helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Visual Basic 2005. Wprowadzenie do programowania w .NET

Autor: Matthew MacDonald
Tłumaczenie: Adam Majczak, Tomasz Walczak
ISBN: 978-83-246-0696-2
Tytuł oryginału: The Book of Visual Basic 2005: NET Insight for Classic VB Developers
Format: B5, stron: 592

Wszechstronny przewodnik po świecie programowania w .NET dla programistów języka Visual Basic

- Jakie nowe funkcje oferuje Visual Basic 2005?
- Jak wykorzystać najnowsze właściwości języka Visual Basic 2005 do przyspieszenia i ułatwienia programowania?
- Jak zwiększyć produktywność, wykorzystując możliwości platformy .NET?

Wciąż zastanawiasz się nad przejściem z Visual Basic 6 na wersję pracującą w środowisku .NET? Najwyższa pora! Visual Basic 2005 to język, na który wielu programistów czekało od lat. Jest w pełni obiektowy, ma usprawnioną obsługę błędów, nowy model obsługi zdarzeń oraz udostępnia wiele innych funkcji, które znacznie zwiększają produktywność. Integracja z .NET pozwala korzystać w języku Visual Basic z wszystkich możliwości tej platformy, pracować we wspólnym środowisku uruchomieniowym (CLR) i używać rozbudowanej biblioteki klas .NET.

„Visual Basic 2005. Wprowadzenie do programowania w .NET" to wszechstronny przewodnik po świecie programowania w najnowszej wersji języka Visual Basic. Czytając tę książkę, dowiesz się, jakie zmiany zostały wprowadzone w wersjach języka Visual Basic zgodnych z platformą .NET. Nauczysz się wykorzystywać je do przyspieszenia i ułatwienia programowania. Poznasz udogodnienia dostępne w środowisku Visual Studio, techniki wygodnej obsługi baz danych przy użyciu ADO.NET, udoskonalone narzędzia diagnostyczne i wiele innych funkcji, które ułatwią Ci tworzenie programów wysokiej jakości.

- Możliwości platformy .NET
- Praca w Visual Studio
- Programowanie obiektowe w Visual Basic 2005
- Tworzenie interfejsów użytkownika przy użyciu formularzy Windows
- Podzespoły i komponenty
- Diagnozowanie programów w Visual Studio
- Praca z bazami danych przy użyciu ADO.NET
- Programowanie wielowątkowe
- Tworzenie aplikacji i usług sieciowych
- Instalowanie programów napisanych w Visual Basic 2005

Poznaj Visual Basic 2005 i dołącz do społeczności programistów .NET

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

WPROWADZENIE **17**

1
REWOLUCJA .NET **23**

Krótka historia języka Visual Basic 23
Wejście w .NET 24
Ograniczenia „klasycznego” Visual Basic 25
Dziwaczna mieszanina Visual Basic 25
Odrębne języki 25
Bóle głowy przy rozwoju projektu 25
Piekło DLL 26
Niepełne wsparcie dla programowania obiektowego 26
Wizja .NET 26
Składniki .NET 27
Common Language Runtime (CLR) 27
Klasy .NET 28
Mówienie w tym samym języku 29
Dogłębna integracja języków 30
Przygotowana infrastruktura 30
Usługi sieciowe i internet kolejnej generacji 31
Otwarte standardy: XML, SOAP, WSDL oraz pozostałe litery alfabetu 31
Metadane. Czy to koniec piekła DLL? 32
Czy VB 2005 to jeszcze wciąż VB? 33
Udoskonalenia, bez których nie da się żyć 33
A teraz zmiany, które mogą frustrować 34
Ciemne strony .NET 34
A co z COM? 35
I co dalej? 35

2
ŚRODOWISKO PROGRAMOWANIA

ŚRODOWISKO PROGRAMOWANIA 37
Nowości w .NET 38
Rozpoczynanie pracy w IDE 39
Strona początkowa 40
Zmiana zachowania przy uruchamianiu 41
Tworzenie projektu 42
Dokumenty z zakładkami 44
Dokowane i zgrupowane okna 45
Zwiedzanie Visual Studio 45
Wyszukiwarka rozwiązań Solution Explorer 46
Przybornik: Toolbox 47
Okno właściwości: Properties 49
Wyświetlanie kodu 50
Dzielenie okien 52
Lista zadań Task List 53
Strzępki kodu: Code Snippets 55
Wstawianie strzępków 55
Zarządzanie strzępkami 56
Makra 57
Makro IDE 58
Tymczasowa makroinstrukcja — Temporary Macro 59
Makra z inteligencją 59
Makroinstrukcje a zdarzenia 60
Najprostszy możliwy program w środowisku .NET 62
Pliki tworzące projekt aplikacji MyFirstConsoleApplication 63
Foldery projektu aplikacji MyFirstConsoleApplication 64
Własności projektu — Project Properties 65
I co dalej? 67

