niewłaściwe funkcjonalności tkwią schwytane w światła reflektorów, bez miejsca do ukrycia. Ostatecznie „wszystko wychodzi w praniu". Oczywiście, refaktoryzacja kodu produkcyjnego wiąże się także z ryzykiem; często pojawiają się niezamierzone efekty uboczne. Sprawia to, że zarządowi trudniej jest zaaprobować wielkie refaktoryzacje bez dodawania funkcjonalności. Jednak dysponując odpowiednimi narzędziami (niektóre zostaną omówione w tej książce) można zminimalizować ryzyko negatywnych efektów ubocznych i poprawić jakość bazy kodu.



Rysunek 1.3 Metodologia czystego kodu jest jak pralka dla naszego kodu. Przyjmuje brudną bieliznę (nasz kod), dodaje mydło i wodę (zasady czystego kodu) i oddziela brud od ubrań (odseparowuje błędy od kodu). To co z niej wychodzi, to ubrania (kod) z mniejszą ilością brudu (błędów), niż było na początku.

Książka zawiera szereg ramek zawierających informacje o zagadnieniach czystego kodu. Jeśli ramka dotyczy czystego kodu, oznakowuję ją jako taką i wyjaśniam zarówno koncepcje, jak i sposoby ich stosowania w rzeczywistym świecie. Dodatek B zawiera listę kontrolną czystego kodu. Można ją wykorzystać w celu ustalenia, czy potrzebujemy zrefaktoryzować istniejący kod. Lista ta służy też jako ściągawka dla niektórych z częściej zapominanych (choć nadal ważnych) koncepcji.

1.4 *Czego można się nauczyć z tej książki*

Książka ta ma na celu nauczenie pisania idiomatycznego i czystego kodu C#. Nie uczy języka C#, .NET 5 ani programowania od podstaw. Będziemy stosować się do praktycznego podejścia: realistycznego scenariusza biznesowego, w którym wykonamy refaktoryzację starego API, aby było czystsze i bezpieczniejsze. Po drodze nauczymy się wielu rzeczy. Oto niektóre z nich:

- Refaktoryzacja starej (istniejącej) bazy kodu pod kątem zabezpieczeń, wydajności i przejrzystości.
- Pisanie samo-dokumentowanego kodu, który może przejść dowolny przegląd.
- Używanie wytwarzania sterowanego testami w celu pisania testów równolegle z kodem implementacji.
- Bezpieczne łączenie się z chmurową bazą danych za pomocą Entity Framework Core.
- Wprowadzenie zasad czystego kodu do istniejącej bazy kodu.
- Czytanie Common Intermediate Language (języka pośredniego) i wyjaśnienie procesu kompilacji C#.

Co zatem musimy wiedzieć, aby najbardziej skorzystać z tej książki? Oczekuję, że Czytelnik zna i rozumie podstawowe zasady programowania obiektowego (dziedziczenie, enkapsulacja, abstrakcja i polimorfizm) i dobrze zna przynajmniej jeden inny język programowania, który pozwala tworzyć kod za pomocą podejścia obiektowego (na przykład C++, Go, Python lub Java).

Po przeczytaniu tej książki Czytelnik będzie pisać czysty, bezpieczny, testowalny kod C#, zgodny z dobrymi praktykami i zasadami projektowania obiektowego. Dodatkowo będzie gotowy na dalsze pogłębianie swojej znajomości C# poprzez zaawansowane źródła. Niektóre sugerowane lektury obejmują: Jon Skeet, *C# in Depth*, 4 wydanie (Manning, 2019), Jeffrey Richter, *CLR via C#*, 4 wydanie (Microsoft Press, 2012), Bill Wagner *Effective C#*, 2 wydanie (Microsoft Press, 2016), Dustin Metzgar, *.NET Core in Action* (Manning, 2018), John Smith, *Entity Framework Core in Action*, 2 wydanie (Manning, 2021) oraz Andrew Lock, *ASP.NET Core in Action*, 2 wydanie (Manning, 2021).

1.5 *Czego nie nauczymy się z tej książki*

Książka ta ma na celu wypełnienie luki pomiędzy źródłami dla początkujących i dla zaawansowanych. Cel ten pociąga za sobą pewne konsekwencje co do założeń, które poczyniłem na temat znajomości języka C# i programowania w ogólności. Jak wyjaśniłem wcześniej, oczekuję, że Czytelnik ma pewne profesjonalne doświadczenia programistyczne i że czuje się swobodnie w zakresie podstaw języka C# lub innego obiektowego języka programowania.

