Wstęp

Popularność platformy .NET i języka C# stale rośnie od ich wprowadzenia w roku 2001. Główny autor języka C#, Anders Hejlsberg, kierował w tym czasie kilkoma grupami programistów pracującymi nad stałym ulepszaniem platformy aż do obecnej wersji .NET 4.6 i bardzo ważnej jej nowej odmiany .NET Core 1.0/1.1. Jego praca związana jest też z nowym językiem TypeScript, który również omówimy w tej książce.

Niniejsza książka stanowi podróż przez różne opcje i możliwości platformy .NET Framework w ogólności oraz języka C# w szczególności. Pokazuje programistom, jak budować aplikacje działające w systemie Windows oraz (co zobaczymy w ostatnim rozdziale) na innych platformach i urządzeniach.

Sądzę, że może być pomocna dla programistów chcących zaktualizować swoją wiedzę do najnowszych wersji tego zestawu technologii, ale również dla tych, którzy przychodzą do .NET i języka C# z innych środowisk i chcieliby rozszerzyć swoje umiejętności oraz zestaw narzędzi programistycznych.

Wszystkie główne punkty tutaj omówione są ilustrowane przykładami, a ważne elementy tych demonstracji są szczegółowo objaśniane, co ułatwia ich dokładne prześledzenie.

Co omawia ta książka

Rozdział 1, *Wewnątrz CLR*, opisuje wewnętrzną strukturę .NET, sposób budowania podzespołów, dostępne narzędzia i zasoby oraz możliwości integracji .NET z systemem operacyjnym.

Rozdział 2, *Najważniejsze pojęcia języka C# i platformy .NET*, obejmuje podstawy języka, jego główne charakterystyki i prawdziwe powody określonego wyglądu pewnych funkcji takich jak delegaty.

Rozdział 3, *Zaawansowane pojęcia języka C# i platformy .NET*, zaczyna od przeglądu wersji 4.0, typowych nowych praktyk dla języka i bibliotek, zwłaszcza związanych z synchronizacją, wykonywaniem wątków i programowaniem dynamicznym. Na

koniec znajdziemy opis wielu nowych aspektów, które pojawiły się w wersjach 6.0 i 7.0, a których celem jest uproszczenie sposobu pisania kodu.

Rozdział 4, *Porównanie różnych typów podejścia do programowania*, zajmuje się dwoma członkami ekosystemu języków .NET: językami F# i TypeScript (zwanymi również językami funkcjonalnymi), które zyskują popularność w środowisku programistów.

Rozdział 5, *Mechanizm refleksji i programowanie dynamiczne*, omawia możliwości programu .NET związane z badaniem, analizą i modyfikowaniem swojej własnej struktury i działania, a także sposobami współpracy z innymi programami takimi jak pakiet Office.

Rozdział 6, *Programowanie baz danych SQL*, zajmuje się dostępem do baz danych zbudowanych zgodnie z zasadami modelu relacyjnego, a w szczególności bazami danych SQL. Omawia Entity Framework 6.0 i krótko przypomina ADO.NET.

Rozdział 7, *Programowanie baz danych NoSQL*, omawia nowszy model baz danych zwany bazami danych NoSQL. Wykorzystamy w nim najpopularniejszą tego rodzaju bazę MongoDB i zobaczymy, jak zarządzać nią z poziomu kodu C#.

Rozdział 8, *Programowanie otwarte*, omawia obecny stan programowania otwartego przy użyciu technologii Microsoft i ekosystem programowania z otwartym kodem źródłowym. Zajmiemy się technologiami Node.js, Roselyn, a także TypeScript, choć z nieco innego punktu widzenia.

Rozdział 9, *Architektura*, zajmuje się strukturą aplikacji i dostępnymi narzędziami służącymi do ich konstruowania, takimi jak MSF, dobre praktyki itd.

Rozdział 10, *Wzorce projektowe*, skupia się na jakości kodu i jego strukturze pod względem efektywności, precyzji i łatwości utrzymania. Zajmuje się zasadami SOLID, wzorcami Gang of Four i innymi projektami.

Rozdział 11, *Bezpieczeństwo*, analizuje 10 najważniejszych rekomendacji OWASP dotyczących bezpieczeństwa z punktu widzenia programisty .NET.

Rozdział 12, *Wydajność*, zajmuje się typowymi problemami, jakie może napotkać programista w odniesieniu do wydajności aplikacji oraz tym, jakie techniki i wskazówki mogą służyć do uzyskania elastycznego i dobrze zachowującego się oprogramowania ze specjalnym podkreśleniem wydajności WWW.

Rozdział 13, *Tematy zaawansowane*, omawia interakcję z systemem operacyjnym poprzez tworzenie podklas, usługi platform/invoke, pobieranie danych systemowych przez WMI, programowanie równoległe oraz wprowadzenie do nowych technologii wieloplatformowych .NET Core i ASP.NET Core.

Co jest potrzebne do korzystania z tej książki

Ponieważ ta książka jest poświęcona językowi C# i platformie .NET, głównym wykorzystywanym narzędziem będzie Visual Studio. Można jednak korzystać z różnych jego wersji przy stosowaniu kodu z większości fragmentów tej książki.

Ja korzystałem z wersji Visual Studio 2015 Ultimate Update 3, ale można też korzystać z darmowej edycji Community (również w najnowszej wersji 2017) w przypadku ponad 90% omawianej treści. Innymi dostępnymi opcjami jest też darmowa edycja Visual Studio 2015 Express oraz darmowy i wieloplatformowy program Visual Studio Code.

Ponadto wymagana jest też podstawowa instalacja SQL Server Express 2014 (również darmowa) wraz z SQL Server Management Studio (wersja 2016 będzie działać równie dobrze w odniesieniu do omawianych tutaj tematów).

Na potrzeby rozdziału dotyczącego NoSQL wymagana jest podstawowa instalacja MongoDB.

Do debugowania serwisów WWW dobrze jest zainstalować przeglądarkę Chrome Canary lub Firefox Developer Edition, ponieważ mają rozbudowane funkcjonalności przeznaczone dla programistów.

Inne narzędzia można zainstalować korzystając z opcji **Extensions and Updates** w menu **Tools** w różnych wersjach Visual Studio.

W kilku przypadkach będą przydatne dodatkowe narzędzia, które można pobrać z serwisów internetowych wskazanych w tej książce, choć nie są one absolutnie wymagane do pełnego zrozumienia zawartości tej książki.

Dla kogo jest ta książka

Ta książka została napisana wyłącznie dla programistów .NET. Tworzącym aplikacje C# dla swoich klientów, w pracy lub w domu, książka ta pomoże rozwinąć umiejętności potrzebne do tworzenia nowoczesnych, wspaniałych i wydajnych aplikacji w języku C#.

Nie jest wymagana żadna wiedza dotycząca C# 6/7 albo .NET 4.6 – wszystkie najnowsze funkcje zostaną omówione, aby pomóc w pisaniu aplikacji na różne platformy. Trzeba dobrze znać Visual Studio, choć omówione też zostaną wszystkie nowe funkcje w Visual Studio 2015.

Konwencje

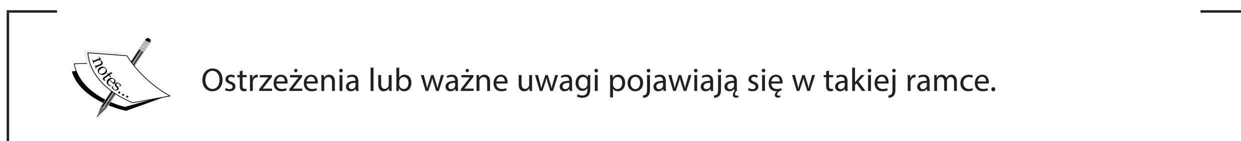
W tej książce znajdziemy wiele stylów tekstu, którymi oznaczane są różne rodzaje informacji. Oto kilka przykładów tych stylów oraz wyjaśnienie ich znaczenia.

Elementy kodu w tekście, nazwy tabel bazodanowych, nazwy folderów, nazwy plików, rozszerzenia plików, nazwy ścieżek dostępu, dane wpisywane przez użytkowników oraz identyfikatory Twitter są przedstawiane następująco: „Korzystamy z metody `ForEach`, która otrzymuje argument `Action` będący delegatem”.

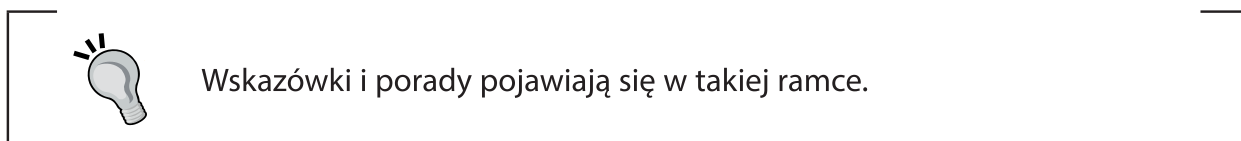
Blok kodu jest formatowany następująco:

```
static void GenerateStrings()
{
  string initialString = "Initial Data-";
  for (int i = 0; i < 5000; i++)
  {
    initialString += "-More data-";
  }
  Console.WriteLine("Strings generated");
}
```

Nowe terminy i *ważne słowa* są wyróżnione bezszeryfową kursywą. Słowa, które są widoczne na ekranie, na przykład w menu lub oknach dialogowych, pojawiają się w tekście w następującej formie: „Na karcie **Memory Usage** możemy zobaczyć aktualny stan tego, co się dzieje”.



Ostrzeżenia lub ważne uwagi pojawiają się w takiej ramce.



Wskazówki i porady pojawiają się w takiej ramce.

Informacje od czytelników

Informacje zwrotne od naszych czytelników są zawsze mile widziane. Prosimy o opinie na temat tej książki – co się w niej podoba, a co nie. Informacje od czytelników są dla nas ważne, żebyśmy mogli przygotowywać najbardziej pożądane tytuły.

W celu przekazania ogólnych uwag, wystarczy wysłać e-maila na adres *feedback@packtpub.com* podając tytuł książki w temacie wiadomości.

Jeśli ktoś jest ekspertem w jakiejś dziedzinie i chciałby napisać lub uczestniczyć w pracach nad książką, może zajrzeć do naszego przewodnika dla autorów pod adresem *www.packtpub.com/authors*.

Obsługa klienta

Dla dumnego posiadacza książki wydawnictwa Packt mamy wiele rzeczy pomocnych w maksymalnym skorzystaniu ze swojego zakupu.

Pobieranie kodu przykładowego

Pliki z przykładowym kodem dla tej książki można pobrać ze swojego konta pod adresem *http://www.packtpub.com*. Jeśli książka została zakupiona gdzie indziej, można odwiedzić adres *http://www.packtpub.com/support*, aby się zarejestrować i otrzymać pliki pocztą elektroniczną.

Pliki z kodem można pobrać korzystając z następujących kroków:

1. Zalogować się lub zarejestrować w naszym serwisie korzystając ze swojego adresu e-mail i hasła.
2. Wskazać kursorem myszy kartę **SUPPORT** u góry.
3. Kliknąć **Code Downloads & Errata**.
4. Wpisać nazwę książki w polu **Search**.
5. Wybrać książkę, dla której poszukiwane są pliki z kodem.
6. Wskazać w rozwijanym menu, gdzie została zakupiona książka.
7. Kliknąć **Code Download**.

Można też pobrać pliki z kodem klikając przycisk **Code Files** na stronie WWW książki w serwisie Packt Publishing. Dostęp do tej strony można uzyskać wpisując nazwę książki w polu **Search**. Trzeba być zalogowanym do swojego konta Packt.