3
PODSTAWY VB 2005

PODSTAWY VB 2005 69
Nowości w .NET 70
Wprowadzenie do biblioteki klas 72
Namespace 72
Assembly — skompilowany kod namespace 75
Typy danych 75
Stosowanie biblioteki klas 77
Dodanie referencji do pliku assembly 78
Jak importować namespace? 80
Wykorzystanie przestrzeni namespace z biblioteki klas 82
Szczególny obiekt — My 85
Pliki kodu 88
Bloki klas i modułów 89
Bloki namespace 90
Dodanie plików kodu 91

Typy danych 91
Typy danych w namespace System 92
Wspólna deklaracja wielu zmiennych 92
Inicjator VB — Initializer 93
Typy danych jako obiekty 93
Typ String 94
Bardziej efektywne obiekty klasy String 97
Czas i data 98
Macierze 98
Macierze i interfejs IEnumerable 99
Wbudowane cechy macierzy 100
Macierze jako typy referencyjne 101
Zmiany w działaniu operatorów 102
Skrócony zapis operatorów przypisania 102
Konwersja typu zmiennej 103
Math, czyli nieco matematyki 104
Random numbers — liczby pseudolosowe 104
Pewne nowe reguły dostępności — Scope 104
Logika na skróty 105
Szybkie pomijanie iteracji w pętli 107
Rozszerzone procedury 107
Wywołanie metody 108
Poprzez wartość lub poprzez referencję: ByVal, ByRef 108
Słowo kluczowe Return 109
Opcjonalne parametry 110
Wartości domyślne 111
Przeciążenie metod 111
Delegaty 113
I co dalej? 116

4
FORMULARZE WINDOWS 117
Nowości w .NET 118
Rozpoczęcie pracy 119
Zasobnik komponentów: Component Tray 120
Zindywidualizowane kreatory: Custom Designers 120
Blokowanie obiektów sterujących 121
Układ graficzny obiektów sterujących 122
Kotwiczenie 122
Dokowanie 124
Maksymalny i minimalny rozmiar okna 126
Automatyczne przewijanie 127
Podzielone okna 127
Obiekty sterujące będące pojemnikami 130

Obiekty sterujące i zdarzenia 131
Obsługiwanie więcej niż jednego zdarzenia 133
Przyciski Accept i Cancel 134
Badanie formularzy .NET 135
Dwa sposoby na pokazanie formularza 136
Formularze i obiekt My 138
Formularze modalne 139
Formularz startowy i tryb wyłączenia 140
Zdarzenia aplikacji 140
Osobliwości formularzy 142
Wewnętrzne działanie formularzy 144
Formularze „pod kapeluszem” Visual Basic 6 144
Formularze „pod kapeluszem” w Visual Basic 2005 146
Przechodzenie przez „bagno i breję” 147
A co z informacjami binarnymi? 149
Dynamiczne dodawanie obiektów sterujących 149
Dynamiczne podłączenie obsługi zdarzenia 151
Interakcja pomiędzy formularzami 153
Problem z interakcją przykładowego formularza 153
Okna dialogowe 155
Formularze przyporządkowane 156
Interfejsy MDI 157
Więcej obiektów sterujących w .NET 159
Paski i menu 160
Ikony powiadamiające 164
Nowy rodzaj komponentów — Provider 167
I co dalej? 168

5
PROGRAMOWANIE OBIEKTOWE 169
Nowości w .NET 170
Wprowadzenie do programowania obiektowego 171
Co to jest programowanie obiektowe? 172
Problemy z tradycyjnym programowaniem strukturalnym 172
Najpierw były sobie struktury… 173
Bardzo prosta struktura danych personalnych 174
Jak do struktury wbudować inteligencję? 175
Utworzenie instancji obiektu danej klasy 177
Obiekty — rzut oka na zaplecze 178
Klasy w kawałkach 181
Rozbudowa klasy poprzez dodanie własności 182
Własności tylko do odczytu 185
Rozbudowa klasy poprzez dodanie konstruktora 186
Konstruktory pobierające argumenty 187
Klasy z wieloma konstruktorami 188

Konstruktor domyślny 190
Destruktory 191
Mechanizm „Garbage Collection” w Visual Studio 191
Rozbudowa klasy poprzez dodanie zdarzeń 194
Zdarzenie w akcji, czyli jak to działa 195
Zdarzenia o różnych sygnaturach 198
Numeratory 200
Jak powstał pomysł na numerację? 202
Co się dzieje „pod kapeluszem” w typie wyliczeniowym? 203
Stosowanie typu wyliczeniowego w połączeniu ze zdarzeniami 205
Współużytkowane komponenty klas i obiektów 207
Współużytkowane metody 208
Współużytkowane własności 210
Zaglądamy pod maskę modułu 211
Ocena klas 212
Typy — obraz ogólny 213
Inspekcja obiektów w naszej aplikacji 214
I co dalej? 215