Co przez to rozumiem? Aby jak najwięcej skorzystać z tej książki, Czytelnik powinien rozumieć zasady programowania obiektowego i być w stanie tworzyć podstawowe aplikacje lub API w swoim ulubionym języku programowania. W rezultacie książka ta nie próbuje uczyć żadnego z poniższych zagadnień, które zazwyczaj są zawarte w podręcznikach programowania dla początkujących:

- Samego języka C#. To nie jest książka w rodzaju *C# od podstaw* albo *Elementarz C#*. Zamiast tego zamierzam pokazać, jak podnieść już posiadaną wiedzę o języku C# lub programowaniu obiektowym na wyższy poziom.
- Składni instrukcji warunkowych i sterujących, które nie są specyficzne dla C# (`if`, `for`, `foreach`, `while`, `do-while`, itp.).
- Czym są polimorfizm, enkapsulacja i dziedziczenie (choć pojęcia te będą regularnie używane w całej ksiażce).
- Czym jest klasa i jak wykorzystujemy klasy do modelowania obiektów świata rzeczywistego.
- Czym są zmienne ani jak przypisywać do nich wartości.

Jeśli ktoś dopiero rozpoczyna przygodę z programowaniem, zdecydowanie zalecam przeczytanie najpierw innej książki, takiej jak *Head First C#* autorstwa Jennifer Greene i Andrew Stellmana, 4 wydanie (O'Reilly, 2020) albo *Structure and Interpretation of Computer Programs* Harolda Abelsona, Geralda Jaya Sussmana i Julie Sussman, 2 wydanie (The MIT Press, 1996)[3].

Książka ta nie obejmuje również poniższych, bardziej specjalistycznych sposobów posługiwania się językiem C#:

- Architektura mikrousług. W tej książce nie zajmuję się tym, czym są mikrousługi i jak ich używać. Architektura mikrousług jest obecnie wiodącym trendem i jest użyteczna w bardzo wielu sytuacjach, ale nie jest powiązana z C# ani z profesjonalnym tworzeniem kodu. Trzy wspaniałe źródła, z których można dowiedzieć się więcej na temat mikrousług, to książki Chrisa Richardsona *Microservices Patterns* (Manning, 2018), Prabatha Siriwardena oraz Nuwan Diasa *Microservices Security in Action* (Manning, 2019) oraz Christiana Horsdala Gammelgaarda *Microservices in .NET Core* (Manning, 2020).

3 Książka jest dostępna za darmo w witrynie MIT Press pod adresem *https://mitpress.mit.edu/sites/default/files/sicp/index.html*.

- Wykorzystywanie C# w środowiskach skonteneryzowanych, takich jak Kubernetes i/lub Docker. Choć jest to bardzo praktyczne i stosowane w wielu środowiskach deweloperskich, umiejętność korzystania z Kubernetes lub Dockera nie gwarantuje, że będziemy umieli „kodować jak profesjonalista" w C#. Więcej informacji na temat tych technologii można znaleźć np. w książkach Marko Lukša *Kubernetes in Action*, 2 wydanie (Manning, 2021), Eltona Stonemana *Learn Docker in a Month of Lunches* (Manning, 2020) oraz Ashley'a Davisa *Bootstrapping Microservices with Docker, Kubernetes, and Terraform* (Manning, 2021).
- Współbieżność w C#, poza zagadnieniami wielowątkowości i blokad (omówionymi w rozdziale 6). Często napotykamy na to zagadnienie w wysokowątkowych, wydajnościowo istotnych scenariuszach. Większość deweloperów jednak nie ma do czynienia z kodem tego rodzaju. Jeśli jednak ktoś znalazł się w takiej sytuacji, doskonałym źródłem nauczenia się więcej o programowaniu współbieżnym w C# jest książka Joe Duffy'ego *Concurrent Programming on Windows* (Addison-Wesley, 2008).
- Głębokie wewnętrzne szczegóły tak CLR, jak i samego .NET Framework. Choć CLR i .NET 5 są interesującymi tematami, znajomość ich wszystkich drobnych detali nie ma praktycznego zastosowania dla większości deweloperów. Książka ta omawia CLR oraz .NET Framework do pewnego poziomu szczegółów, ale zatrzymuje się, gdy dalsze detale stają się nieprzydatne lub nieporęczne. „Biblią" dla CLR i .NET Framework jest książka Jeffrey'a Richtera *CLR via C#*, 4 wydanie (Microsoft Press, 2012).

Czytelnik ma do wyboru dwa sposoby czytania tej książki. Sposób zalecany to przeczytanie jej w całości, od początku do końca i w tej kolejności. Jeśli jednak ktoś jest zainteresowany jedynie refaktoryzacją i najlepszymi praktykami, może przeczytać jedynie części od 3 do 6.