Po pobraniu pliku można go rozpakować korzystając z najnowszych wersji oprogramowania:

- WinRAR / 7-Zip dla Windows
- Zipeg / iZip / UnRarX dla Mac
- 7-Zip / PeaZip dla systemu Linux

Pakiet z kodem dla tej książki jest też dostępny w serwisie GitHub pod adresem *https://github.com/PacktPublishing/Mastering-C-Sharp-and-.NET-Framework*. Również inne

pakiety kodu z naszego bogatego katalogu książek i filmów są dostępne pod adresem *https://github.com/PacktPublishing/*. Warto je sprawdzić!

Errata

Chociaż podjęliśmy wszelkie starania, aby zapewnić poprawność treści tej książki, błędy czasem się zdarzają. W razie znalezienia błędu w jednej z naszych książek – na przykład błędu w tekście lub w kodzie – bylibyśmy wdzięczni za zgłoszenie nam go. Dzięki temu możemy oszczędzić innym czytelnikom kłopotów i poprawić kolejne wersje tej książki. Wszelkie poprawki prosimy zgłaszać pod adresem *http://www.packtpub.com/submit-errata*, wybierając daną książkę, klikając łącze **Errata Submission Form** i wprowadzając szczegóły błędu. Po zatwierdzeniu zgłoszenia, zostanie ono przyjęte i opublikowane na naszej stronie WWW lub dodane do listy istniejących poprawek w części Errata dla danego tytułu.

Wcześniej przesłane poprawki można znaleźć pod adresem *https://www.packtpub.com/books/content/support* wpisując tytuł (oryginalny, tzn. angielski) książki w polu wyszukiwania. Żądane informacje pojawią się w części **Errata**.

Piractwo

Internetowe piractwo materiałów chronionych prawem autorskim jest stałym problemem. W wydawnictwie Packt bardzo poważnie traktujemy ochronę naszych praw autorskich i licencji. W razie natrafienia w Internecie na nielegalne egzemplarze naszych prac w jakiejkolwiek formie prosimy o podanie nam adresu lub nazwy strony WWW, abyśmy mogli podjąć stosowne działania.

Prosimy o kontakt pod adresem *copyright@packtpub.com* z podaniem łącza do materiału podejrzanego o piractwo.

Dziękujemy za pomoc w ochronie naszych autorów i naszych możliwości dostarczania cennych treści.

Pytania

W przypadku problemów z dowolnym aspektem tej książki prosimy o kontakt pod adresem *questions@packtpub.com*, a postaramy się pomóc w miarę naszych możliwości.

1

Wewnątrz CLR

Ponieważ CLR (Common Language Runtime – wspólne środowisko uruchomieniowe języka) jest tylko ogólną nazwą dla różnych narzędzi i programów opartych na dobrze znanych i przyjętych zasadach programowania, zaczniemy od przeglądnięcia kilku najważniejszych pojęć związanych z programowaniem, które często przyjmujemy za rzecz oczywistą. Aby więc nadać sprawom odpowiedni kontekst, rozdział ten omówi najważniejsze pojęcia związane z motywacjami do utworzenia .NET, jak platforma ta integruje się z systemem operacyjnym Windows i co sprawia, że CLR jest tak świetną platformą uruchomieniową.

W skrócie, rozdział ten omawia następujące zagadnienia:

- Krótki, ale starannie dobrany słownik typowych pojęć i terminów wykorzystywanych w programowaniu ogólnym i związanym z .NET.
- Szybki przegląd celów stworzenia platformy .NET i głównych zasad architektonicznych związanych z jej budową.
- Wyjaśnienie każdej z głównych części składających się na platformę uruchomieniową CLR, jej narzędzi i sposobu ich działania.
- Podstawowe podejście do złożoności algorytmów i sposobów jej mierzenia.
- Wybraną listę najbardziej wyróżniających się charakterystyk związanych z CLR, które pojawiły się w najnowszych wersjach.

Uwagi dotyczące kilku ważnych pojęć komputerowych

Przypomnijmy sobie kilka ważnych pojęć używanych w dziedzinie tworzenia oprogramowania, z którymi mamy często do czynienia przy programowaniu dla platformy .NET.

Kontekst

Jak stwierdza Wikipedia:

> *W informatyce kontekst zadania jest minimalnym zbiorem danych wykorzystywanym przez zadanie (mogące być procesem lub wątkiem), który należy zapisać, aby można było przerwać zadanie w danym momencie, a później wznowić to zadanie w punkcie przerwania w dowolnym przyszłym momencie.*

Innymi słowy, kontekst jest pojęciem związanym z danymi obsługiwanymi przez wątek. Dane takie są w miarę zapotrzebowania zapisywane i wydobywane przez system.

Praktyczne podejścia do tego pojęcia obejmują scenariusze związane z zapytaniami i odpowiedziami HTTP oraz bazami danych, w których kontekst odgrywa bardzo ważną rolę.

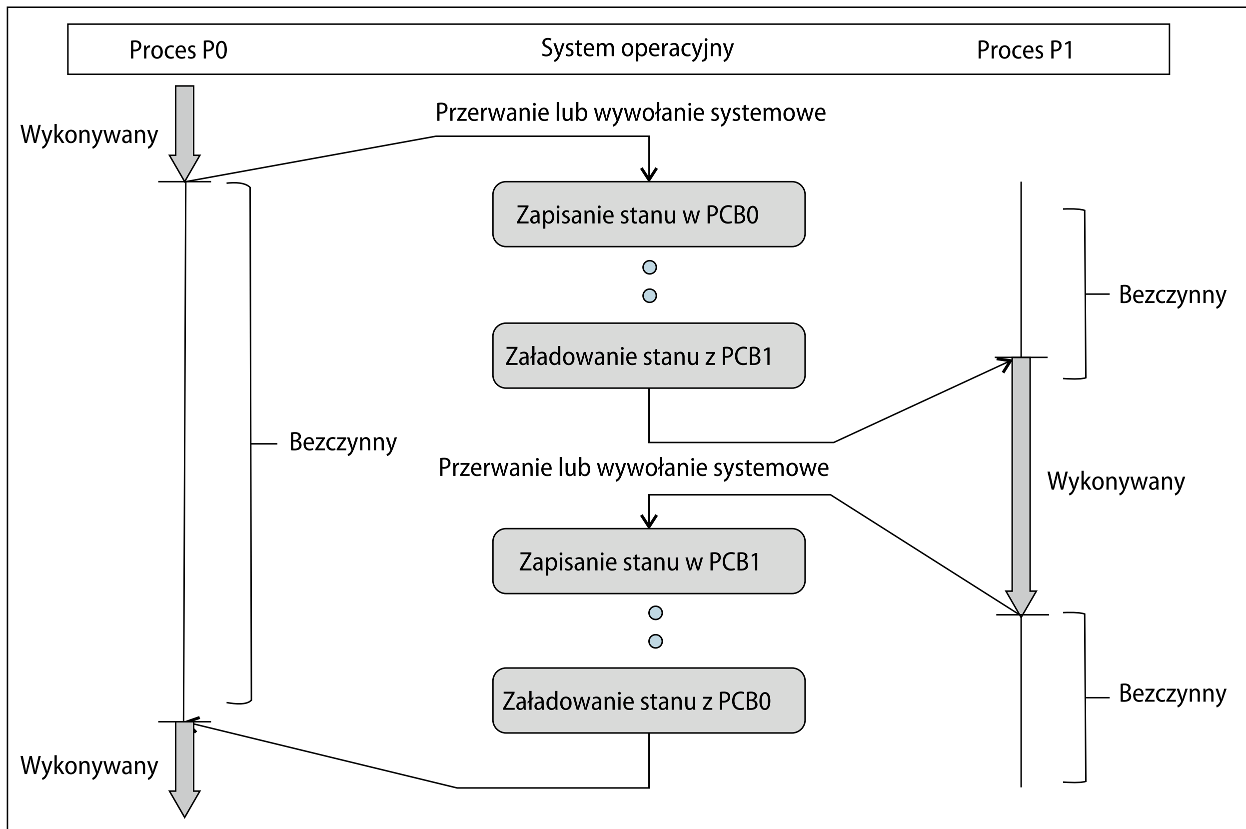
Model wykonywania wielu zadań w systemie operacyjnym

Procesor komputera jest w stanie zarządzać wieloma procesami w danym okresie czasu. Jak wspominaliśmy, można to osiągnąć przez zapisywanie i przywracanie (w bardzo szybki sposób) kontekstu wykonywania przy pomocy techniki zwanej przełączaniem kontekstu.

Gdy wątek przestaje być wykonywany, mówimy, że jest w stanie *bezczynności* (*idle*). Ta kategoryzacja może być przydatna podczas analizowania wykonywania procesów przy pomocy narzędzi, które mogą wyodrębniać wątki będące w stanie *bezczynności* (patrz ilustracja na sąsiedniej stronie).

Typy kontekstu

W niektórych językach, takich jak C# mamy również do czynienia z pojęciem bezpiecznego kontekstu. W pewnym sensie jest to związane z tak zwanym bezpieczeństwem wątków.



Bezpieczeństwo wątków

Mówi się, że fragment kodu jest wątkowo bezpieczny, jeśli jedynie manipuluje wspólnymi strukturami danych w sposób gwarantujący bezpieczne wykonywanie wielu wątków w tym samym czasie. Istnieją różne strategie używane do tworzenia bezpiecznych wątkowo struktur danych, a platforma .NET zwraca szczególną uwagę na to pojęcie i jego implementacje.

W istocie oficjalna dokumentacja MSDN zawiera w dolnej części opisu znacznej większości typów określenie *ten typ jest wątkowo bezpieczny*.

Stan

Stan oprogramowania komputerowego jest terminem technicznym obejmującym całość przechowywanej w danym momencie czasu informacji, do której dany program ma dostęp. Wyjście programu komputerowego w danym momencie jest całkowicie określane przez jego aktualne wejścia oraz jego stan. Bardzo ważnym wariantem tego pojęcia jest stan programu.

Stan programu

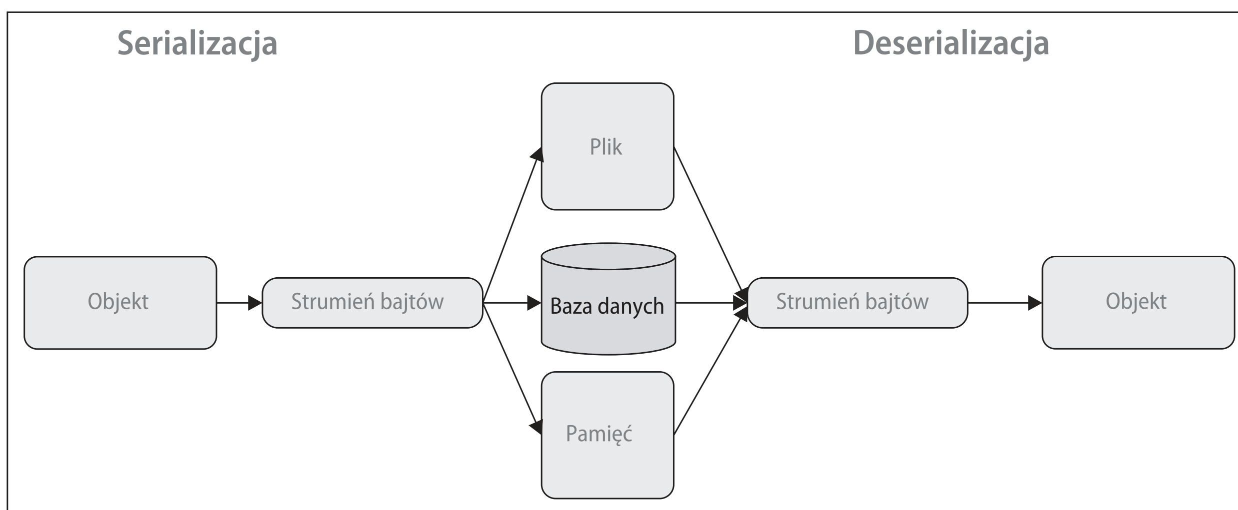
To pojęcie jest szczególnie ważne i ma kilka znaczeń. Wiemy, że program komputerowy przechowuje dane w zmiennych, które są po prostu nazwanymi miejscami w pamięci komputera. Zawartość tych miejsc w pamięci w danym momencie wykonywania programu jest zwana stanem programu.