6
OPANOWANIE OBIEKTÓW 217
Nowości w .NET 218
Filozofia programowania obiektowego 218
Idea czarnej skrzynki 219
Luźne łączenie 220
Spójność 221
Co reprezentują klasy? 221
Dziedziczenie 222
Podstawy dziedziczenia 223
Konstruktory w klasach potomnych 225
Komponenty klas o statusie Protected 227
Przysłanianie metod 228
Casting 231
Klasy abstrakcyjne i słowo kluczowe MustInherit 232
Przysłanianie obowiązkowe MustOverride 233
Dziedziczenie wielopoziomowe 234
Czy dziedziczenie jest dobrym pomysłem? 235
Używanie dziedziczenia w celu rozszerzenia klas .NET 236
Interfejsy 241
Używanie interfejsów 243
Interfejsy i wsteczna kompatybilność 244
Używanie typowych interfejsów .NET 245
Klasy kolekcji 251
Podstawowa kolekcja 252
Klasa NuclearFamily 252

Kolekcje wyspecjalizowane 255
Kolekcje ogólne 255
I co dalej? 257

7
PLIKI SKOMPILOWANE I KOMPONENTY 259
Nowości w .NET 260
Wprowadzenie do plików skompilowanych typu „assembly” 261
Pliki skompilowane typu „assembly” kontra komponenty, które używają COM 261
Dlaczego nigdy wcześniej nie spotkaliśmy się z tymi cechami? 264
Postrzeganie programów jako plików skompilowanych typu „assembly” 265
Ustawianie informacji dotyczących pliku skompilowanego typu „assembly” 267
Pobieranie informacji o pliku skompilowanym typu „assembly” 270
Tworzenie komponentu .NET 273
Tworzenie projektu biblioteki klas 273
Tworzenie klienta 274
Globalny bufor plików skompilowanych GAC 276
GAC „pod maską” 278
Tworzenie współdzielonego pliku skompilowanego typu „assembly” 279
Pliki strategii 281
Tworzenie strategii wersji 282
Zasoby 284
Dodawanie zasobów 284
Używanie zasobów 286
I co dalej? 288

8
UODPORNIANIE NA BŁĘDY 291
Nowości w .NET 292
Skąd się biorą błędy? 293
Zasady ochrony przed błędami 295
Błędy podczas kompilacji 296
Zastosowanie opcji Option Explicit i Option Strict 298
Numery wierszy kodu 301
Narzędzia diagnostyczne w Visual Studio 301
Podglądamy nasz program w działaniu 302
Polecenia dostępne w trybie kroczącym Break Mode 304
Okno punktów krytycznych Breakpoints 305
Licznik Hit Count 307
Okna podglądu: Autos, Locals i Watch 308
Okno natychmiastowe Immediate 310
Błędy w ruchu — Runtime Errors 310
Strukturalna obsługa wyjątków 312
Mechanizm Error Call Stack 313
Ewolucja od mechanizmu On Error Goto 314
Wyjątek jako obiekt 315

Filtrowanie przy użyciu wyjątku 317
Typy wyjątków 318
Filtrowanie warunkowe 319
Wygenerowanie własnego wyjątku 320
Udoskonalenie indywidualnej klasy wyjątków 322
Ostatnia linia obrony: zdarzenie UnhandledException 323
Programowanie defensywne 325
Zasady programowania defensywnego 325
Testowanie założeń za pomocą Assert() 326
Użycie metody Debug.WriteLine() 328
Śledzenie za pomocą pliku Log oraz metody Trace 329
I co dalej? 331

9
PRACA Z DANYMI: PLIKI, DRUKOWANIE I XML 333
Nowości w .NET 334
Interakcja z plikami 335
Odczytywanie i zapisywanie plików 336
Tworzenie pliku za pomocą obiektu My 336
Tworzenie pliku za pomocą klasy FileStream 337
Klasy StreamWriter i StreamReader 338
Klasy BinaryWriter i BinaryReader 339
Dostęp do plików w stylu Visual Basic 343
Trochę więcej na temat łańcuchów 343
Kompresja plików 345
Zarządzanie plikami i folderami 347
Klasa FileInfo 347
Prosta przeglądarka katalogów 351
„Przeglądanie” systemu plików 352
Serializacja obiektu 354
Zachowywanie i pobieranie obiektu zdatnego do serializacji 355
Precyzyjnie dostrojona serializacja 356
Klonowanie obiektów za pomocą serializacji 357
Drukowanie i podgląd danych 358
Drukowanie danych z macierzy 359
Drukowanie tekstu z przełamywaniem wierszy 361
Drukowanie obrazów 363
Ustawienia drukowania 363
Podgląd wydruku 365
Praca z rejestrem 367
Pliki XML 369
Czym tak w ogóle są pliki XML? 370
Pisanie prostego dokumentu XML 372
Odczytywanie plików XML 373
Zaawansowane pliki XML 376
I co dalej? 376

