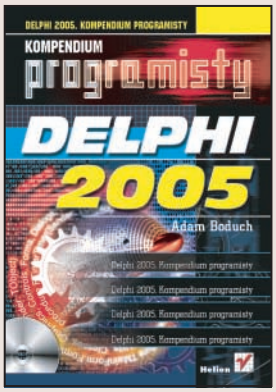
Delphi 2005. Kompendium programisty

Autor: Adam Boduch
ISBN: 83-7361-780-9
Format: B5, stron: 1048



Nowa wersja środowiska programistycznego Delphi to kolejne nowości – pełna integracja zarówno z platformą .NET, jak i Win32, nowe technologie tworzenia aplikacji, nowe komponenty i możliwość stosowania nie tylko języka Delphi, ale również C#. Za pomocą najnowszej wersji Delphi można tworzyć nie tylko „standardowe" aplikacje, ale również wykorzystywać możliwości oferowane przez .NET. Tworzenie aplikacji klient-serwer, usług sieciowych, intranetów i dynamicznych witryn WWW w oparciu o tę platformę stało się szybsze i łatwiejsze. Większe możliwości pociągnęły za sobą również sporo zmian w samym środowisku i sposobie korzystania z niego.

Książka „Delphi 2005. Kompendium programisty" przedstawia najnowszą wersję tego popularnego narzędzia. Jej pierwsza część opisuje zagadnienia podstawowe – interfejs użytkownika i zasady programowania, najważniejsze elementy platformy .NET oraz podstawy wizualnego tworzenia aplikacji. Kolejne części zawierają bardziej zaawansowane tematy – wykrywanie błędów w programach, tworzenie aplikacji dla platformy .NET, komunikacja z bazami danych oraz programowanie sieciowe. Czytając książkę, dowiesz się, jak tworzyć aplikacje i usługi sieciowe, jak korzystać z technologii XML i IntraWeb oraz jak tworzyć wydajne aplikacje klient-serwer.

- Interfejs użytkownika Delphi 2005
- Język programowania Delphi
- Podstawowe elementy .NET
- Wizualne tworzenie aplikacji i stosowanie komponentów
- Środowisko .NET Framework
- Zasady programowania obiektowego
- Wykrywanie błędów w aplikacjach
- Aplikacje bazodanowe
- Technologia ADO.NET i BDE.NET
- Wykorzystanie języka XML
- Tworzenie dynamicznych witryn WWW w języku ASP.NET
- Usługi sieciowe
- Podstawy języka C#

Jeśli planujesz wykorzystanie Delphi 2005 do budowania aplikacji dla platformy .NET, ta książka będzie dla Ciebie niezastąpionym źródłem informacji

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

5.

Przegląd .NET Framework

W rozdziale 2. omówiłem podstawowe aspekty platformy .NET. Czytelnik powinien już znać podstawowe pojęcia związane z tą technologią, a także umieć pisać proste aplikacje w Delphi. Jak dotąd jednak przykłady prezentowane przeze mnie w rozdziale 3. były oparte na programowaniu dla Win32.

W tym rozdziale zajmiemy się tylko i wyłącznie programowaniem dla platformy .NET oraz dokładniej omówimy technologie związane z tą platformą.

Środowisko .NET Framework obejmuje swym zakresem wszystkie warstwy tworzenia oprogramowania: od systemu operacyjnego po bibliotekę klas (jak np. Windows Forms).

W tym rozdziale:

- przedstawię szczegóły działania środowiska CLR,
- omówię nowe pojęcia: kod zarządzany i niezarządzany,
- opiszę, czym jest CLS i CTS.

Środowisko CLR

Wspólne środowisko uruchomieniowe (CLR) stanowi podstawowy element platformy .NET Framework. W momencie uruchomienia aplikacji .NET, CLR odpowiada za jej sprawne wykonanie (załadowanie do pamięci), przydział pamięci, obsługę błędów itp. W standardowym modelu programowania — Win32, za tego typu czynności odpowiadał zwyczajnie system operacyjny (Windows). Po zainstalowaniu na komputerze środowiska .NET Framework, odpowiednie biblioteki systemu pozwalają na rozpoznanie, czy dany program jest aplikacją Win32, czy też

.NET. Jeżeli jest to aplikacja .NET, uruchomione zostaje środowisko CLR, pod kontrolą którego działa program. To, co dzieje się „w tle" nas nie interesuje, nie trzeba się tym przejmować.