W językach zorientowanych obiektowo mówi się, że klasa definiuje swój stan poprzez pola, a wartości tych pól w danym momencie wykonywania określają stan danego obiektu. Choć nie jest to obowiązkowe, to dobrą praktyką w programowaniu zorientowanym obiektowo jest, aby jedynym celem metod klasy było zachowywanie spójności i logiki jej stanu.

Ponadto typologia języków programowania określa dwie kategorie: programowanie imperatywne i deklaratywne. Języki C# lub Java są przykładami tej pierwszej kategorii, a HTML ma typową składnię deklaratywną. W programowaniu imperatywnym instrukcje mają tendencję do zmieniania stanu programu, natomiast w podejściu deklaratywnym języki jedynie wskazują na pożądany rezultat bez określania, jak dany silnik zajmie się uzyskaniem tych wyników.

Serializacja

Serializacja jest procesem tłumaczenia struktur danych lub stanu obiektu na format, który można zapisać (na przykład w pliku lub w buforze pamięci) albo przesłać przez połączenie sieciowe, a później zrekonstruować w tym samym lub innym środowisku komputerowym.



Zwykle mówimy, że serializacja obiektu oznacza przekonwertowanie jego stanu na strumień bajtów w taki sposób, że ten strumień bajtów można przekonwertować z powrotem na kopię tego obiektu. Niezależnie od wcześniejszych formatów

(w tym binarnych) pojawiły się też i przyjęły popularne formaty tekstowe takie jak XML i JSON.

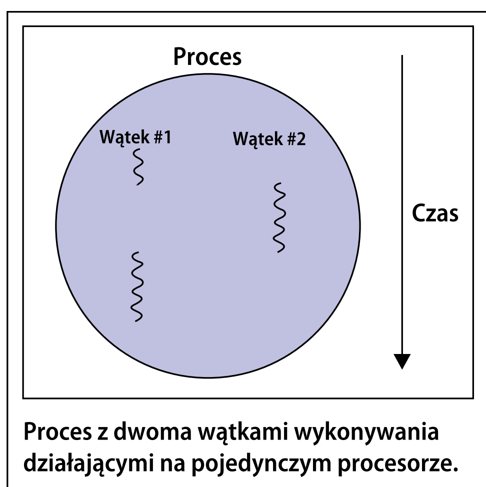
Proces

System operacyjny dzieli wykonywane operacje pomiędzy kilka jednostek funkcjonalnych. Dokonuje się to przez przydzielanie różnych obszarów pamięci dla każdej jednostki wykonawczej. Ważne jest rozróżnienie pomiędzy procesami a wątkami.

Każdy proces otrzymuje od systemu operacyjnego zestaw zasobów, co w przypadku systemu Windows oznacza, że proces dostanie swoją własną wirtualną przestrzeń adresową, która będzie odpowiednio przydzielana i zarządzana. Gdy system Windows inicjuje proces, to w istocie ustanawia kontekst wykonywania, który obejmuje blok środowiska procesu zwany w skrócie PEB oraz strukturę danych. Należy jednak wyjaśnić: system operacyjny nie wykonuje procesów; jedynie ustanawia kontekst wykonywania.

Wątek

Wątek jest jednostką funkcjonalną procesu. To jest właśnie ten element, który jest wykonywany przez system operacyjny. Pojedynczy proces może mieć kilka wątków wykonywania, co zdarza się bardzo często. Każdy wątek ma swoją własną przestrzeń adresową w ramach zasobów przydzielonych wcześniej podczas tworzenia procesu. Zasoby te są wspólnie wykorzystywane przez wszystkie wątki związane z danym procesem:



Proces z dwoma wątkami wykonywania działającymi na pojedynczym procesorze.

Ważne jest, aby pamiętać, że wątek należy jedynie do pojedynczego procesu, a więc ma dostęp tylko do zasobów zdefiniowanych przez ten proces. Przy korzystaniu z narzędzi, które zaraz omówimy, możemy oglądać wiele wątków wykonywanych

współbieżnie (co oznacza, że zaczynają one działanie w niezależny sposób) i korzystających ze wspólnych zasobów takich jak pamięć i dane.

Różne procesy nie współdzielą tych zasobów. W szczególności wątki procesu korzystają wspólnie z jego instrukcji (wykonywalnego kodu) oraz jego kontekstu (wartości jego zmiennych w danym momencie).

Języki programowania, takie jak języki platformy .NET, Java lub Python udostępniają programiście obsługę wątków, ukrywając przy tym specyficzne dla poszczególnych platform różnice implementacyjne.

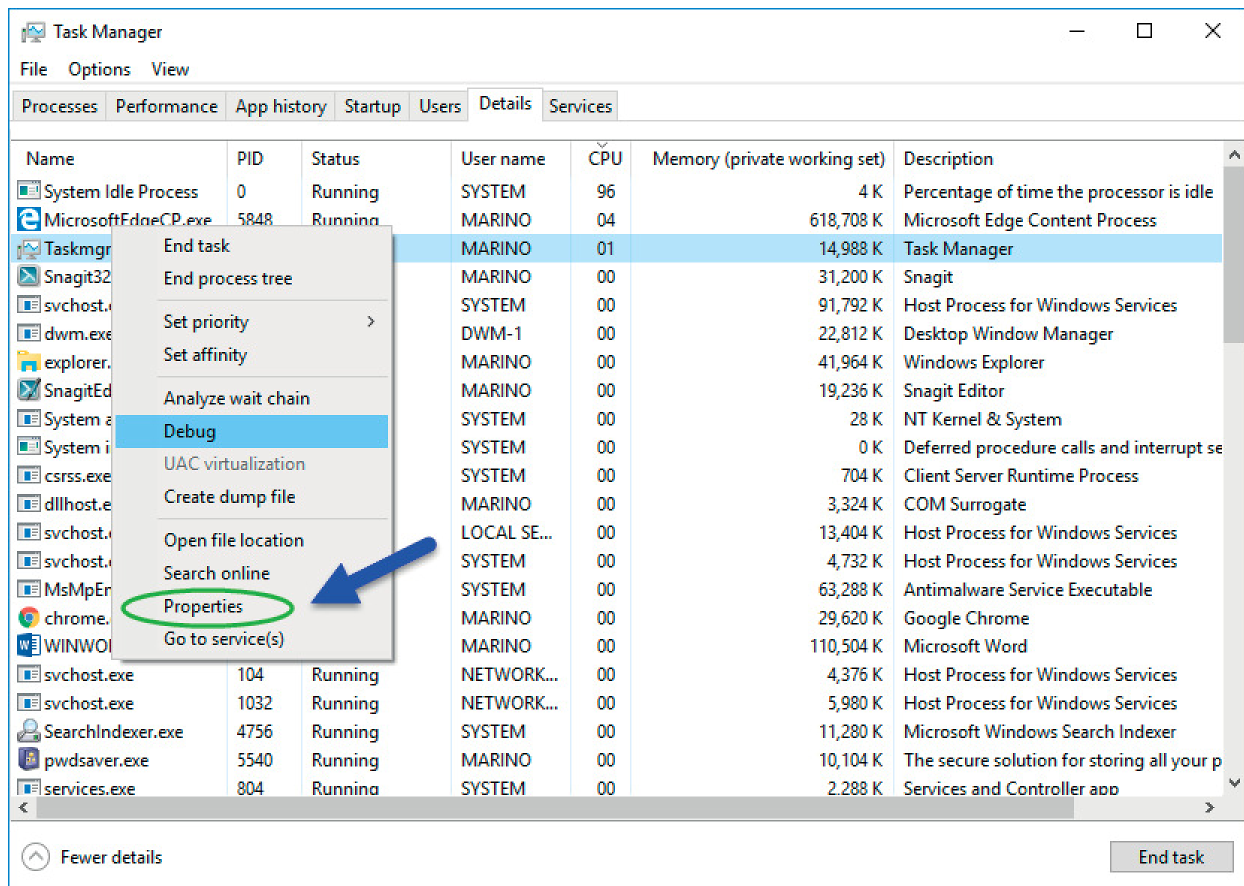
Komunikacja pomiędzy wątkami jest możliwa poprzez wspólny zestaw zasobów inicjowanych przez utworzenie procesu.

Oczywiście na temat tych dwóch pojęć napisano o wiele więcej, co wykracza poza zakres tej książki (po więcej szczegółów można sięgnąć do Wikipedii: *https://en.wikipedia.org/wiki/Thread_(computing)*). System zapewnia nam mechanizmy sprawdzania wykonywania dowolnego procesu oraz sprawdzania, które wątki są wykonywane.

Jeśli ktoś jest ciekawy lub chciałby sprawdzić, czy coś idzie nie tak, polecam dwa główne narzędzia: Menedżer zadań (zawarty w systemie operacyjnym) oraz jedno z narzędzi z zestawu ponad 50 programów zaprojektowanych przez wybitnego inżyniera Marka Russinowitcha i dostępnych za darmo.

Niektóre z nich mają interfejs okienkowy, a inne są narzędziami konsolowymi, ale wszystkie są niezwykle zoptymalizowanymi i konfigurowalnymi programami do monitorowania i sterowania wewnętrznymi aspektami systemu operacyjnego w danym momencie. Dostępne są za darmo pod adresem *https://technet.microsoft.com/en-us/sysinternals/bb545021.aspx*.

Jeśli ktoś nie chce instalować dodatkowego oprogramowania, może otworzyć program **Menedżer zadań** (wystarczy kliknąć prawym przyciskiem myszy pasek narzędzi, aby uzyskać do niego dostęp) i wybrać kartę **Properties** (**Szczegóły**). Zobaczymy na niej bardziej dokładny opis każdego procesu, procentowe użycie procesora przez każdy proces, pamięć przydzieloną dla każdego procesu itd. Można nawet kliknąć jeden z procesów prawym przyciskiem myszy i zobaczyć menu kontekstowe oferujące kilka możliwości, między innymi opcję uruchomienia nowego okna dialogowego pokazującego kilka właściwości związanych z danym procesem (ilustracja na sąsiedniej stronie).