10
BAZY DANYCH I ADO.NET 379
Nowości w .NET 380
Wprowadzenie do ADO.NET 381
Obsługa danych relacyjnych 381
Baza danych Northwind 382
SQL Server 2005 Express Edition 383
Model bazujący na dostawcach 383
Podstawowe obiekty ADO.NET 385
Dostęp do przodu w trybie tylko do odczytu 386
Obiekty Connection 387
Obiekty Command 390
Obiekty DataReader 392
Aktualizowanie danych za pomocą obiektów Command 395
Dlaczego warto używać obiektów Command? 396
Przykładowa aktualizacja danych 396
Wywoływanie procedur składowanych 398
Używanie poleceń z parametrami 400
Przykład transakcji 402
Używanie obiektów DataSet 403
Kiedy należy używać obiektów DataSet? 404
Zapełnianie obiektów DataSet za pomocą obiektów DataAdapter 404
Dostęp do informacji zapisanych w obiektach DataSet 405
Usuwanie rekordów 406
Dodawanie informacji do obiektów DataSet 407
Korzystanie z wielu tabel 409
Relacje między obiektami DataTable 409
Używanie obiektu DataSet do aktualizowania danych 412
Aktualizowanie źródła danych 415
Ręczne tworzenie obiektów DataSet 418
Wiązanie danych 421
I co dalej? 424

11
WĄTKI 427
Nowości w .NET 428
Wprowadzenie do wątków 428
Wątki „pod maską” 429
Porównywanie programów jedno- i wielowątkowych 430
Skalowalność i prostota 431
Zegary a wątki 431
Używanie komponentu BackgroundWorker
do wykonywania podstawowych operacji na wątkach 433
Przesyłanie danych do komponentu BackgroundWorker i pobieranie ich z niego 436
Śledzenie postępu 439
Możliwość anulowania operacji 441

Zaawansowane operacje na wątkach przy użyciu klasy Thread 442
Prosta aplikacja wielowątkowa 443
Przekazywanie danych do wątku 445
Wielowątkowość i interfejs użytkownika 447
Podstawy zarządzania wątkami 449
Metody klasy Thread 449
Priorytety wątków 452
Kiedy zbyt wiele to wciąż za mało? 452
Program wykorzystujący priorytety wątków 452
Diagnozowanie wątków 455
Synchronizacja wątków 456
Potencjalne problemy z wątkami 456
Podstawy synchronizacji 456
Przykładowy problem z synchronizacją 457
Rozwiązanie problemu za pomocą bloku SyncLock 460
I co dalej? 461

12
ASP.NET I FORMULARZE SIECIOWE 463
Nowości w .NET 464
Zarys tworzenia aplikacji sieciowych 464
Co było nie tak z klasyczną ASP? 465
Wprowadzenie do aplikacji sieciowych 466
Tworzenie aplikacji sieciowych 467
Składniki projektu ASP.NET 469
Projektowanie formularzy sieciowych 471
Podstawowe kontrolki 471
Dodawanie kontrolek do formularzy sieciowych 472
Uruchamianie strony internetowej 475
Dodawanie metody obsługi zdarzenia 477
Jak to działa? 477
Właściwość AutoPostback 478
Zdarzenia kontrolek sieciowych 478
Formularze sieciowe „pod maską” 479
Stan widoku 481
Cykl przetwarzania stron 482
Inne kontrolki 483
Rozmyślania o stanie 484
Budowa żądania 485
Wprowadzenie do problemu 485
Zapisywanie dodatkowych informacji w stanie widoku 486
Przesyłanie informacji 488
Przekazywanie informacji w łańcuchach znaków zapytania 488
Używanie stanu sesji 491
Używanie stanu aplikacji 493
Podsumowanie różnych technik zarządzania stanem 495

Wyświetlanie danych za pomocą wiązania ... 495
Podstawy wiązania danych w ASP.NET ... 496
Kontrolki działające jako źródła danych ... 497
Instalowanie witryn internetowych ... 499
Instalacja IIS ... 499
Katalogi wirtualne ... 501
I co dalej? ... 504