Podsumowanie

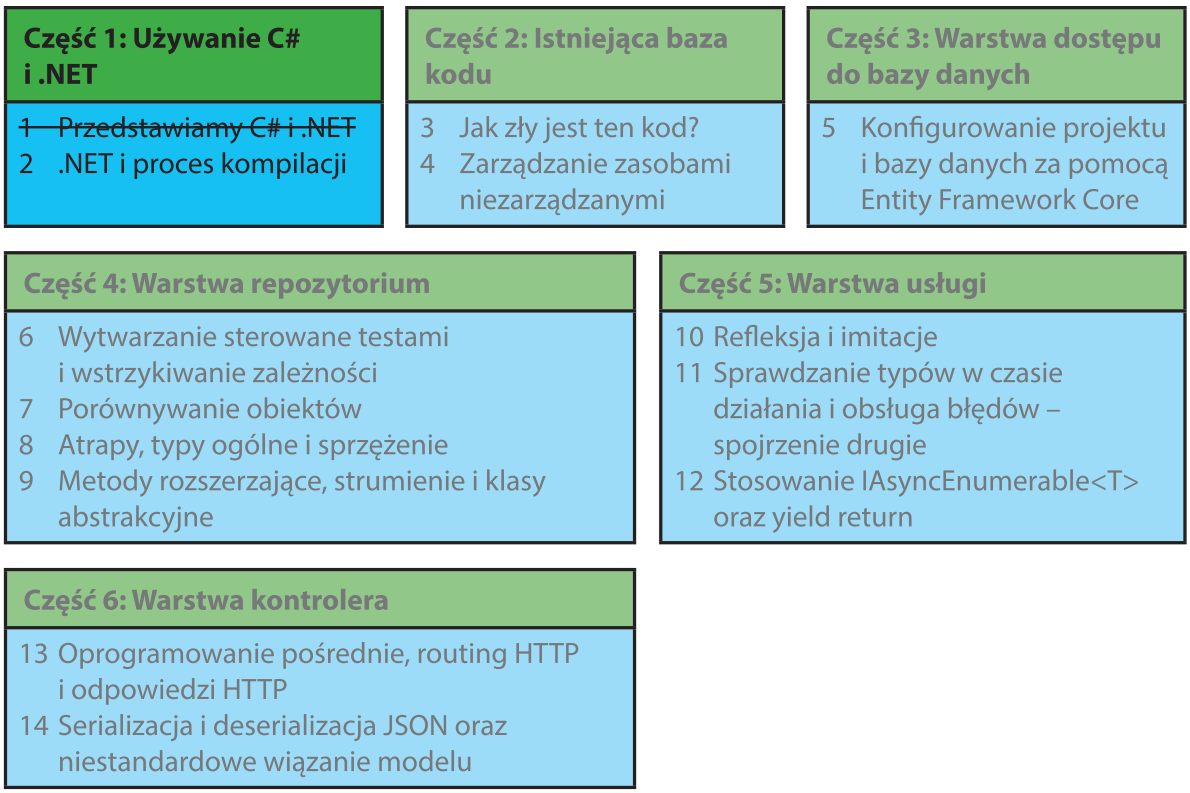
- Książka ta nie omawia „programowania od podstaw". Zakłada (dość zaawansowaną) znajomość programowania obiektowego. Pozwala to skupić się na praktycznych koncepcjach.
- C# i .NET 5 szczególnie wyróżniają się w tworzeniu skalowalnego oprogramowania klasy przedsiębiorstwa, ze skupieniem uwagi na stabilność i łatwość utrzymania. Sprawia to, że C# i .NET są doskonałym wyborem platformy zarówno dla firm, jak i indywidualnych deweloperów.
- C# i .NET 5 nie sprawdzają się w takich zagadnieniach, jak tworzenie systemów operacyjnych, oprogramowania wbudowanego czasu rzeczywistego czy przetwarzaniu numerycznym (albo analizach). Do takich zadań lepiej pasują C lub MATLAB.

2 .NET i proces kompilacji

W tym rozdziale

- Kompilowanie C# do kodu natywnego (maszynowego)
- Odczytywanie i rozumienie języka pośredniego

W roku 2020 Microsoft udostępnił .NET 5, wszechstronną platformę wytwarzania oprogramowania. Wcześniej jednak, na przełomie XX i XXI wieku Microsoft stworzył .NET Framework, który był prekursorem .NET 5. Oryginalnym przypadkiem użycia dla .NET Framework było budowanie aplikacji Windows dla przedsiębiorstw. W istocie w rozdziałach 3 i 4 wykorzystamy .NET Framework do zbadania takiej właśnie bazy kodu. Platforma .NET Framework wiąże ze sobą wielki zbiór bibliotek. Choć .NET Framework i C# często są używane wspólnie, napotykamy również przypadki użycia .NET Framework bez wykorzystania C# (innymi słowy, z użyciem odmiennego języka z rodziny .NET). Najważniejszymi filarami .NET Framework są Framework Class Library (FCL, ogromna biblioteka klas, która stanowi kręgosłup .NET Framework) oraz Common Language Runtime (CLR, środowisko wykonawcze .NET, które zawiera kompilator JIT, mechanizm odśmiecający, proste typy danych i jeszcze więcej). Innymi słowy, FCL zawiera wszystkie biblioteki, których zapewne będziemy chcieli używać, zaś CLR odpowiada za wykonanie kodu. W późniejszym czasie Microsoft wprowadził środowisko .NET Core, nakierowane na programowanie wieloplatformowe. Rysunek 2.1 pokazuje, gdzie rozdział ten plasuje się w schemacie tej książki.