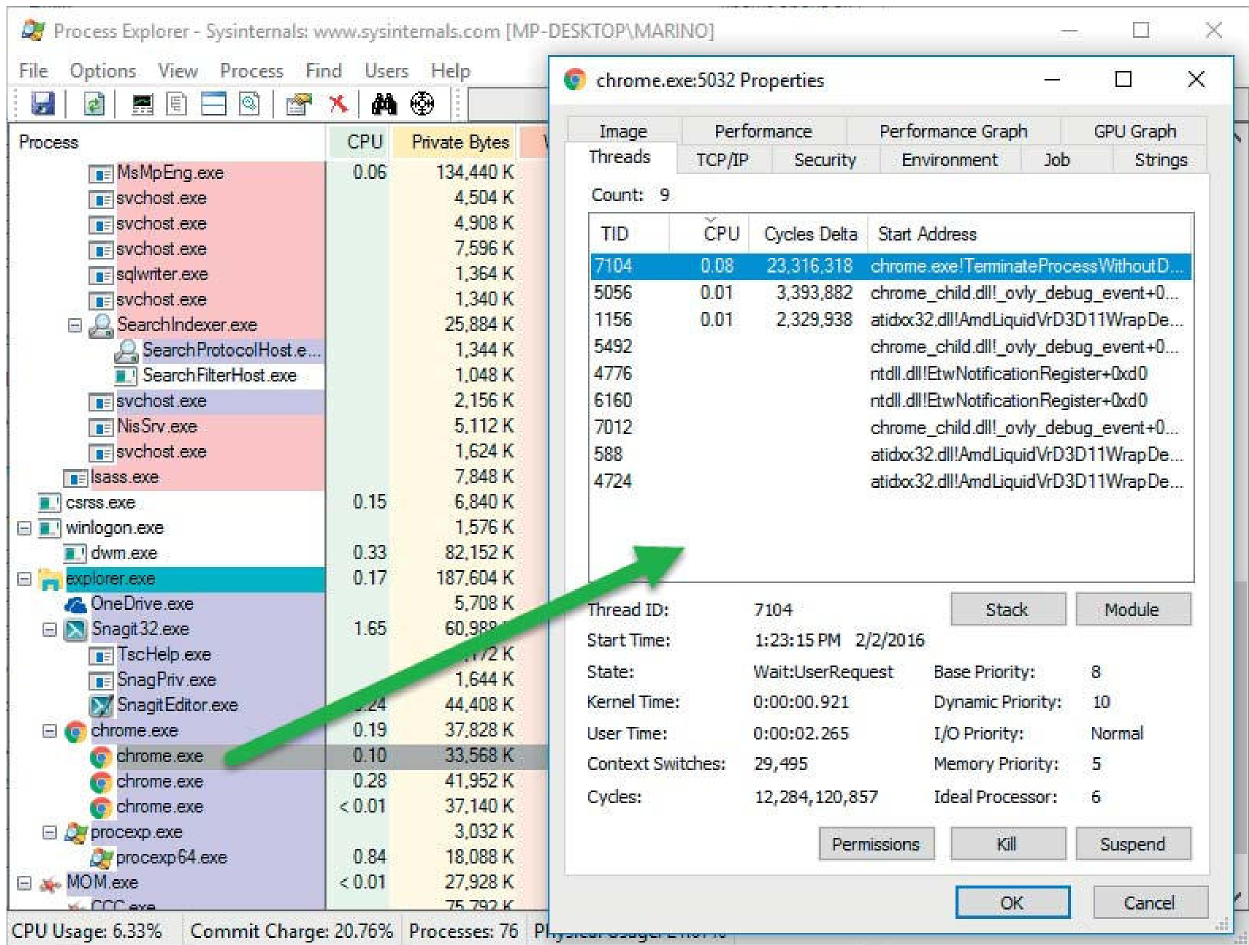
SysInternals

Jeśli ktoś chce naprawdę dowiedzieć się, jak zachowuje się dany proces, to należy skorzystać z narzędzi SysInternals. Po skorzystaniu z łącza wskazanego wcześniej zobaczymy narzędzia specjalnie dedykowane do zadań związanych z procesami. Jest kilka opcji do wyboru, ale najbardziej rozbudowane są narzędzia **Process Explorer** i **Process Monitor**.

Programy **Process Explorer** i **Process Monitor** nie wymagają instalacji (są napisane w C++), więc można je wykonywać bezpośrednio z dowolnego urządzenia na platformie Windows, a nawet ze strony internetowej .

Jeśli na przykład uruchomimy program **Process Explorer**, zobaczymy rozbudowane okno pokazujące szczegóły wszystkich procesów aktualnie aktywnych w systemie.

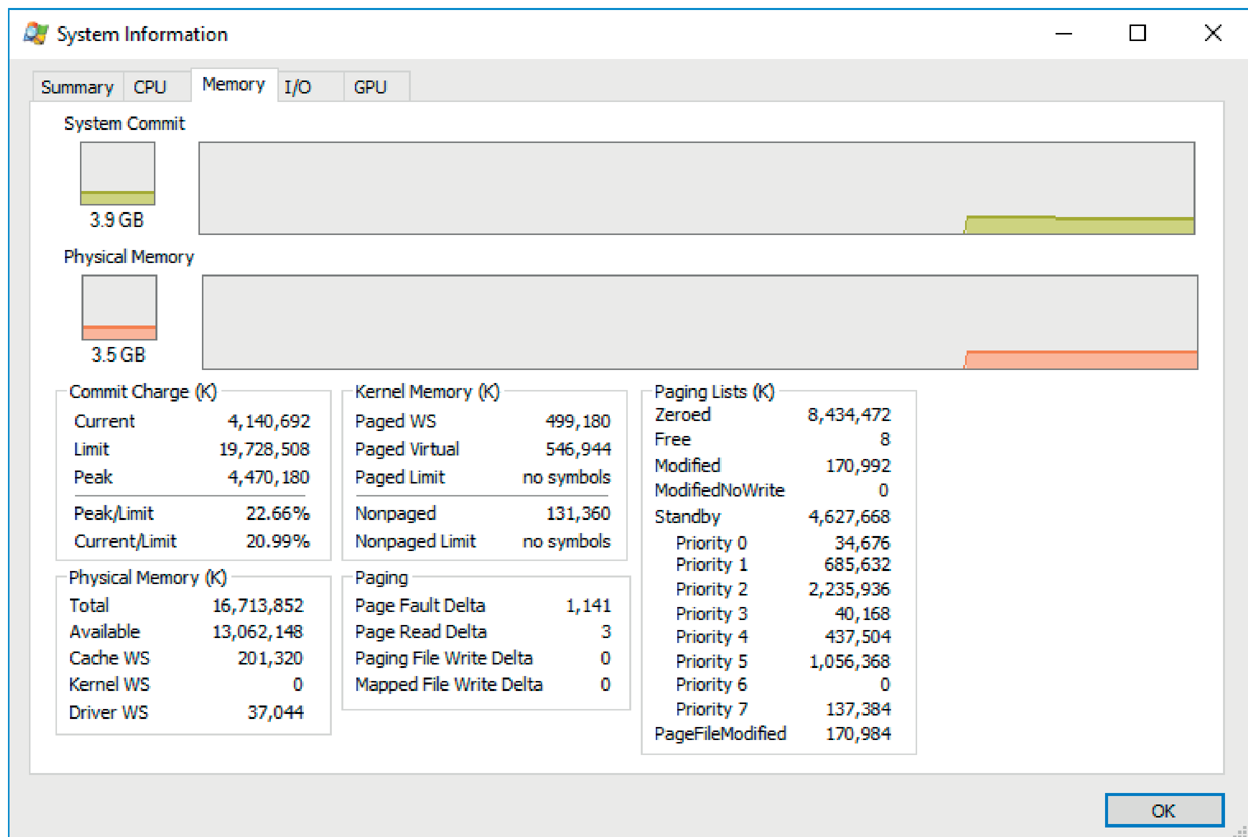
Przy pomocy narzędzia **Process Explorer** możemy dowiedzieć się, jakie pliki, klucze rejestru i inne obiekty zostały otwarte przez procesy, a także jakie załadowały one biblioteki DLL, kto jest właścicielem każdego procesu itd. Widoczny jest każdy wątek i szczegółowe informacje dostępne w bardzo intuicyjnym interfejsie użytkownika (pokazany na następnej stronie).



Narzędzie to jest też bardzo przydatne do sprawdzania ogólnego zachowania systemu w czasie rzeczywistym, ponieważ graficznie przedstawia m.in. użycie procesora, wejścia/wyjścia, pamięci, jak pokazano na zrzucie ekranu na sąsiedniej stronie.

W podobny sposób **Process Monitor** skupia się na monitorowaniu systemu plików, rejestru oraz wszystkich procesów i wątków w czasie rzeczywistym, gdyż w istocie jest połączeniem dwóch wcześniejszych narzędzi: *FileMon* (*File Monitor*) i *RegMon* (*Registry Monitor*), które nie są już dostępne.

Jeśli sięgniemy po Process Monitor, zobaczymy kilka informacji, które są też zawarte w Process Explorer, ale poza tym wiele informacji dostarczanych wyłącznie przez Process Monitor.



Pamięć statyczna i dynamiczna

Gdy program rozpoczyna wykonywanie, system operacyjny przypisuje do niego proces, korzystając z harmonogramu: metody, w której określona praca jest przypisywana do zasobów, które mają ją wykonać. To oznacza, że zasoby są przypisywane do procesu, co wiąże się z przydzieleniem pamięci.

Jak zobaczymy, istnieją przede wszystkim dwa typy przydzielania pamięci:

- Pamięć stała (połączona ze stosem), określana w czasie kompilacji. Zmienne lokalne są deklarowane i wykorzystywane na stosie. Jest to ciągły blok pamięci alokowany podczas początkowego przydzielania zasobów dla procesu. Mechanizm przydzielania jest bardzo szybki (choć dostęp nie tak bardzo).
- Innym rodzajem jest pamięć dynamiczna (sterta), która może rosnąć w miarę wymagań programu i jest przypisywana w trakcie jego działania. W tym miejscu alokowane są zmienne obiektowe (te, które wskazują na wystąpienia klas, czyli obiekty).

Zwykle w przypadku pierwszego z tych typów pamięci obliczenia są dokonywane w czasie kompilacji, ponieważ kompilator wie, ile pamięci będzie potrzebne

do alokacji zadeklarowanych zmiennych w zależności od ich typów (`int`, `double` itd.). Są one deklarowane wewnątrz funkcji przy pomocy składni takiej jak `int x = 1;`

Drugi typ wymaga operatora `new` do wywołania. Powiedzmy, że mamy w kodzie klasę o nazwie `Book` i tworzymy wystąpienie tej klasy `Book` przy pomocy wyrażenia następującego typu:

```
Book myBook = new Book();
```

Nakazuje to środowisku uruchomieniowemu przydzielenie odpowiedniej ilości miejsca na stercie do przechowywania wystąpienia tego typu razem z jego polami; stan klasy jest alokowany na stercie. Oznacza to, że cały stan programu będzie przechowywany w różnych miejscach w pamięci (i opcjonalnie na dysku).

Oczywiście trzeba wziąć pod uwagę dodatkowe aspekty, co omówimy w dalszej części rozdziału pod nagłówkiem *Stos i sterta*. Na szczęście środowisko programistyczne pozwala nam obserwować i analizować wszystkie te aspekty podczas debugowania programu.

Odśmiecanie pamięci

Odśmiecanie pamięci (*GC — Garbage Collection*) jest formą automatycznego zarządzania pamięcią. Odśmiecanie pamięci w .NET próbuje odzyskać pamięć zajmowaną przez obiekty, które nie są już używane przez program. Wracając do wcześniejszej deklaracji obiektu klasy `Book`, gdy nie ma żadnych odwołań do tego obiektu `Book` na stosie, GC odzyska to miejsce dla systemu zwalniając pamięć (jest to w istocie nieco bardziej skomplikowane i omówimy to dokładniej w dalszej części rozdziału, gdy będziemy mówić o zarządzaniu pamięcią).

Warto zauważyć, że odśmiecanie pamięci nie jest czymś wyjątkowym tylko dla platformy .NET. Faktycznie można je spotkać na różnych platformach i w różnych programach nawet mając do czynienia z przeglądarkami. Na przykład aktualne silniki języka JavaScript, takie jak V8 w Chrome czy Chakra firmy Microsoft (a także inne) również wykorzystują mechanizm odśmiecania pamięci.