13
USŁUGI SIECIOWE ... 505
Nowości w .NET ... 506
Wizja interaktywnej sieci ... 506
Usługi sieciowe — model COM dla internetu? ... 507
Współczesne usługi sieciowe ... 507
Czy usługi sieciowe to obiekty? ... 508
Tworzenie pierwszej usługi sieciowej ... 508
Przygotowywanie usługi sieciowej ... 509
Projekt usługi sieciowej ... 510
Klasa usługi sieciowej ... 512
Dostęp do usługi sieciowej ... 513
Testowanie usług sieciowych ... 514
Działanie usługi sieciowej ... 515
Stosowanie otwartych standardów ... 517
XML i WSDL ... 517
SOAP ... 518
Konsumowanie usług sieciowych ... 520
Klasa pośrednicząca ... 520
Tworzenie aplikacji klienckiej ... 521
Dodawanie referencji sieciowej ... 522
Sprawdzanie klasy pośredniczącej ... 523
Używanie klasy pośredniczącej ... 525
Diagnozowanie projektów usług sieciowych ... 526
Asynchroniczne wywoływanie usług sieciowych ... 527
Obsługa asynchroniczności w klasie pośredniczącej ... 528
Przykładowy klient asynchroniczny ... 529
Anulowanie asynchronicznych żądań ... 530
I co dalej? ... 531

14
INSTALACJA I WDRAŻANIE ... 533
Nowości w .NET ... 534
Programy instalacyjne ... 534
Wymagania aplikacji .NET ... 535
ClickOnce ... 536
Publikowanie w internecie lub w sieci ... 537
Instalowanie aplikacji za pomocą ClickOnce ... 541

Aktualizowanie aplikacji ClickOnce 542
Publikowanie na płytach CD 544
Tworzenie projektu instalacyjnego Visual Studio 545
Podstawowe opcje projektów instalacyjnych 546
Okno File System 547
Okno Registry 550
Okno File Types 551
Interfejs użytkownika 553
Niestandardowe operacje 557
Warunki uruchamiania 558
I co dalej? 559

SKOROWIDZ 561

7

Pliki skompilowane i komponenty

Część spośród najistotniejszych zmian dotyczących sposobu, w jaki programiści posługujący się VB uprawiają swój biznes w świecie .NET, wynika z wprowadzenia *plików skompilowanych typu assembly* (ang. *assemblies*), uogólnionego terminu oznaczającego w .NET pliki wykonywalne aplikacji (*.exe*) oraz skompilowane komponenty. W Visual Basic 6 tworzenie i wielokrotne wykorzystywanie komponentu było często skomplikowane, zwłaszcza jeżeli chciało się współdzielić i współużytkować rezultaty swojej pracy z aplikacjami zakodowanymi w innych językach programowania. Kłopoty z rejestracją w systemie Windows i konflikty wersji — które występują, gdy dwa programy oczekują różnych wersji tego samego komponentu — pojawiają się zawsze wtedy, kiedy najmniej się tego spodziewamy, a uporanie się z nimi może zająć wiele godzin. Czytając rozdział, dowiesz się, w jaki sposób .NET rozwiązuje te przyprawiające o ból głowy problemy oraz zapoznasz się z nowym, lepszym modelem współdzielenia komponentów, który oferuje .NET. Dowiesz się także wystarczająco dużo, by stawić czoła rozdziałowi 14., w którym omówiono, w jaki sposób można utworzyć zindywidualizowane programy instalacyjne do rozpowszechniania gotowych aplikacji.

Nowości w .NET

Czy to możliwe, by nastąpił nareszcie koniec przyprawiających o ból głowy kłopotów z rozpowszechnianiem wersji, trudów związanych z wdrażaniem oraz wielokrotnych konfliktów? Niniejszy rozdział prezentuje zmiany, które pojawiają się w nowej filozofii wdrażania Microsoftu. Niektórymi z tych zmian są:

Pliki skompilowane typu assembly

Jeżeli jesteś doświadczonym programistą, widziałeś już, jak COM może ułatwić ponowne wykorzystanie kodu, pozwalając programistom na tworzenie i współdzielenie oddzielnych komponentów. Być może widziałeś także, jak wiele problemów może być spowodowane przez konflikt pomiędzy różnymi współdzielonymi komponentami oraz jak zainstalowanie jednego programu powoduje uszkodzenie innego. Pliki skompilowane typu ***assembly*** są zamiennikami Microsoftu zastępującymi komponenty i tradycyjne pliki aplikacji, zawierającymi wbudowane metadane, a zaprojektowanymi po to, by uniknąć piekła z dołączaniem dynamicznych bibliotek .DLL.

Nigdy więcej rejestrowania

Pojawienie się plików skompilowanych typu *assembly* oznacza, że już nigdy więcej nie trzeba będzie polegać na rejestrze, by zachować ważne informacje o własnych komponentach. Zamiast tego wszystkie te informacje są przechowywane bezpośrednio w plikach programu, co czyni prostym kopiowanie aplikacji i komponentów z jednego komputera na drugi oraz umożliwia użytkownikowi współdzielenie komponentów w jego programach, bez konieczności martwienia się szczegółami konfiguracji.