Rysunek 2.1 Jak dotąd, Czytelnik dowiedział się, czego może oczekiwać od tej książki. W tym rozdziale zagłębimy się w rozważania, czym jest .NET i jakie są jego cechy. Dzięki omówieniu ekosystemu .NET uzyskamy podstawową wiedzę, która będzie nam dobrze służyć w dalszej części książki.

W tym rozdziale omówimy kilka cech .NET 5 i porównamy je z implementacjami (a niekiedy z ich brakiem) na innych platformach, takich jak Java, Python i Go. Następnie poznamy proces kompilacji kodu C#, pokazując, jak metoda napisana w C# jest tłumaczona na język pośredni (Common Intermediate Language – CIL) i później na kod natywny (maszynowy). Te podstawowe cegiełki pozwolą nam uzyskać solidny fundament poznawania języka C# i ekosystemu .NET. Jeśli Czytelnik już zna C# i .NET, rozdział ten będzie dla niego rodzajem powtórki. Jednak nawet wówczas zalecam przeczytanie przynajmniej podrozdziału 2.3. Przedstawione tu omówienie procesu kompilacji C# jest bardziej pogłębione, niż spotykane w większości podręczników i zakłada wiedzę na temat niektórych zaawansowanych zasobów C#. Podrozdziały 2.2 i 2.3 zawierają ćwiczenia, które Czytelnik może wykorzystać do sprawdzenia swojej wiedzy.

2.1 *Czym jest .NET Framework?*

Na początku… był sobie .NET Framework – staromodny sposób używania .NET. Środowisko .NET Framework zostało wprowadzone przez Microsoft na początku lat 2000. Programiści mogli używać C# wspólnie z tym środowiskiem, aby tworzyć aplikacje biurkowe dla przedsiębiorstw. Jako że Microsoft ma oczywisty interes w nakierowaniu na Windows, .NET Framework działa tylko w systemach Windows i polega na wielu API

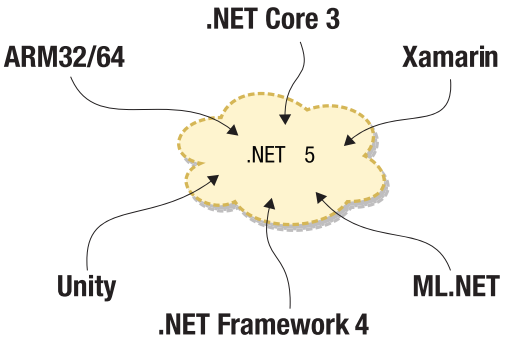
Windows przy wykonywaniu operacji graficznych. Jeśli ktoś pracuje w dowolnej aplikacji biurkowej napisanej w C# przed końcem roku 2020 (czyli przed wprowadzeniem .NET 5), mogę zagwarantować, że została ona napisana przy użyciu .NET Framework.

Z upływem czasu .NET Framework przechodziło różne iteracje, ale najnowsze dostępne wydanie (z lipca 2019) ma wersję 4.8.0. Nie będzie już kolejnych wydań .NET Framework, jako że zostało ono zastąpione przez .NET 5, ale wiele starszych aplikacji nadal będzie je wykorzystywać. Znaczna część materiału omawianego w tej książce dotyczy .NET Framework. W istocie zobaczymy aplikację środowiska .NET Framework w rozdziałach 3 i 4.

2.2 *Czym jest .NET 5?*

W tym rozdziale zajmiemy się tym, czym jest .NET 5 i dlaczego powstało. Od roku 2016 ekosystem .NET istniał w formie dwóch głównych strumieni: .NET Framework oraz .NET Core. Nowe środowisko .NET 5 łączy oba te strumienie (a także kilka innych pomocniczych strumieni, takich jak Xamarin i Unity), co pokazuje rysunek 2.2. W rezultacie .NET 5 można uznać za rebranding .NET Core, formującego podstawy nowego .NET. Powinniśmy zatem postrzegać .NET 5 nie jako kolejną iterację .NET Framework lub .NET Core, ale raczej jako nowy początek wynikający ze scalenia starszych technologii.

Umieszczenie wszystkich technologii .NET pod jednym parasolem daje nam dostęp do wszystkich narzędzi i przypadków użycia, jakich możemy potrzebować. W ramach tego samego frameworku możemy opracowywać oprogramowanie dla przedsiębiorstw, witryny Web, gry wideo, oprogramowanie Internetu rzeczy (Internet of Things – IoT), wbudowane aplikacje działające w systemach Windows/macOS/Linux, programy dla procesorów ARM (ARM32/64), usługi uczenia maszynowego (ML.NET), aplikacje biurkowe, usługi chmurowe i aplikacje mobilne. A ponieważ .NET Framework jest zgodne z .NET Standard, wszystkie istniejące bazy kodu i biblioteki powinny być kompatybilne z .NET 5 (o ile .NET 5 wspiera leżące w tle mechanizmy pakietów i funkcjonalności używane przez te bazy kodu).