Przetwarzanie współbieżne

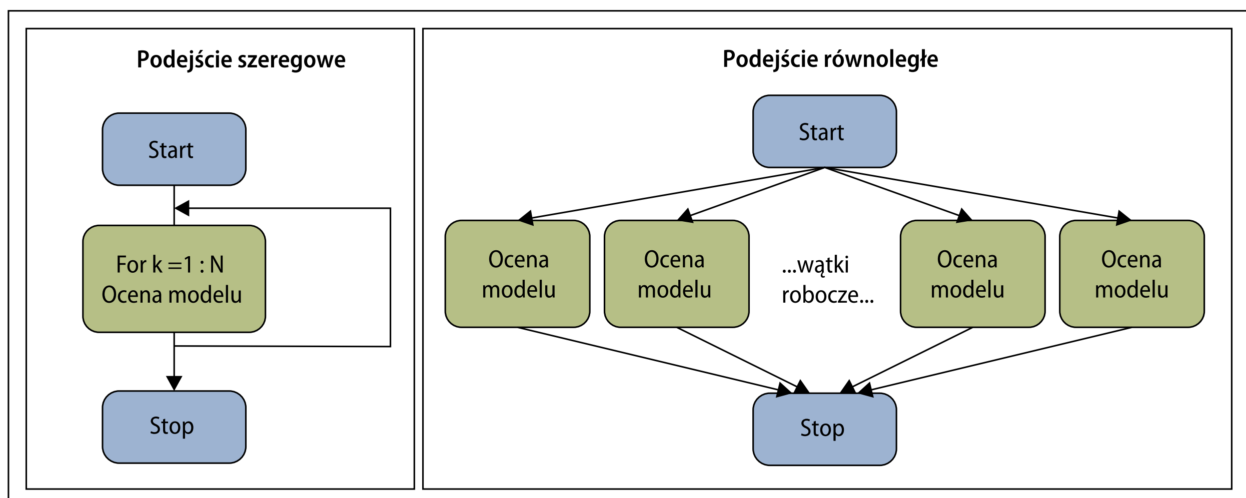
Przetwarzanie współbieżne jest obecnie bardzo popularnym pojęciem i zetkniemy się z nim kilkukrotnie w tej książce. Oficjalna definicja w Wikipedii (*https://en.wikipedia.org/wiki/Concurrent_computing*) mówi:

> *Przetwarzanie współbieżne (ang. concurrent computing) – przetwarzanie oparte na współistnieniu wielu wątków lub procesów, operujących na współdzielonych danych. Wątki uruchomione na tym samym procesorze są przełączane*

w krótkich przedziałach czasu, co sprawia wrażenie, że wykonują się równolegle. W przypadku procesorów wielordzeniowych lub wielowątkowych, możliwe jest faktycznie współbieżne przetwarzanie. Tego rodzaju przetwarzanie jest też możliwe w architekturach wieloprocesorowych. W takiej sytuacji wydajność poszczególnych wątków zasadniczo nie jest degradowana przez inne wątki, z wyjątkiem sytuacji, kiedy wątki muszą rywalizować o wspólne zasoby, np. przepustowość magistral i urządzeń lub czas procesora, lub muszą synchronizować swoją pracę.

Przetwarzanie równoległe

Przetwarzanie równoległe jest typem przetwarzania, w którym wiele obliczeń jest przeprowadzanych jednocześnie, wykorzystując zasadę, że większe problemy można często podzielić na mniejsze, które da się rozwiązywać w tym samym czasie. Platforma .NET oferuje kilka wariantów tego typu przetwarzania, które omówimy w kilku następnych rozdziałach:



Programowanie imperatywne

Programowanie imperatywne jest paradygmatem oprogramowania, który opisuje obliczenia przy pomocy stanu programu. C#, JavaScript, Java lub C++ są typowymi przykładami języków imperatywnych.

Programowanie deklaratywne

W przeciwieństwie do programowania imperatywnego, języki uważane za deklaratywne jedynie opisują pożądane wyniki nie podając bezpośrednio poleceń lub kroków,

które należy wykonać. Wiele języków znacznikowych, takich jak HTML, XAML lub XSLT należy do tej kategorii.

Ewolucja .NET

Do momentu pojawienia się .NET, ekosystem programowania Microsoft był opanowany przez kilka klasycznych języków, których typowymi przykładami były Visual Basic i C++ (z biblioteką Microsoft Foundation Classes).

Microsoft Foundation Classes (*MFC*) jest biblioteką opakowującą elementy interfejsu Windows API w formie klas C++, włączając w to funkcjonalności umożliwiające im wykorzystanie domyślnej platformy aplikacyjnej. Definiuje klasy dla wielu obiektów Windows zarządzanych przez dojścia, a także predefiniowane kontrolki okienkowe. Biblioteka ta została wprowadzona w roku 1992 wraz z kompilatorem C/C++ 7.0 na użytek 16-bitowych wersji Windows jako niezwykle cienkie, obiektowo zorientowane opakowanie dla interfejsu Windows API w formie klas C++.

Wielkie zmiany zaproponowane przez .NET wprowadziły jednak zupełnie inne podejście do modelu komponentów. Aż do roku 2001, gdy oficjalnie pojawiła się platforma .NET, takim modelem komponentów był *COM* (*Component Object Model*), wprowadzony przez firmę Microsoft w roku 1993. COM stanowi podstawę dla kilku innych technologii i platform Microsoft, w tym OLE, automatyzacji OLE, ActiveX, COM+, DCOM, powłoki Windows, DirectX, *UMDF* (*User-Mode Driver Framework*) oraz środowiska uruchomieniowego Windows.

Platforma programowania sterowników urządzeń (Windows Driver Development Kit) wprowadzona została po raz pierwszy wraz z systemem operacyjnym Windows Vista. Umożliwia tworzenie sterowników dla pewnych klas urządzeń.

COM ma konkurenta w postaci innej specyfikacji o nazwie *CORBA* (*Common Object Request Broker Architecture*), która jest standardem zdefiniowanym przez *Object Management Group* (*OMG*) i zaprojektowanym w celu umożliwienia komunikacji pomiędzy systemami wdrażanymi na różnych platformach. CORBA umożliwia współpracę pomiędzy systemami działającymi na różnych systemach operacyjnych, w różnych językach programowania i na różnym sprzęcie komputerowym. W swoim cyklu życia była mocno krytykowana zwłaszcza ze względu na słabe implementacje standardu.

.NET jako reakcja na świat języka Java

W roku 1995 powstał nowy model mający zastąpić COM i niepożądane efekty z nim związane, zwłaszcza wersje i wykorzystanie Rejestru Windows, od którego technologia COM była zależna i w którym definiowała dostępne interfejsy lub kontrakty. Uszkodzenie lub zmodyfikowanie fragmentu rejestru mogło spowodować, że dany składnik nie był dostępny w czasie wykonywania programu. Do instalowania aplikacji wymagane były zwiększone uprawnienia, ponieważ Rejestr Windows jest wrażliwą częścią systemu.

Rok później różne działy firmy Microsoft zaczęły kontaktować się z najbardziej uznanymi inżynierami oprogramowania, a te kontakty pozostały aktywne przez wiele lat. Należeli do nich architekci oprogramowania, tacy jak Anders Hejlsberg (który został głównym autorem języka C# i głównym architektem platformy .NET), Jean Paoli (jeden z sygnatariuszy standardu XML i wcześniejszy ideolog technologii AJAX), Don Box (który uczestniczył w utworzeniu technologii SOAP i XML Schemas), Stan Lippman (jeden z ojców języka C++), Don Syme (architekt typów ogólnych i główny autor języka F#) oraz wielu innych.

Celem tego projektu było utworzenie nowej platformy wykonawczej wolnej od zastrzeżeń związanych z COM i mogącej wykorzystywać zbiór języków do wykonywania kodu w sposób bezpieczny i łatwy do rozszerzania. Nowa platforma miała umożliwiać programowanie i integrowanie nowego świata usług WWW (które właśnie się pojawiły) opartych na XML i innych technologiach. Pierwszą nazwą nowej propozycji były usługi Windows nowej generacji – *Next Generation Windows Services* (*NGWS*).

Pod koniec roku 2000 zostały wypuszczone pierwsze wersje beta platformy .NET, a pierwsza oficjalna wersja pojawiła się 13 lutego 2002. Od tego czasu nowe wersje .NET były zawsze powiązane z nowymi wersjami środowiska programistycznego (Visual Studio). Obecna wersja klasycznej platformy .NET Framework w momencie pisania tej książki nosi numer 4.6.1, a jej szczegółami zajmiemy się w dalszej części tego rozdziału.

W roku 2015 po raz pierwszy pojawiła się alternatywna wersja .NET. Na konferencji `//BUILD/` firma Microsoft ogłosiła utworzenie i dostępność innej wersji .NET o nazwie .NET Core.

Ruch otwartego oprogramowania i .NET Core

Po części pomysł na .NET Core wziął się z głębokich zmian w obecnym sposobie tworzenia oprogramowania w Redmond. Gdy Satya Nadella przejął funkcję CEO w firmie Microsoft, nowym hasłem przewodnim stało się: *przede wszystkim oprogramowanie mobilne i chmura obliczeniowa*. Microsoft zaczął się też określać jako *firma tworząca oprogramowanie i usługi*.

Oznaczało to przyjęcie zasad otwartego oprogramowania ze wszystkimi tego konsekwencjami. W rezultacie duża część platformy .NET została otwarta dla społeczności i niektórzy twierdzą, że ten ruch będzie trwał aż do otwarcia całej platformy. Oprócz tego drugim celem (kilkukrotnie jasno postawionym na konferencji `//BUILD/`) stało się stworzenie ekosystemu programistycznego pozwalającego każdemu zaprogramować dowolny typ aplikacji na dowolną platformę lub urządzenie. Firma Microsoft zaczęła więc wspierać aplikacje dla systemów Mac OS i Linux, a także kilka narzędzi do budowania aplikacji dla systemów Android i iOS.

Ma to jednak głębsze konsekwencje. Chcąc budować aplikacje dla systemów MacOS i Linux, trzeba korzystać z innego środowiska *Common Language Runtime* (*CLR*), które jest w stanie działać na tych platformach bez utraty wydajności. To właśnie jest rola nowej platformy .NET Core.

Do momentu napisania tej książki firma Microsoft opublikowała kilka ambitnych usprawnień ekosystemu .NET opartych głównie na dwóch odmianach .NET:

Pierwszym elementem jest dotychczas dostępna odmiana .NET (.NET Framework 4.6.x), a drugim jest nowa wersja mająca pozwalać na kompilacje działające nie tylko w systemach Windows, ale również Linux i Mac OS.

.NET Core jest ogólną nazwą dla nowej wersji CLR z otwartym kodem źródłowym udostępnionej w roku 2015 (zaktualizowanej niedawno do wersji 1.1), której zadaniem jest obsługa wielu elastycznych implementacji .NET. Ponadto zespół pracuje nad technologią zwaną *.NET Native*, która pozwala na kompilację do kodu naturalnego dla każdej platformy docelowej.

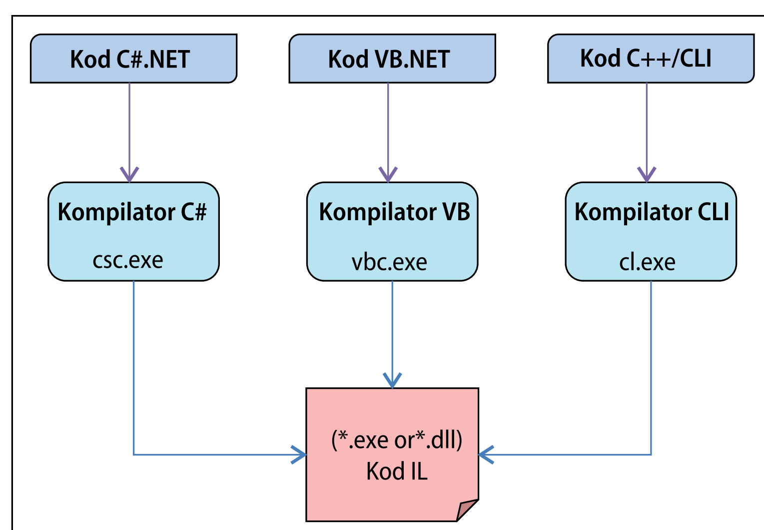
Pozostańmy jednak przy głównych pojęciach związanych z CLR z punktu widzenia niezależnego od wersji.

Cały projekt jest dostępny w serwisie GitHub pod adresem *https://github.com/dotnet/coreclr*.