Strategia numeracji i zgodności wersji

.NET zapewnia użytkownikowi pełną kontrolę nad wersjami. Tylko użytkownik ma możliwość decydowania o tym, którą wersję komponentu będzie wykorzystywała jego aplikacja. Jeżeli w innej aplikacji zastosuje się zaktualizowaną wersję współdzielonego komponentu, nadal swobodnie można używać oryginalnej wersji, na której zbudowana i przetestowana została własna aplikacja, nie tracąc gwarancji niezachwianej stabilności.

Zasoby

Zachodzi konieczność użycia w aplikacji binarnego obrazu, jednak obawiasz się, co się stanie, gdy ktoś usunie Twój plik obrazu, przemieści go lub będzie chciał dokonać w nim zmian? Z rozdziału dowiesz się, w jaki sposób zamieścić obraz w postaci danych binarnych wewnątrz pliku skompilowanego aplikacji, gdzie nikt nie będzie mógł się do nich dostać. Co więcej, Visual Studio nadal będzie umożliwiało Ci bezproblemowe aktualizowanie go.

Wprowadzenie do plików skompilowanych typu „assembly”

Plik skompilowany typu assembly jest w .NET odpowiednikiem plików typu *.dll* lub *.exe*. Po prostu plik skompilowany typu „assembly” to jakieś zgrupowanie zestawu funkcji programu, które odpowiada pojedynczemu komponentowi lub aplikacji. W programach, którym przyglądaliśmy się do tej pory, wszystkie funkcje i cechy były zakodowane wewnątrz jednego pliku skompilowanego typu „assembly”, który po skompilowaniu stawał się plikiem *.exe*. Jednak jeżeli chciało się utworzyć oddzielnie dystrybuowane komponenty, trzeba było podzielić dany program na kilka oddzielnych jednostek funkcjonalnych, które stawały się następnie oddzielnymi plikami skompilowanymi typu „assembly”.

Pliki skompilowane typu „assembly” kontra komponenty, które używają COM

Na tym etapie prawdopodobnie zastanawiasz się, dlaczego programiści potrzebują plików skompilowanych typu „assembly”, choć istnieje już COM — system służący do tworzenia i współdzielenia komponentów, który jest umieszczony w systemie operacyjnym Windows. Tutaj .NET także dokonuje wyraźnego przełomu. Dzięki wprowadzeniu plików skompilowanych typu „assembly” uciążliwe problemy z konfliktami wersji, z którymi programiści borykali się przez lata, nareszcie mają się ku końcowi. Nie jest to tylko czcza gadanina — pliki skompilowane typu „assembly” posiadają kilka unikalnych cech, które czynią je całkowicie niepodobnymi do czegokolwiek, czego programiści Windows używali kiedykolwiek wcześniej.

Pliki skompilowane typu „assembly” są samoopisywalne

Najbardziej rewolucyjnym aspektem plików skompilowanych typu „assembly” jest fakt, że są samoopisywalne. Każdy plik skompilowany typu „assembly”, który się utworzy, zawiera jeden lub więcej plików programu oraz własny, specyficzny wykaz nazywany *manifestem*. Taki manifest zawiera dodatkowe informacje zwane *metadanymi* (ang. *metadata*). Metadane to „dane o danych”. Zasadniczo kod programu to dane, a metadane to informacje o programie, takie jak jego nazwa, wersja, typy dostępne publicznie oraz zależności). Manifest zastępuje bibliotekę typów i informacje z rejestru Windows, używane w połączeniu z komponentami COM. I w ten sposób przechodzimy do następnej kwestii.

Pliki skompilowane typu „assembly” nie potrzebują rejestru

Wszystkie informacje niezbędne do używania komponentu lub uruchamiania aplikacji są zawarte w manifeście pliku skompilowanego typu „assembly”, który zawarty jest dokładnie wewnątrz odpowiadającego mu pliku *.dll* lub *.exe*. Nie jest możliwym utworzenie programu w .NET bez automatycznego wygenerowania manifestu i odpowiedniego pliku skompilowanego typu „assembly”. Oznacza to, że

można skopiować aplikacje i komponenty na dowolny inny komputer używający .NET, a one automatycznie będą na nim działać. Nie ma potrzeby bawić się *regsvr32.exe* czy innymi niewygodnymi narzędziami, by dodać informacje do rejestru.

W Visual Basic 6 można było stosunkowo łatwo przetransferować prostą aplikację z jednego komputera na inny. Jeśli jednak jakaś aplikacja używała innych komponentów COM lub obiektów sterujących ActiveX, trzeba było stawić czoła kilku nowym problemom. Mianowicie zachodziła potrzeba zarejestrowania tych plików na każdym komputerze, który ich używał, a proces rejestracji mógł spowodować konflikt z innymi zainstalowanymi aplikacjami. W .NET można po prostu identyfikować i kopiować potrzebne pliki — nie zachodzi potrzeba dokonywania jakichkolwiek rejestracji. Oczywiście jeżeli zapomni się podporządkowanego, pomocniczego pliku, nadal popada się w tarapaty.