Rysunek 2.2 .NET 5 łączy .NET Framework 4 z ARM32/64, Xamarin, .NET Core 3, Unity i ML.NET. Pozwala to na używanie wszystkich odmian .NET w jednym środowisku: .NET 5.

.NET 5, podobnie jak .NET Framework i .NET Core, jest implementacją .NET Standard – specyfikacji, która była używana do opracowania rozmaitych implementacji: .NET 5, .NET Framework, .NET Core, Mono (technologia międzyplatformowa, na której zbudowano

.NET Core), Unity (programowanie gier wideo) oraz Xamarin (wytwarzanie programów dla iOS i Androida). Mają one różne zastosowania, ale z natury są bardzo podobne. Opracowanie implementacji zgodnej z .NET Standard oznacza, że współużytkowanie kodu pomiędzy implementacjami jest tak płynne i bezproblemowe, jak to możliwe.

.NET Standard zawiera informacje o tym, jakie API są dostępne do użycia podczas interakcji z CLR (silnikiem wykonawczym, od którego zależy C#). Przed wprowadzeniem .NET Standard nie mieliśmy praktycznego sposobu zagwarantowania, że nasz kod lub biblioteka będzie działać w różnych implementacjach .NET, poza użyciem przenośnych bibliotek klas (Portable Class Libraries – PCL). Są to biblioteki, które można współdzielić pomiędzy różnymi projektami, ale są nacelowane tylko na określoną wersję implementacji .NET (nazywaną „profilem"). Obecnie takie PCL nazywane są „PCL-ami opartymi na profilach". Biblioteki nakierowane na implementacje .NET zgodne z .NET Standard również są PCL, ale zamiast wskazywania konkretnej implementacji, określają wersję .NET Standard. Aby odróżnić je od PCL opartych na profilach, będziemy je nazywać „PCL-ami opartymi na .NET Standard". Specyfikacja .NET Standard opakowuje wiele API Windows używanych przez biblioteki napisane w czasach przed powstaniem .NET Standard (a tym samym również PCL-e oparte na profilach). W rezultacie możemy bez problemów używać tych bibliotek w dowolnej implementacji .NET Standard. Pierwszą wersją .NET Framework, która implementowała .NET Standard, była wersja 4.5.

Zgodnie ze zwróceniem się Microsoftu w stronę oprogramowania open source, .NET 5 i wszystkie powiązane z nim repozytoria są oprogramowaniem otwartym, dostępnym w serwisie GitHub (*https://github.com/dotnet*). Więcej informacji na temat nowych funkcjonalności, które mają zostać dołączone w nowych wersjach .NET można znaleźć w mapie drogowej CoreFX dostępnej pod adresem *https://github.com/dotnet/corefx/milestones*. Samą specyfikację .NET Standard można znaleźć pod adresem *https://github.com/dotnet/standard*.

Ćwiczenia

Ćwiczenie 2.1

Który z poniższych systemów operacyjnych nie jest wspierany przez .NET 5?

a. Windows
b. macOS
c. Linux
d. AmigaOS

Ćwiczenie 2.2

Co oznacza akronim „CLR"?

a. Creative License Resources
b. Class Library Reference
c. Common Language Runtime

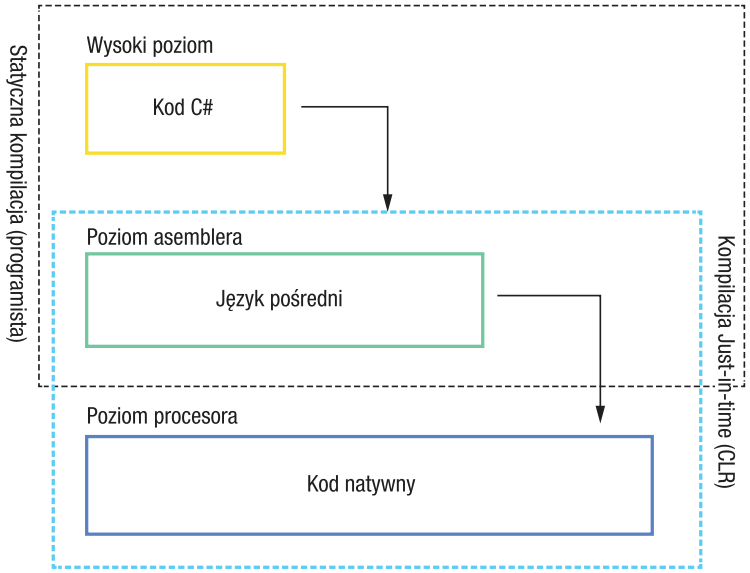
Ćwiczenie 2.3

Wypełnij luki: „.NET Standard jest _____, która nakazuje szczegóły implementacyjne wszystkich platform .NET w celu umożliwienia współdzielenia kodu”.

a. implementacją
b. prekursorem
c. narzędziem
d. specyfikacją

2.3 *Jak kompilowane są języki zgodne z CLI*

W tym podrozdziale przedstawię pogłębiony obraz tego, jak przebiega kompilacja kodu C# (a także innych języków zgodnych z Common Language Infrastructure; patrz podrozdział 2.3.2). Poznanie całego procesu kompilacji pomoże w owocnym wykorzystaniu wszystkich cech C#, a zarazem zapewni zrozumienie pewnych pułapek związanych z pamięcią i wykonywaniem. Proces kompilacji C# rozróżnia trzy stany (C#, język pośredni CIL oraz kod natywny) i dwie fazy, co pokazuje rysunek 2.3: przejście od C# do CIL oraz od języka pośredniego do kodu natywnego.