W jaki jednak sposób system operacyjny rozpoznaje, która aplikacja jest aplikacją .NET? Dzieje się tak dlatego, że aplikacja wykonywalna .NET (plik *.exe*) jest inaczej zbudowana niż standardowe programy Win32. Tę kwestię postaram się wyjaśnić w paru kolejnych podrozdziałach.

Kod pośredni IL

Kompilatory działające pod kontrolą systemu Windows kompilują kody źródłowe do postaci 32-bitowego kodu maszynowego. W efekcie otrzymujemy aplikacje *.exe* czy biblioteki *.dll*. Taki sposób uniemożliwia przeniesienie aplikacji na urządzenia czy systemy, które działają pod kontrolą innych procesorów. Może także stwarzać problemy z innymi wersjami systemu operacyjnego (w tym wypadku Windows). Wszystko dlatego, że aplikacje wykonywalne komunikują się z API, które może różnić się w poszczególnych wersjach systemu.

Rozwiązaniem tego problemu jest kompilacja programu do kodu pośredniego, nazywanego *Common Language Infrastructure* (z ang. *architektura wspólnego języka*, CLI).

Wskazówka

CLI na platformie .NET jest często nazywany MSIL (*Microsoft Intermediate Language*) lub po prostu — IL.

Kod pośredni IL przypomina kod języka Asembler:

```
.method family hidebysig virtual instance void
        Dispose(bool Disposing) cil managed
{
  // Code size       30 (0x1e)
  .maxstack  2
  IL_0000:  ldarg.1
  IL_0001:  brfalse.s  IL_0016
  IL_0003:  ldarg.0
  IL_0004:  ldfld      class [System]System.ComponentModel.Container
                       WinForm.TWinForm1::Components
  IL_0009:  brfalse.s  IL_0016
  IL_000b:  ldarg.0
  IL_000c:  ldfld      class [System]System.ComponentModel.Container
                       WinForm.TWinForm1::Components
  IL_0011:  callvirt   instance void [System]System.ComponentModel.Container::Dispose()
  IL_0016:  ldarg.0
  IL_0017:  ldarg.1
  IL_0018:  call       instance void
                       [System.Windows.Forms]System.Windows.Forms.Form::Dispose(bool)
  IL_001d:  ret
} // end of method TWinForm1::Dispose
```

W takiej postaci kod IL jest kompilowany do kodu maszynowego, który może już zostać uruchomiony. Tak właśnie działa język Java, z którego pomysł zaczerpnął Microsoft projektując platformę .NET.

Wszystko to jest możliwe dzięki tzw. maszynom wirtualnym, czyli aplikacjom przystosowanym do konkretnej wersji systemu/procesora. Środowisko CLR odpowiada za kompilację kodu pośredniego na maszynowy, w trakcie uruchamiania aplikacji. Dzieje się to jednak na tyle szybko, że dla użytkownika jest to niezauważalne.

Wskazówka

Możliwe jest również jednorazowe skompilowanie danego programu od razu na kod maszynowy, dzięki temu przy każdym uruchamianiem programu są oszczędzane zasoby systemowe, potrzebne do uruchomienia kompilatora JIT (ang. *Just-In-Time*).

Kod zarządzany i niezarządzany

Platforma .NET definiuje dwa nowe pojęcia: kod zarządzany (*managed code*) oraz niezarządzany (ang. *unmanaged code*), które są istotne z punktu widzenia CLR. Kod niezarządzany jest zwykłym kodem, wykonywanym poza środowiskiem .NET, zatem określenie to oznacza stare aplikacje kompilowane dla środowiska Win32. Natomiast — jak nietrudno się domyśleć — kod zarządzany jest wykonywany pod kontrolą CLR.

Zmiany w Delphi 2005 oraz w .NET w porównaniu z Win32 są na tyle duże, że problemem staje się współdziałanie obu rodzajów aplikacji (aplikacji .NET oraz Win32), jak również korzystanie z zasobów starszych aplikacji Win32 — np. bibliotek DLL. Dlatego też w .NET w tym celu wykorzystuje się mechanizm zwany *marshalingiem* [1], który jest związany z określeniem sposobu, w jaki dane mają być przekazywane z kodu niezarządzanego do zarządzanego. Tym jednak nie należy się teraz przejmować — podejmę ten temat w rozdziale 11.