Pliki typu „assembly" mogą być prywatnie współużytkowane

Aplikacja Visual Basic 6 może używać dwóch ogólnych typów komponentów: tych, które zostały zaprojektowane wewnątrzzakładowo w celu współdzielenia zestawu funkcji specyficznych dla danej firmy, oraz takich, które zostały utworzone (a nawet mogą być sprzedawane) przez postronnych producentów komponentów lub Microsoft. Ten drugi typ komponentów wymaga pewnego rodzaju centralnego składu (ang. *central repository*). W tradycyjnym programowaniu wykorzystującym komponenty COM to katalog systemowy Windows jest miejscem, w którym w pewnym sensie w sposób niezorganizowany składowane są wszystkie te (pomocnicze) pliki. W projekcie .NET *globalny bufor pliku skompilowanego* (ang. *Global Assembly Cache*), czyli *GAC*, pełni mniej więcej tę samą funkcję. Zajmiemy się nim później w tym rozdziale.

W drugim ze scenariuszy — modułach kodu specyficznych dla danej firmy — komponenty nie muszą być współdzielone przez wszystkie aplikacje i programy znajdujące się w komputerze. Mogą być używane przez zaledwie jedną aplikację lub kilka aplikacji utworzonych przez tych samych programistów. Przy projektowaniu wykorzystującym komponenty COM nie istniał łatwy sposób implementowania tej techniki. Prywatne komponenty nadal musiały być wrzucane do katalogu systemowego wraz ze wszystkim, co oznaczało dodatkowe etapy rejestracji, niepotrzebne informacje dodawane do rejestru (takie jak unikalny identyfikator globalny, czyli GUID) oraz stos komponentów, które nie mogły być wykorzystywane ponownie przez inne programy.

Jeżeli kiedykolwiek próbowałeś przejrzeć całą listę komponentów COM i dodać niektóre z nich do swoich projektów Visual Basic, bez wątpienia zorientowałeś się, że wiele z elementów, które pojawiają się na liście, nie stanowi w istocie przykładów współdzielonych procedur ani zasobów dostarczanych przez innych projektantów aplikacji. Zamiast tego są one zaprojektowane z myślą o używaniu ich wraz z konkretną aplikacją, która jest zainstalowana na Twoim komputerze, i w zasadzie są bezużyteczne poza tym programem. Mogą nawet posiadać ograniczenia licencyjne, które uniemożliwiać będą utworzenie egzemplarza takiego komponentu w Twoich aplikacjach.

W .NET prywatne pliki skompilowane typu „assembly" są prywatne. Przechowujesz te komponenty w katalogu aplikacji lub podkatalogu aplikacji.

Pliki skompilowane typu „assembly" są rygorystycznymi tropicielami wersji

Manifest zapisuje także informacje dotyczące aktualnych wersji wszystkich dołączonych plików. Za każdym razem, gdy kompiluje się program VB 2005, informacje o wersjach są automatycznie wpisywane do manifestu, co oznacza, iż nie istnieje zagrożenie, że dane te staną się nieaktualne lub nie będą zsynchronizowane z przedmiotowym kodem aplikacji. Manifest zawiera także krótki blok kodu kryptograficznego, wygenerowanego w oparciu o wszystkie pliki znajdujące się w pliku skompilowanym typu „assembly". Za każdym razem, gdy uruchamia się plik skompilowany typu „assembly", wspólne środowisko uruchomieniowe (CLR) sprawdza, czy ten kod kryptograficzny odpowiada stanowi faktycznemu. Jeżeli wykryta zostaje jakaś zmiana, która nie jest obecna w manifeście (co nie jest możliwe, jeżeli plik nie jest uszkodzony ani nie został zmodyfikowany przez złośliwego użytkownika za pomocą narzędzia niskiego poziomu), wspólne środowisko uruchomieniowe nie zezwala na uruchomienie aplikacji.

Widać tu wyraźną różnicę w działaniu w stosunku do komponentów COM oraz obiektów sterujących ActiveX. W przypadku komponentów COM trzeba wierzyć, że informacje w rejestrze i wszystkie powiązane biblioteki typów są zaktualizowane — a wiara ta może nie zostać nagrodzona.