Rysunek 2.3 Pełny proces kompilacji programu w języku C#. Prowadzi on od kodu C# do Common Intermediate Language i następnie do kodu natywnego. Zrozumienie procesu kompilacji zapewnia wiedzę o niektórych z wewnętrznych wyborów dokonanych w C# i .NET.

Uwaga Kod natywny jest też nazywany *kodem maszynowym*.

Dzięki analizie tego, co jest potrzebne, aby przejść z jednego kroku do następnego i podążaniu za przekształcaniem kodu wysokiego poziomu (C#) na wykonywalny kod maszynowy uzyskamy zrozumienie złożonej maszynerii, jaką jest C# i .NET 5. Omówienie tego procesu

często jest pomijane w źródłach dla początkujących, ale zaawansowane źródła wymagają jego znajomości.

Użyjemy połączenia kompilacji statycznej oraz kompilacji JIT, aby przekształcić kod C# na kod natywny, jak następuje:

1. Gdy programista napisze kod C#, wywołuje jego kompilację. Powoduje to powstanie kodu w języku pośrednim (Common Intermediate Language), zapisanego w plikach Portable Executable (PE w przypadku kodu 32-bitowego, PE+ dla 64-bitowego), takich jak pliki .exe i .dll dla systemu Windows. Pliki te są następnie dystrybuowane lub wdrażane u użytkowników.
2. Gdy uruchamiamy program .NET, system operacyjny wywołuje Common Language Runtime. Kompilator JIT należący do CLR kompiluje CIL do kodu natywnego odpowiedniego dla platformy (maszyny i systemu operacyjnego), w którym został uruchomiony. Dzięki temu programy napisane w językach zgodnych z CLI mogą działać w wielu platformach. Jednak byłoby rzeczą niewłaściwą, gdybyśmy nie wspomnieli o głównej negatywnej implikacji używania maszyny wirtualnej i kompilatora JIT do uruchamiania kodu: wydajności.

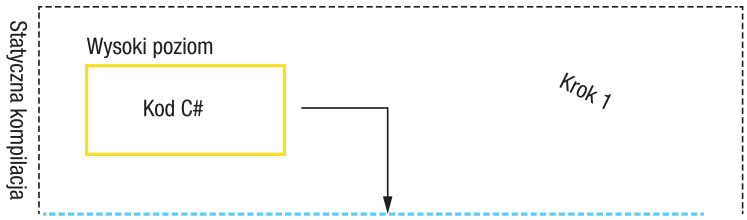
Statycznie skompilowany program ma przewagę podczas wykonywania, gdyż nie musi czekać na skompilowanie kodu przez środowisko wykonawcze.

> **Definicja** Kompilacja statyczna oraz JIT to dwie powszechnie używane metody kompilowania kodu. C# używa połączenia kompilacji statycznej i JIT. Kompilacja JIT oznacza, że kod jest tłumaczony „w dół" do kodu maszynowego w ostatnim możliwym momencie. Kompilacja statyczna oznacza skompilowanie całości kodu źródłowego zawczasu.

2.3.1 *Krok 1: Kod C# (wysokiego poziomu)*

Po raz pierwszy zetknąłem się z twierdzeniem Pitagorasa w roku 2008. Chodziłem wówczas do duńskiego odpowiednika gimnazjum i w podręczniku do matematyki zauważyłem, że będziemy w tej klasie omawiać twierdzenie Pitagorasa. Po kilku dniach, późnym wieczorem, jechałem samochodem ze swoim ojcem. Jechaliśmy już jakiś czas, gdy zupełnie niespodziewanie zapytałem „Co to jest twierdzenie Pitagorasa?" Pytanie wyraźnie go zaskoczyło, gdyż do tej pory nie wykazywałem naukowych zainteresowań, a zwłaszcza w matematyce. Przez kolejnych dziesięć minut próbował wyjaśnić mi, mającego zdolności matematyczne grapefruita, o co chodzi w tym twierdzeniu. Byłem zaskoczony tym, że rzeczywiście zrozumiałem wszystko, o czym mówił i dziś, wiele lat później, mogę użyć tego doświadczenia jako doskonałego źródła do zaprezentowania pierwszego kroku w procesie kompilacji C#.