Common Language Runtime

Aby poradzić sobie z niektórymi problemami związanymi z COM i wprowadzić nowe możliwości mające stanowić część nowej platformy, zespół pracowników firmy Microsoft zaczął rozwijać wcześniejsze pomysły (a także nazwy związane z nową platformą). Wkrótce więc platforma została przemianowana na *Component Object Runtime* (*COR*), a przy publikacji pierwszej wersji beta otrzymała nazwę Common Language Runtime, podkreślającą fakt, że nowa platforma nie jest związana z jednym językiem.

Istnieją faktycznie dziesiątki kompilatorów, z których można korzystać w .NET, a wszystkie one generują kod pośredni, który ostatecznie jest konwertowany na kod rodzimy podczas wykonywania programu, jak pokazano na poniższym rysunku:



Środowisko CLR, podobnie jak COM, skupia się na kontraktach pomiędzy składnikami, a te kontrakty opierają się na typach – tutaj jednak podobieństwa się kończą. W odróżnieniu od COM, środowisko CLR ustanawia dobrze zdefiniowaną formę określającą kontrakty, znaną zwykle jako metadane.

CLR zawiera też możliwość odczytywania metadanych bez żadnej wiedzy na temat wewnętrznego formatu plików. Co więcej, takie metadane można rozszerzać poprzez niestandardowe atrybuty, które same mają silnie określone typy. Inną interesującą informacją zawartą w metadanych jest informacja o wersji (nie powinno być żadnych zależności od rejestru) i zależnościach od innych składników.

Dla każdego składnika (zwanego podzespołem) obecność metadanych jest obowiązkowa, co oznacza, że nie jest możliwy dostęp do składnika bez odczytania jego metadanych. W początkowych wersjach implementacje zabezpieczeń były oparte głównie na pewnych elementach zawartych w metadanych. Takie metadane są dostępne dla każdego innego programu wewnątrz lub poza CLR poprzez proces zwany *refleksją*.

Inną ważną różnicą jest to, że kontrakty .NET opisują logiczną strukturę typów. Nie ma między innymi reprezentacji w pamięci, odczytywania sekwencji kolejności, wyrównywania, czy też konwencji dotyczących parametrów, jak to szczegółowo wyjaśnia Don Box w swojej wspaniałej książce Essential .NET (*http://www.amazon.com/Essential-NET-Volume-Language-Runtime/dp/0201734117*).

Common Intermediate Language

Te wcześniejsze konwencje i protokoły są rozpatrywane w CLR za pośrednictwem techniki zwanej wirtualizacją kontraktów. Zakłada to, że większość kodu (jeśli nie całość) napisanego dla CLR nie zawiera kodu maszynowego, ale język pośredni zwany *Common Intermediate Language* (*CIL*) lub po prostu *Intermediate Language* (*IL*).

CLR nigdy nie wykonuje instrukcji CIL bezpośrednio. Zamiast tego, kod CIL jest zawsze tłumaczony na rodzimy kod maszynowy przed jego wykonaniem przy pomocy techniki zwanej kompilacją *JIT* (*Just-In-Time*). Proces JIT zawsze dostosowuje wynikowy kod wykonywalny do komputera docelowego (niezależnie od programisty). Istnieje kilka trybów przeprowadzania procesu JIT i przyjrzymy się im dokładniej w dalszej części tego rozdziału.

CLR moglibyśmy więc nazwać platformą skupiającą się na typach. Dla CLR wszystko jest typem, obiektem lub wartością.

Zarządzane wykonywanie kodu

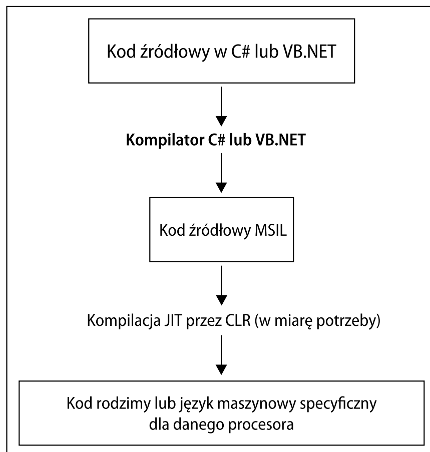
Innym istotnym czynnikiem, związanym z zachowaniem CLR, jest fakt, że programiści mogą zapomnieć o jawnym zarządzaniu pamięcią i ręcznym zarządzaniu wątkami (istotnym zwłaszcza w przypadku korzystania z języków takich jak C i C++) oraz powinni przyjąć nowy sposób wykonywania kodu proponowany przez CLR: zarządzane wykonywanie kodu.

W przypadku zarządzanego wykonywania kodu środowisko CLR ma pełną wiedzę na temat wszystkiego, co dzieje się w jego kontekście wykonywania. Obejmuje to każdą zmienną, metodę, typ, zdarzenie itd. Na wiele sposobów pobudza to produktywność i ułatwia debugowanie.

Dodatkowo CLR wspiera tworzenie kodu dla środowiska uruchomieniowego poprzez narzędzie zwane CodeDOM. Dzięki tej funkcji można generować kod w różnych językach i kompilować go (oraz wykonywać) bezpośrednio w pamięci.

Wszystko to prowadzi nas do kolejnych nasuwających się pytań: które języki są dostępne przy korzystaniu z tej infrastruktury, jakie są ich wspólne cechy, w jaki sposób kod wynikowy jest budowany i przygotowywany do wykonania, jakie są jednostki przechowywania informacji (jak wspominałem, zwane są one podzespołami)

i wreszcie, jak jest zorganizowana cała ta informacja w formie podzespołów oraz jaka jest ich struktura?



Składniki i języki

Każde środowisko wykonawcze posługuje się pojęciem składników oprogramowania. W przypadku CLR takie składniki muszą być napisane w języku zgodnym z CLI i odpowiednio skompilowane. Listę języków CLI można znaleźć w Wikipedii. Czym jest jednak język zgodny z CLI?

CLI oznacza *Common Language Infrastructure* (wspólna infrastruktura językowa) i jest to specyfikacja oprogramowania ustandaryzowana przez *ISO* i *ECMA*, opisująca kod wykonywalny i środowisko uruchomieniowe, które pozwalają na korzystanie z wielu wysokopoziomowych języków na różnych platformach komputerowych bez konieczności przepisywania kodu dla konkretnych rodzajów architektury. Platforma .NET Framework oraz darmowa i otwarta platforma Mono są implementacjami CLI.

Oto oficjalne serwisy dla wymienionych organizacji i technologii:

- *ISO*: *http://www.iso.org/iso/home.html*
- *ECMA*: *http://www.ecma-international.org/*
- *MONO*: *http://www.mono-project.com/*
- *Języki CLI*: *https://en.wikipedia.org/wiki/List_of_CLI_languages*

Najważniejszymi cechami CLI są (za Wikipedią):

- Po pierwsze, w celu zastąpienia COM, metadane są kluczowe i zapewniają informacje dotyczące architektury podzespołów takie jak spis tego, co można znaleźć w środku. Każdy program może odczytać te informacje, ponieważ nie zależą one od języka.

- Powinien być więc wspólny zestaw reguł dotyczących typów danych i operacji na nich. Jest nim wspólny system typów – *Common Type System* (*CTS*). Wszystkie języki stosujące się do CTS mogą działać na tym samym zbiorze zasad.
- Inny zestaw reguł służy minimalnej współpracy pomiędzy językami i powinien być wspólny dla wszystkich języków z tej grupy, aby biblioteka DLL stworzona w jednym języku, a następnie skompilowana, mogła być wykorzystana przez inną bibliotekę DLL skompilowaną w innym języku CTS.
- Wreszcie mamy system wirtualnego wykonywania kodu – Virtual Execution System, który jest odpowiedzialny za wykonywanie tej aplikacji i wiele innych zadań, takich jak zarządzanie pamięcią programu, organizowanie bloków wykonywania kodu itd.

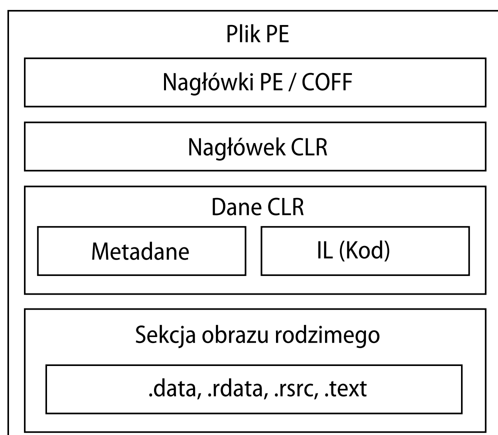
Mając to wszystko na uwadze, podczas korzystania z kompilatora .NET generujemy strumień bajtów zapisywany zwykle w pliku w lokalnym systemie plików lub na serwerze WWW.

Struktura pliku podzespołu

Pliki generowane w procesie kompilacji są zwane podzespołami, a każdy podzespół stosuje się do podstawowych zasad dla każdego innego pliku wykonywalnego w systemie Windows i dodaje kilka rozszerzeń oraz informacji koniecznych do wykonywania go w zarządzanym środowisku.

W skrócie podzespół jest po prostu zbiorem modułów zawierających kod IL i metadane, służącym jako podstawowa jednostka składnika oprogramowania w CLI. Bezpieczeństwo, wersjonowanie, ustalanie typów, procesy (domeny aplikacji) itd. działają na bazie pojedynczego podzespołu.

Oznacza to zmiany w strukturze plików wykonywalnych. Prowadzi to do nowej architektury plików przedstawionej na poniższym rysunku:



Plik PE jest zgodny z formatem Portable / Executable – formatem plikowym dla plików wykonywalnych, kodu obiektowego, bibliotek DLL, plików czcionek (FON) i innych używanych w 32-bitowych i 64-bitowych wersjach systemów operacyjnych Windows. Został wprowadzony po raz pierwszy przez Microsoft w systemie Windows NT 3.1, a wszystkie późniejsze wersje Windows również obsługują tę strukturę plików.

Dlatego właśnie w formacie tym znajdziemy nagłówek PE/COFF, który zawiera kompatybilne informacje wymagane przez system. Jednakże z punktu widzenia programisty .NET najważniejsze jest, że podzespół zawiera trzy główne obszary: nagłówek CLR, kod IL oraz sekcję z zasobami (na rysunku: *Sekcja obrazu rodzimego*).

Szczegółowy opis formatu PE jest dostępny pod adresem *http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx*.

Wykonywanie programu

Wśród bibliotek powiązanych z CLR znajdziemy kilka odpowiedzialnych za ładowanie podzespołów do pamięci oraz uruchamianie i inicjowanie kontekstu wykonywania. Są one ogólnie nazywane modułem ładującym CLR (CLR Loader). Razem z kilkoma innymi narzędziami zapewniają następujące funkcje:

- Automatyczne zarządzanie pamięcią
- Wykorzystanie odśmiecania pamięci
- Dostęp do metadanych w celu znalezienia informacji o typach
- Ładowanie modułów
- Analizowanie zarządzanych bibliotek i programów
- Solidny podsystem zarządzania wyjątkami pozwalający programom na zgłaszanie i reagowanie na błędy w ustrukturyzowany sposób.
- Współpraca ze starszym i rodzimym kodem
- Kompilacja JIT kodu zarządzanego do kodu rodzimego
- Złożona infrastruktura zabezpieczeń

Moduł ładujący wykorzystuje usługi systemu operacyjnego do ładowania, kompilowania i wykonywania podzespołu. Jak wspomnieliśmy wcześniej, CLR służy jako abstrakcja wykonywania kodu dla języków .NET. Aby to osiągnąć, wykorzystuje zestaw bibliotek DLL, które służą jako warstwa pośrednia pomiędzy systemem operacyjnym a aplikacją. Trzeba pamiętać, że samo środowisko CLR jest zbiorem bibliotek DLL, a te biblioteki DLL działają wspólnie, definiując wirtualne środowisko wykonawcze.