Moduł zarządzany

Każdy moduł zarządzany jest przenośnym plikiem systemu Windows. Moduły zarządzane są generowane przez kompilatory zgodne z platformą .NET, a w ich skład wchodzą następujące elementy:

- ✧ Nagłówek PE (ang. *PE Header*) — standardowy nagłówek pliku wykonywalnego systemu Windows,
- ✧ Nagłówek CLR (ang. *CLR Header*) — dodatkowe informacje, charakterystyczne dla danego środowiska CLR,

[1] Niestety nie znalazłem dobrego, polskiego odpowiednika tego słowa, które w pełni oddawałoby istotę rzeczy.

- Metadane (ang. *metadata*) — informacje o typach modułów używanych w programie oraz ich wzajemnych powiązaniach,
- Kod zarządzany (ang. *managed code*) — wspólny kod generowany przez kompilatory .NET (kod IL).

Wskazówka

Informacje o metadanych znajdują się w 8. rozdziale tej książki.

Podzespoły

To jest bardzo ważne pojęcie, którym nieraz będę się posługiwał w tej książce. W najprostszym ujęciu podzespołem (ang. *assembly*) nazywamy każdą aplikację działającą pod kontrolą .NET.

Każdy moduł zarządzany wymaga podzespołu. Jeden podzespół może zawierać jeden lub więcej modułów zarządzanych. Podzespół może zawierać również inne pliki, takie jak dokumenty, grafikę itp.

Na tym etapie zagadnienie to może wydać się Czytelnikowi niezwykle skomplikowane. Podzespoły mogą zawierać jeden lub więcej modułów zarządzanych, w których z kolei znajdują się kod zarządzany oraz metadane.

Działanie CLR

Wspomniałem wcześniej, że CLR zajmuje się uruchamianiem aplikacji napisanej w .NET oraz ogólnie — zarządzaniem procesem jej działania. Prześledźmy proces uruchamiania aplikacji w .NET. Oto kilka etapów, które przechodzi aplikacja od momentu, gdy użytkownik zarządzi jej uruchomienie:

- ładowanie klasy (ang. *Class Loader*),
- weryfikacja,
- kompilacja.

Class Loader

Dotychczas jedynym formatem, rozpoznawanym przez systemy Windows jako aplikacja wykonywalna, był stary format PE. Stary format PE zawierał skompilowany kod maszynowy aplikacji. Instalując bibliotekę .NET w systemie, dodawane jest uaktualnienie mówiące o nowym formacie PE, dzięki czemu system jest w stanie rozpoznać również nowy format plików wykonywalnych.

Skoro teraz aplikacja .NET jest rozpoznawana przez system, ten w momencie jej uruchamiania oddaje sterowanie do CLR. CLR odczytuje zawartość pliku oraz listę klas używanych w programie.

Lista klas jest odczytywana z przeróżnych miejsc — szczególnie jest to manifest oraz metadane, a także plik *.config*, który może być dołączany do programu. Po odczytaniu klas następuje obliczenie ilości pamięci potrzebnej, aby załadować klasy. Następnie klasy są ładowane do pamięci.

Weryfikacja

Po załadowaniu klas do pamięci zawartość programu, czyli metadane oraz kod IL, zostają poddane weryfikacji. Jest to ważny etap, gdyż w przypadku niepowodzenia kod IL nie zostanie przekazany kompilatorowi JIT.

Kompilator JIT

Kompilator JIT odgrywa znaczącą rolę w procesie uruchamiania aplikacji. Po weryfikacji kodu jest on przekazywany do kompilatora, który kompiluje go do kodu maszynowego, a następnie gotowy program zostaje załadowany do pamięci. Znajduje się on w tej pamięci do czasu zakończenia działania aplikacji.

System CTS

Wiadomo już, czym są typy języka programowania. Wspólny system typów (*Common Type System*) jest bogatym zbiorem typów opracowanych przez firmę Microsoft na potrzeby .NET. W rzeczywistości, podczas programowania w Delphi z użyciem VCL.NET lub *WinForms* i zastosowaniem typu `Integer`, korzystamy z typu `System.Int32`, który należy do specyfikacji CTS.