W tym punkcie zajmiemy się pierwszym krokiem procesu kompilacji C#: tłumaczeniem kodu C#, co pokazuje rysunek 2.4. Program, który wykorzystamy w tym celu, będzie obliczać wzór Pitagorasa. Powód, dla którego użyłem takiego właśnie programu, jest prosty: jesteśmy w stanie skondensować twierdzenie Pitagorasa w paru wierszach kodu, które są zrozumiałe na poziomie gimnazjalnym, jeśli chodzi o wiedzę matematyczną. Pozwoli to się skupić na przebiegu kompilacji, a nie na szczegółach implementacyjnych.



Rysunek 2.4 Proces kompilacji C#, krok 1: kod C#. Jest to faza kompilacji statycznej.

> **Uwaga** Dla przypomnienia, twierdzenie Pitagorasa stwierdza, że w trójkącie prostokątnym o bokach a, b i c spełniona jest równość $a^2 + b^2 = c^2$. Twierdzenie to jest typowo używane do znajdowania długości przeciwprostokątnej trójkąta poprzez obliczenie pierwiastka kwadratowego z sumy kwadratów boków przylegających do kąta prostego.

Rozpoczniemy od napisania prostej metody obliczającej kwadrat długości przeciwprostokątnej dla podanych dwóch argumentów (długości boków), pokazanej w poniższym listingu.

Listing 2.1 Wzór Pitagorasa (w języku wysokiego poziomu)

```
public double Pythagoras(double sideLengthA, double sideLengthB) {                ❶
    double squaredLength = sideLengthA * sideLengthA + sideLengthB * sideLengthB;  ❷
  return squaredLength;
}
```

❶ Deklarujemy metodę z modyfikatorem dostępu public, zwracającą liczbę zmiennoprzecinkową, nazwaną Pythagoras, która oczekuje dwóch argumentów zmiennoprzecinkowych (double): sideLengthA oraz sideLengthB.

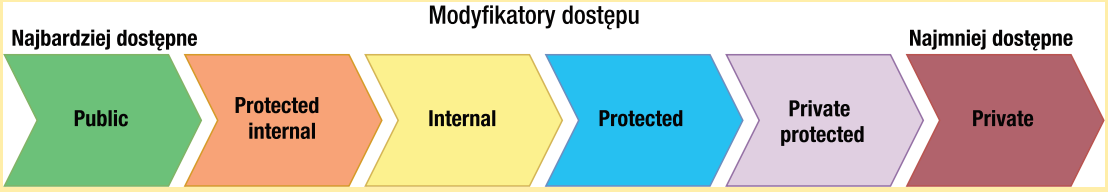
❷ Wykonujemy wzór Pitagorasa i przypisujemy wynik do zmiennej o nazwie squaredLength.

Jeśli uruchomimy ten kod i przekażemy do niego argumenty `[3, 8]`, zobaczymy, że wynik wynosi 73, co jest wartością poprawną. Ponieważ używamy 64-bit liczb zmienno-przecinkowych, możemy przetestować również takie argumenty, jak 34.8706 i 24.0267. Wynik to 1793.236356.

Modyfikatory dostępu, asemblacje i przestrzenie nazw

Język C# rozróżnia sześć modyfikatorów dostępu (od najbardziej otwartego po najbardziej restrykcyjny): `public`, `protected internal`, `internal`, `protected`, `protected private` oraz `private`. Dwa najczęściej używane w codziennej pracy to `public` i `private`. *Public* oznacza dostępność we wszystkich klasach i projektach (odpowiada to koncepcji „exported" w niektórych językach; w odróżnieniu od niektórych z tych języków programowania, wielkość liter w nazwie metody nie ma znaczenia, jeśli chodzi o modyfikatory dostępu w C#), zaś *private* sprawia, że dany element jest widoczny tylko z wnętrza bieżącej klasy.

Pozostałe cztery modyfikatory są rzadziej używane, ale warto je znać. *Internal* zapewnia dostęp ze wszystkich klas w tej samej asemblacji, zaś *protected* ogranicza dostęp tylko dla tych klas, które dziedziczą z oryginalnej klasy. Kolejny to *internal protected*. Ten modyfikator dostępu jest kombinacją modyfikatorów `internal` i `protected`. Zapewnia dostęp z klas potomnych i w ramach tej samej asemblacji. *Private protected* zapewnia dostęp w ramach tej samej asemblacji, ale tylko dla kodu z tej samej klasy albo klasy potomnej.



Modyfikatory dostępu C# od najbardziej otwartego po najbardziej restrykcyjny. Używanie właściwych modyfikatorów jest pomocne w enkapsulacji danych i zabezpieczaniu klas.