Najważniejszymi z tych bibliotek są:

- `mscoree.dll` (będąca w zasadzie fasadą dla faktycznych bibliotek DLL składających się na CLR)
- `clr.dll`
- `mscorsvr.dll` (dla wielu procesorów) lub `mscorwks.dll` (dla pojedynczego procesora)

W praktyce jedną z głównych ról biblioteki `mscoree.dll` jest wybranie odpowiedniej wersji (jednoprocesorowej lub wieloprocesorowej) w oparciu o wiele czynników, w tym właściwości sprzętu (ale nie tylko).

Biblioteka `clr.dll` jest faktycznym zarządcą, a pozostałe są narzędziami dodatkowymi wykorzystywanymi do różnych celów. Ta biblioteka jest jedyną z bibliotek CLR, która znajduje się w `$System.Root$`, co możemy odkryć poprzez proste wyszukiwanie:

```
C:\Windows>dir mscoree.dll /s
 Volume in drive C is Almacenamiento
 Volume Serial Number is B453-8D83

 Directory of C:\Windows\System32

10/30/2015  08:18 AM           396,288 mscoree.dll
               1 File(s)        396,288 bytes

 Directory of C:\Windows\SysWOW64

10/30/2015  08:18 AM           339,968 mscoree.dll
               1 File(s)        339,968 bytes
```

Mój system pokazuje dwie wersje (może być więcej) gotowe do uruchamiania programów skompilowanych dla wersji 32-bitowych i 64-bitowych. Pozostałe biblioteki DLL znajdują się w innym miejscu: zabezpieczonym zestawie katalogów zwanym globalnym magazynem podzespołów – *Global Assembly Cache* (*GAC*).

Najnowsze wydanie Windows 10 instaluje pliki dla wszystkich wersji magazynu GAC, odpowiadające wersjom 1.0, 1.1, 2.0, 3.0, 3.5 i 4.0, choć niektóre zawierają tylko minimum informacji, a pełne wersje można znaleźć jedynie dla .NET 2.0, .NET 3.5 (częściowo) i .NET 4.0.

W przypadku tych wersji, które nie są w pełni zainstalowane, możliwe jest doinstalowanie składników, jeśli byłyby wymagane przez jakieś starsze oprogramowanie. Wykonywanie programu .NET opiera się na wersji wskazanej w jego metadanych.

Można sprawdzić, które wersje .NET są zainstalowane w systemie, korzystając z narzędzia `CLRver.exe`, jak pokazano na kolejnym rysunku:

```
C:\Windows>clrver

Microsoft (R) .NET CLR Version Tool  Version 4.6.1055.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Versions installed on the machine:
v2.0.50727
v4.0.30319
```

Wewnętrznie przed wykonaniem programu ma miejsce kilka operacji. Gdy uruchamiamy program .NET, postępujemy tak samo, jakby był to po prostu standardowy plik wykonywalny systemu Windows.

Za kulisami system odczyta nagłówek, w którym zostanie poinstruowany o uruchomienie `mscore.dll`, co z kolei rozpocznie cały proces w środowisku zarządzanym. Pominiemy tutaj wszystkie zawiłości tego procesu, ponieważ wykracza to znacznie poza zakres tej książki.

Metadane

Wspomnieliśmy, że kluczowym aspektem nowego modelu programowania jest mocna zależność od metadanych. Co więcej, możliwość odczytu metadanych pozwala na techniki programowania, w których programy są generowane przez inne programy, a nie przez ludzi, w czym bierze udział CodeDOM.

Omówimy niektóre aspekty narzędzia CodeDOM i jego zastosowania, gdy będziemy zajmować się językiem, a także przyjrzymy się, jak środowisko programistyczne wykorzystuje tę funkcję za każdym razem, gdy tworzy kod źródłowy z szablonu.

Żeby pomóc środowisku CLR w znalezieniu różnych elementów podzespołu, każdy podzespół ma dokładnie jeden moduł, którego metadane zawierają manifest podzespołu: dodatkowy element metadanych CLR, który działa jako katalog plików uzupełniających, zawierających dodatkowe definicje typów i kod. Co więcej, CLR może bezpośrednio ładować moduły, które zawierają manifest podzespołu.

Jak więc wygląda manifest w rzeczywistym programie i jak możemy zbadać jego zawartość? Na szczęście mamy garść narzędzi .NET (technicznie rzecz biorąc nienależących do CLR, ale do ekosystemu platformy .NET), które pozwalają nam na łatwe zwizualizowanie tych informacji.

Wprowadzenie do metadanych przy pomocy podstawowego programu Hello World

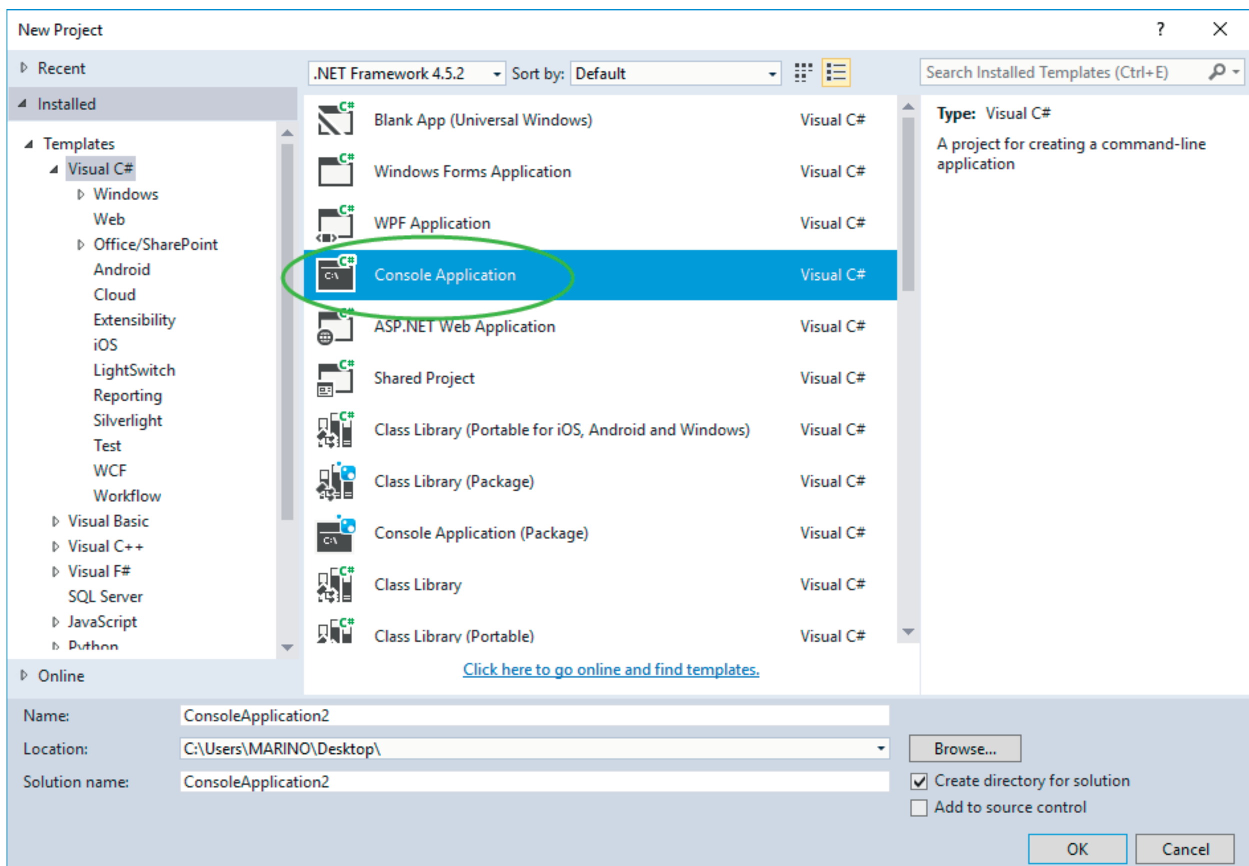
Zbudujmy typowy program „Hello World” i przeanalizujmy jego zawartość po skompilowaniu, abyśmy mogli zbadać, jak kod jest konwertowany na język pośredni *Intermediate Language* (*IL*) i gdzie są te metainformacje, o których mówimy.

W tej książce będę korzystać z Visual Studio 2015 Community Edition Update 1 (lub nowszej wersji, jeśli pojawi się aktualizacja) ze względów, które wyjaśnię później. Wersję tę można zainstalować za darmo; jest to w pełni funkcjonalna wersja z dużą liczbą typów projektów, narzędzi itd.

Visual Studio 2015 CE update 1 można znaleźć pod adresem *https://www.visualstudio.com/vs/community/*.

Jedynym wymaganiem jest darmowa rejestracja w celu uzyskania licencji programisty, wykorzystywanej przez Microsoft do celów statystycznych – to wszystko.

Po uruchomieniu Visual Studio, wybieramy w głównym menu opcje **New Project** i przechodzimy do szablonów **Visual C#**, gdzie środowisko programistyczne oferuje kilka typów projektów, a następnie wybieramy aplikację konsolową, jak pokazano na poniższym zrzucie ekranu:



Visual Studio utworzy podstawową strukturę kodu, składającą się z kilku odwołań do bibliotek (więcej na ten temat później) oraz blok przestrzeni nazw, zawierający

klasę `Program`. Wewnątrz tej klasy można znaleźć punkt wejścia do aplikacji podobny do tego, co można znaleźć w językach C++ lub Java.

Aby wypisać coś na konsoli, skorzystamy z dwóch metod statycznych klasy `Console`: metody `WriteLine`, która wypisuje łańcuch tekstu dodając na końcu przejście do nowego wiersza oraz metody `ReadLine`, która zatrzymuje program w oczekiwaniu na wpisanie jakichś znaków przez użytkownika i naciśnięcie klawisza Enter.

Po usunięciu odwołań, z których nie będziemy korzystać i wprowadzeniu kilku instrukcji wspomnianych powyżej, kod będzie wyglądać następująco:

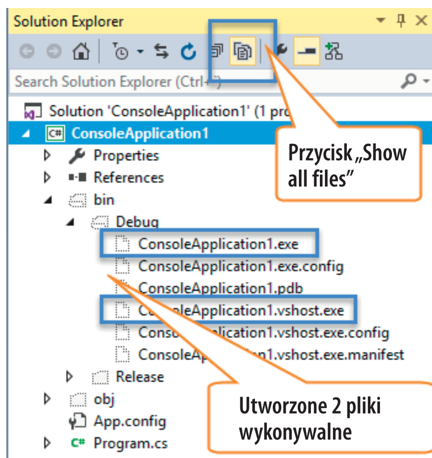
```
using System;
namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Hello! I'm executing in the CLR context.");
      Console.ReadLine();
    }
  }
}
```

Aby przetestować ten program, musimy nacisnąć *F5* lub przycisk **Start**, a zobaczymy odpowiedni tekst w oknie konsoli (nic szczególnego, więc nie dołączam zrzutu ekranu).

Podczas edytowania kodu można zauważyć kilka przydatnych cech edytora zintegrowanego środowiska programistycznego: kolorowanie kodu (rozróżniające różne elementy: klasy, metody, argumenty, literały itd.); technologię *IntelliSense*, która podpowiada pasujące do kontekstu elementy podczas pisania kodu, podpowiedzi wyświetlające typ zwracany przez metody, wartości stałych i liczbę odwołań w innych częściach kodu do każdego składnika programu.

Środowisko programistyczne ma też setki innych przydatnych funkcji, ale zaczniemy je testować od następnego rozdziału, gdy zajmiemy się dokładniej aspektami języka C#.