Otwórzmy teraz okno repozytorium i wybierzmy kategorię *Delphi for .NET Projects*, a następnie *Console Application*. Można zadeklarować dwie zmienne i spróbować skompilować program:

```
var
  I1 : Integer;
  I2 : System.Int32;
```

Kompilacja odbędzie się bezproblemowo. Typy `Integer` oraz `System.Int32` są sobie równoważne. Typy takie jak `Integer` czy `String` są zachowane ze względu na kompatybilność projektów ze starszymi wersjami Delphi. Programista nie musi uczyć się nazw nowych typów — wykorzystuje nazwy dobrze mu znane i kompatybilne z .NET. Typy języka Delphi oraz ich odpowiedniki w CTS znajdują się w tabeli 5.1.

System CTS jest składnikiem CLR, odpowiada za weryfikowanie i zarządzanie tymi typami. Wszystkie typy, również te przedstawione w tabeli 5.1 wywodzą się z głównej klasy — `System.Object`.

Tabela 5.1. *Typy CTS*

Typ Delphi	Typ .NET Framework
String	System.String
Boolean	System.Boolean
Char	System.Char
Double	System.Double
Integer	System.Int32
Smallint	System.Int16
Word	System.UInt16
Int64	System.Int64
Byte	System.Byte

Specyfikacja CLS

Do tej pory możliwości w komunikowaniu się pomiędzy aplikacjami były nieco ograniczone. Kod aplikacji w środowisku Win32 może być dzielony pomiędzy biblioteki DLL. Po umieszczeniu funkcji w bibliotece DLL, istnieje możliwość ich eksportu, co z kolei pozwala na ich wykorzystanie przez aplikacje EXE.

W środowisku .NET aplikacje mają pełną zdolność do komunikowania się. Jeden podzespół może wykorzystywać funkcje drugiego i na odwrót. Wszystko to dzięki kompilacji do kodu pośredniego IL, do którego są kompilowane wszystkie aplikacje, bez względu na to, czy są pisane w C#, czy w Delphi. Tak więc, jeżeli napisano program do szyfrowania danych, można (jeżeli tylko programista tego zechce) udostępnić jego funkcjonalność innym podzespołom. Praktyczne przykłady tego zagadnienia zaprezentuję w rozdziale 8.

Nie tylko język pośredni ma tu znaczenie, ale również specyfikacja CLS (ang. *Common Language Specification*, czyli *wspólna specyfikacja języków*). Jest to zestaw reguł określających nazewnictwo oraz inne kluczowe elementy języka programowania. Jeśli projektanci języka programowania, który docelowo ma działać dla .NET, chcą, aby był on kompatybilny z CLS oraz miał zdolność do komunikowania się z pozostałymi podzespołami, muszą dostosować swój produkt do określonych wymagań.

Wskazówka

Na stronach firmy Microsoft można także znaleźć dodatkowe informacje na temat specyfikacji CLS:

http://msdn.microsoft.com/net/ecma/.

Biblioteka klas

Czytelnik wciąż spotyka się ze słowami: *klasa*, *obiekt*, których zresztą często używam w tym rozdziale. Wspominałem wcześniej, że na platformie Win32 aplikacje korzystały z WinAPI, czyli z zestawu funkcji pomocnych przy programowaniu. Wkrótce po tym pojawiły się takie biblioteki jak VCL, które jeszcze bardziej ułatwiały programowanie (VCL korzysta z funkcji WinAPI).

Biblioteka klas .NET Framework (*.NET Framework Class Library*) stanowi zestaw setek klas, bibliotek, interfejsów, typów, które mają zastąpić WinAPI. W założeniu FCL ma połączyć funkcjonalność WinAPI oraz dodatkowych bibliotek, takich jak VCL czy MFC (ang. *Microsoft Fundation Classes*).

Dla przykładu — w rozdziale 3. korzystaliśmy z funkcji `Writeln`, `Readln`, które są funkcjami Delphi umożliwiającymi operacje na konsoli. Odpowiednikiem tych funkcji na platformie .NET jest klasa `Console` oraz metody `WriteLine` oraz `ReadLine`. Oto prosty przykład programu, który pobiera informacje o imieniu użytkownika. Program korzysta z funkcji konsolowych biblioteki FCL:

```
program Project3;

{$APPTYPE CONSOLE}

var
  S : System.String;

begin
  Console.WriteLine('Cześć! Podaj swoje imię!');
  S := Console.ReadLine;

  Console.WriteLine('Cześć ' + S + '! Ja mam na imię Adam!');
  Console.ReadLine;
end.
```