Możemy teraz skompilować kod. Przyjmijmy, że metoda z listingu 2.1 jest częścią klasy o nazwie `HelloPythagoras`, która jest częścią projektu i rozwiązania również nazwanego `HelloPythagoras`. Aby skompilować rozwiązanie .NET 5 (albo .NET Framework/.NET Core) do języka pośredniego zapisanego w pliku PE/PE+, można użyć polecenia build lub przycisku kompilacji w używanym IDE, albo wykonać poniższe polecenie w wierszu poleceń:

```
dotnet build [ścieżka do pliku rozwiązania]
```

Pliki rozwiązań (solution) mają rozszerzenie nazwy .sln. Tak więc polecenie tworzące nasze rozwiązanie wygląda następująco:

```
dotnet build HelloPythagoras.sln
```

Po wywołaniu polecenia uruchomi się kompilator. Najpierw kompilator pobierze wszystkie wymagane pakiety zależności, wykorzystując menedżera pakietów NuGet. Następnie narzędzie skompiluje projekt i zapisze wyjście w nowym folderze o nazwie bin. W tym folderze potencjalnie mogą się znaleźć dwa kolejne foldery, debug i release, zależnie od trybu działania kompilatora (można również definiować własne tryby, jeśli tego potrzebujemy). Domyślnie kompilator działa w trybie debugowania. Tryb ten dołącza wszystkie informacje debugowania (zapisywane w plikach .pdb), które są potrzebne, aby móc krokowo przejść przez aplikację przy użyciu punktów przerwań.

Aby wykonać kompilację w trybie wydania (release) z wiersza poleceń, dodajemy do polecenia flagę `--Configuration release`. Alternatywnie w Visual Studio możemy wybrać tryb debug lub release z listy rozwijanej. Jest to najłatwiejszy, najszybszy i najbardziej prawdopodobny sposób kompilowania kodu.

W tym momencie kod wysokiego poziomu napisany w C# został skompilowany do pliku wykonywalnego, zawierającego kod języka pośredniego.

Tryby budowania debug i release

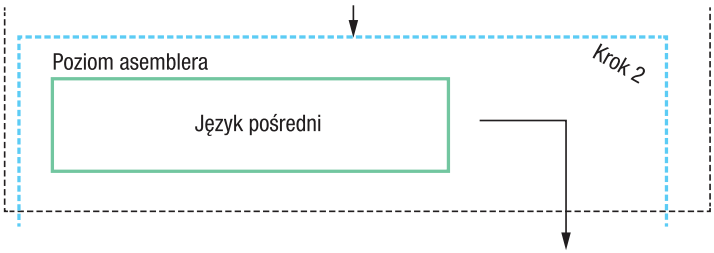
W codziennym życiu praktyczne różnice pomiędzy trybami debug i release objawiają się w wydajności i bezpieczeństwie. Poprzez dołączenie odsyłaczy do plików .pdb w plikach wynikowych do debugowania silnik wykonawczy musi przebiegać przez większą ilość kodu, aby wykonać tę samą logikę, którą zrobiłby w trybie wydania, gdzie odsyłacze te są nieobecne. W rezultacie kod języka pośredniego potrzebny do wymodelowania tego kodu jest większy i jego kompilacja trwa dłużej.

Dodatkowo, jeśli dołączamy informacje debugowania, ktoś z nieczystymi intencjami mógłby potencjalnie wykorzystać te informacje na swoją korzyść i będzie miał łatwiejszy wgląd w nasz kod. Nie oznacza to, że kompilacja w trybie wydania usuwa potrzebę stosowania dobrych praktyk zabezpieczeń. Język pośredni można łatwo zdekompilować (bez względu na to, czy został skompilowany w trybie debugowania, czy wydania) do czegoś podobnego do oryginalnego kodu źródłowego. Jeśli ktoś chce odpowiednio zabezpieczyć swój kod źródłowy, może rozważyć sięgnięcie po rozmaite zaciemniacze (np. Dotfuscator, .NET Reactor) i modele zagrożeń.

Dobrą regułą praktyczną jest tworzenie oprogramowania w trybie debugowania i testowanie kodu w obydwu trybach. Często przybiera to formę testowania lokalnego w trybie debugowania i wykonywania testów akceptacji użytkownika w dedykowanym środowisku za pomocą kompilacji w trybie wydania. Jako że kod wynikowy dla obu trybów różni się nieznacznie, może się okazać, że w trybie wydania znajdziemy błędy, których nie widać w trybie debugowania. Nie chcielibyśmy się znaleźć w sytuacji, gdy całość testów wykonaliśmy tylko w trybie debugowania i odkryjemy blokujący bug w kompilacji finalnej chwilę przed ostatecznym terminem.

2.3.2 *Krok 2: Common Intermediate Language (poziom asemblera)*

Z codziennego punktu widzenia nasza praca jest zakończona. Kod ma postać wykonywalną i możemy odhaczyć kolejną pozycję lub historyjkę użytkownika. Jednak z technicznej perspektywy droga dopiero się zaczęła. Kod C# został statycznie skompilowany do języka pośredniego (Common Intermediate Language), co pokazuje rysunek 2.5, ale system operacyjny nie jest w stanie wykonywać IL.



Rysunek 2.5 Proces kompilacji C#, krok 2: Intermediate Language. W tym miejscu przechodzimy od kompilacji statycznej do just-in-time.