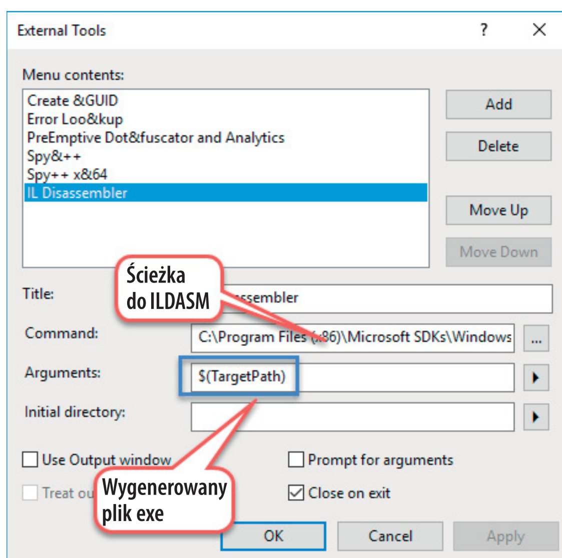
Jeśli chodzi o ten niewielki program, to możemy sprawdzić, co zostało wytworzone w folderze `Bin/Debug` naszego projektu (aby zobaczyć te pliki, trzeba pamiętać o włączeniu przycisku **Show all files** w nagłówku panelu Solution Explorer).



Jak widać, wygenerowane zostały dwa pliki wykonywalne. Pierwszy jest samodzielnym plikiem wykonywalnym, który można uruchomić bezpośrednio z tego foldera. Drugi (z przedrostkiem `.vshost` przed rozszerzeniem) jest wykorzystywany przez Visual Studio w czasie debugowania i zawiera dodatkowe informacje wymagane przez środowisko programistyczne. Oba działają tak samo i wypisują ten sam tekst.

Gdy już mamy plik wykonywalny, czas podłączyć do Visual Studio narzędzie .NET, które pozwoli nam zobaczyć metadane, o których mówiliśmy.

W tym celu przechodzimy do opcji **Tools** | **External Tools** w menu głównym, a zobaczymy konfiguracyjne okno dialogowe przedstawiające kilka już podłączonych narzędzi zewnętrznych; naciskamy przycisk **New** i zmieniamy tytuł na `IL Disassembler`, jak pokazano na poniższym zrzucie ekranu:

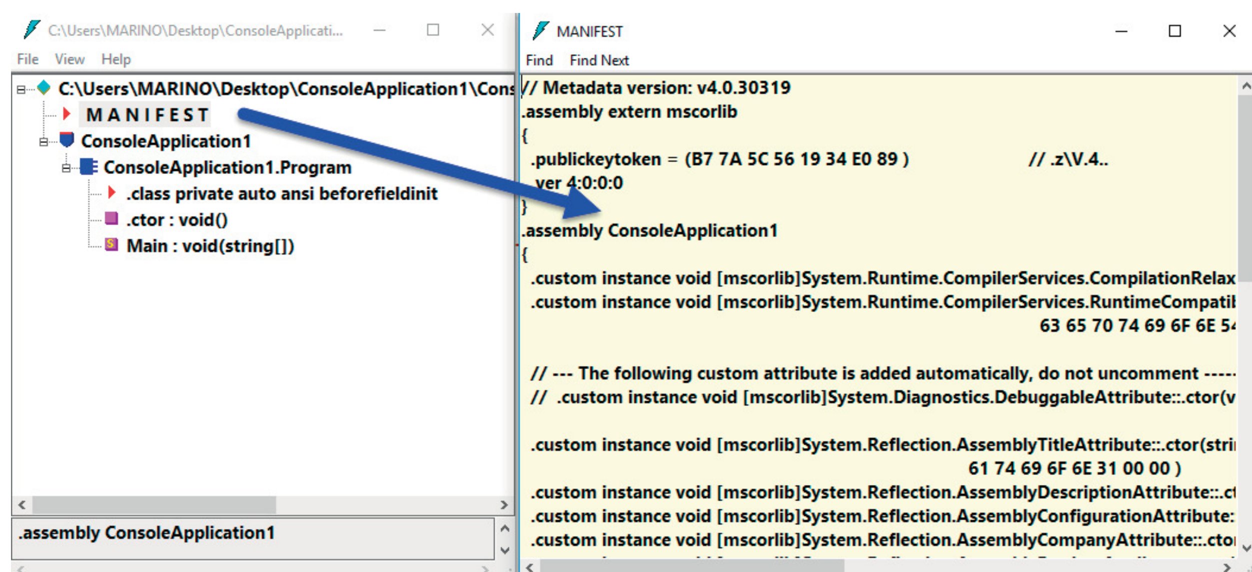


Następnie musimy skonfigurować argumenty, które zamierzamy przekazywać do tego nowego narzędzia: nazwę narzędzia i wymagane parametry. Istnieje kilka wersji tego narzędzia. Będzie to zależeć od konkretnego komputera.

Dla naszych celów wystarczy podać następujące informacje:

- Folder główny narzędzia (o nawie `ILDASM.exe`, na moim komputerze znajduje się ono w folderze `C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools`)
- Ścieżkę do wygenerowanego pliku wykonywalnego – w tym przypadku wykorzystuję predefiniowane makro `$targetpath`.

Zakładając, że nasz program jest już skompilowany, możemy wrócić do menu **Tools** i znaleźć nową opcję `IL Disassembler`. Po jej uruchomieniu pojawi się okno przedstawiające kod IL naszego programu oraz odwołanie o nazwie `Manifest` (pokazujące metadane), możemy też podwójnie kliknąć, aby wyświetlić osobne okno z tą informacją, jak pokazano na poniższym zrzucie ekranu:



Warto zwrócić uwagę, że zmieniłem rozmiar czcionki w ILDASM, aby była lepiej widoczna.

Informacja zawarta w manifeście pochodzi z dwóch źródeł: samego środowiska programistycznego skonfigurowanego na przygotowanie podzespołu do wykonania oraz z konfigurowalnych informacji, które możemy osadzać w manifeście pliku wykonywalnego (takich jak opisy, tytuł podzespołu, informacje o firmie, znakach towarowych, języku, itp.) W następnym rozdziale zbadamy, jak można skonfigurować te informacje.

W ten sam sposób możemy przeanalizować zawartość każdego węzła pokazanego w głównym oknie ILDASM. Na przykład, jeśli chcemy zobaczyć kod IL związany z funkcją punktu wejścia `Main`, narzędzie to pokaże nam osobne okno, w którym

możemy dokładnie przejrzeć kod IL (można zwrócić uwagę na obecność tekstu `cil managed` obok deklaracji funkcji Main):

```
ConsoleApplication1.Program::Main : void(string[])
Find   Find Next
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       19 (0x13)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr      "Hello! I'executing in the CLR context."
  IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000b:  nop
  IL_000c:  call       string [mscorlib]System.Console::ReadLine()
  IL_0011:  pop
  IL_0012:  ret
} // end of method Program::Main
```

Ten kod IL zostanie przekonwertowany na kod maszynowy

Jak zaznaczyłem na tym zrzucie ekranu, wpisy z przedrostkiem `IL_` będą przekonwertowane na kod maszynowy podczas wykonywania programu. Warto zwrócić uwagę na podobieństwo tych instrukcji do asemblera.

Trzeba też pamiętać, że nie zmieniło się to od pierwszej wersji .NET: główne pojęcia i procedury generowania CIL i kodu maszynowego są zasadniczo takie same jak dawniej.

PreJIT, JIT, EconoJIT i RyuJIT

Wspominałem już, że proces konwertowania tego kodu IL do kodu maszynowego jest obsługiwany przez inny element platformy .NET zwany kompilatorem *JIT (Just-In-Time)*. Od samego początku platformy .NET proces ten może być wykonywany na co najmniej trzy różne sposoby, dlatego mamy trzy nazwy kompilatorów z przyrostkiem JIT.

Upraszczając szczegóły tych procesów, możemy stwierdzić, że domyślną metodą kompilacji (i preferowaną w ogólnych sytuacjach) jest zwykła kompilacja JIT (nazwijmy ją trybem Normal JIT):

- W trybie Normal JIT kod jest kompilowany na żądanie i jest zachowywany w pamięci podręcznej na przyszły użytek. W ten sposób w miarę dalszego działania aplikacji każdy kod, którego wykonanie jest wymagane w późniejszym czasie, a który jest już skompilowany, jest po prostu pobierany z obszaru pamięci podręcznej. Proces ten jest wysoce zoptymalizowany, a spadek wydajności jest pomijalny.
- W trybie PreJIT .NET funkcjonuje w inny sposób. Do skorzystania z PreJIT będziemy potrzebować narzędzia o nazwie `ngen.exe` (co jest skrótem od „native generation”) do wytworzenia kodu maszynowego przed pierwszym wykonaniem

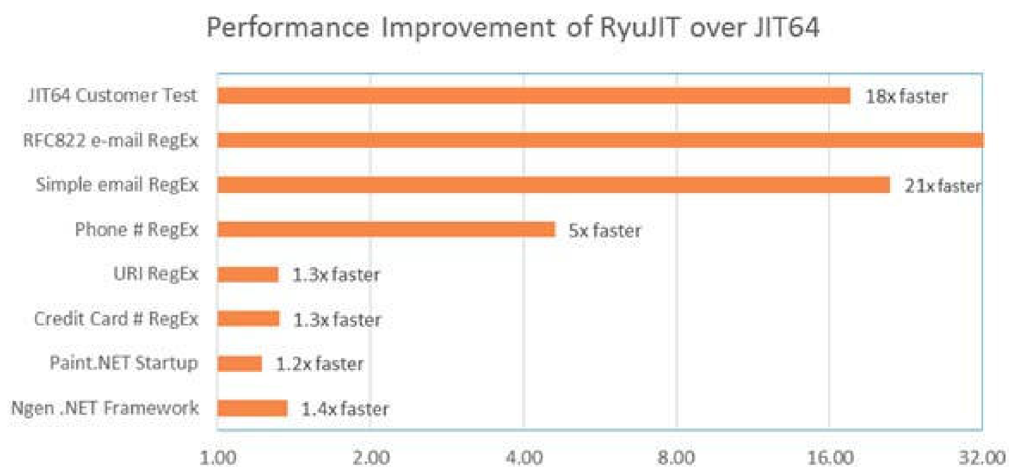
programu. Kod ten jest następnie konwertowany i plik `.exe` jest nadpisywany przez kod maszynowy, co daje pewne optymalizacje, zwłaszcza w trakcie rozruchu aplikacji.

- Jeśli chodzi o tryb EconoJIT, jest on używany głównie w przypadku aplikacji wdrażanych na urządzeniach z niewielką ilością pamięci, takich jak urządzenia mobilne, i jest zbliżony do trybu Normal JIT z tą różnicą, że skompilowany kod nie jest przechowywany w pamięci podręcznej.

W roku 2015 firma Microsoft dalej pracowała nad specjalnym projektem o nazwie Roslyn, który między innymi obejmuje zestaw narzędzi zapewniających dodatkowe funkcje związane z procesem zarządzania kodem, kompilacją i wdrażaniem. W powiązaniu z tym projektem (który zostanie omówiony dogłębnie w rozdziale 4, *Porównanie różnych typów podejścia do programowania*) pojawił się inny wariant JIT o nazwie RyuJIT, który od początku został udostępniony jako projekt z otwartym kodem źródłowym i jest teraz dołączany domyślnie do najnowszej wersji Visual Studio (V. Studio 2015 Update 1).

Pozwolę sobie zacytować, co zespół .NET mówi o swoim nowym kompilatorze:

RyuJIT jest nowym kompilatorem x64 następnej generacji, dwukrotnie szybszym od poprzedniego, co oznacza, że aplikacje kompilowane przez RyuJIT uruchamiają się o 30% szybciej (czas potrzebny na kompilację JIT to tylko jeden ze składników czasu uruchamiania się aplikacji). Co więcej, nowy kompilator JIT nadal tworzy świetny kod, który działa wydajnie w procesach serwerowych.



Powyższy wykres porównuje współczynnik czasu kompilacji („przepustowość”) dla JIT64 i RyuJIT na różnych próbkach kodu. Każdy wiersz pokazuje, ile razy szybszy jest RyuJIT od JIT64, więc wyższe liczby są lepsze.