Pliki assembly umożliwiają równoczesną pracę z różnymi wersjami

Ile razy zdarzyło Ci się zainstalować nową aplikację tylko po to, by odkryć, że nadpisała ona jakiś plik potrzebny innej aplikacji jego nowszą wersją, która załamała wsteczną kompatybilność? Bez względu na to, jak bardzo programiści będą się starali, idealna wsteczna kompatybilność nigdy nie zostanie osiągnięta w sposób uniwersalny, a każdy system, który używa pojedynczego komponentu oraz jednej wersji dla dziesiątek różnych aplikacji, wpędzi użytkownika w kłopoty (lub do piekła z dołączaniem dynamicznych bibliotek DLL, jak często żartobliwie nazywa się taką sytuację).

.NET obchodzi ten problem na dwa sposoby. Pierwszym rozwiązaniem są prywatne pliki skompilowane typu „assembly". Ponieważ każda aplikacja posiada swoją własną, odrębną kopię danego komponentu, można swobodnie aktualizować pliki *.dll* w dowolnym momencie, gdy tylko zechcemy. Od użytkownika zależy, czy zmiana dotyczyć będzie tylko jednej aplikacji, czy ich tuzina.

Oczywiście taki podział nie pomoże, jeżeli podejmie się decyzję, by współdzielić komponenty we wszystkich aplikacjach znajdujących się w komputerze, umieszczając je w globalnym buforze komponentów GAC. Na szczęście .NET pomaga także tutaj uniknąć problemów, prezentując przełomową właściwość o nazwie *równoczesne wykonywanie obok siebie* (ang. *side-by-side execution*). Dzięki temu systemowi można instalować wiele wersji jednego komponentu w GAC.

Kiedy uruchamia się jakąś aplikację, .NET używa wersji komponentu, która była zaprojektowana dla tej aplikacji. Jeżeli równocześnie uruchamia się inny program, który używa równocześnie tego samego komponentu, wspólne środowisko uruchomieniowe (CLR) załaduje także odpowiednią wersję komponentu dla tego programu. Nie nastąpi żadne niespodziewane zachowanie ani brak kompatybilności, ponieważ każda aplikacja używa zestawu komponentów, dla których została zaprojektowana.

Dlaczego nigdy wcześniej nie spotkaliśmy się z tymi cechami?

To, że Microsoft utworzył COM takim, jakim go znamy, uwielbiamy oraz nienawidzimy, a w którym przeżywa się koszmary z różnymi wersjami komponentów, nie było spowodowane krótkowzrocznością. Niektóre z cech używanych przez pliki typu „assembly" w przeszłości nie były po prostu stosowane. Na przykład wykonywanie obok siebie może znacząco zwiększyć ilość wymaganej pamięci, gdy kilka aplikacji działa równocześnie. Każda z aplikacji może używać innej wersji tego samego komponentu, co w praktyce jest równoznaczne z sytuacją, gdy każda aplikacja używa całkowicie odrębnego komponentu. W dzisiejszych czasach stosunkowo proste jest dokupienie kilkuset więcej megabajtów pamięci RAM, by zapobiec takiemu problemowi, jednak w przeszłości system operacyjny zaprojektowany bez brania pod uwagę współdzielenia kodu szybko utknąłby w martwym punkcie. Podobnie umożliwienie śledzenia i przechowywania wielu wersji oddzielnie na komputerze (lub dostarczenie każdej aplikacji jej własnej kopii jakiegoś komponentu z prywatnymi plikami typu „assembly") dawniej nie było po prostu efektywne (w rozumieniu racjonalnego wykorzystania zasobów sprzętowych) w przypadku, gdy posiadało się niewielką ilość miejsca na twardym dysku. W dzisiejszych czasach, gdy pamięć fizyczna jest wręcz absurdalnie tania, ogólny koszt nie jest już tak wielkim problemem.

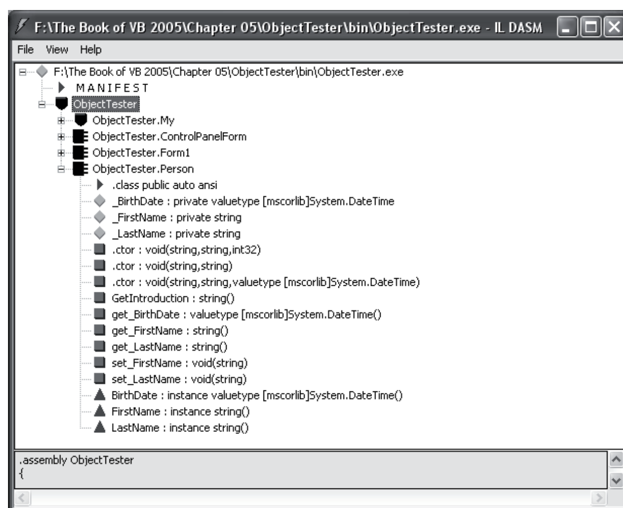
Innymi słowy, COM oraz cała platforma Windows zostały utworzone z myślą o pojedynczym, scentralizowanym repozytorium komponentów. Skoncentrowano się na zaoszczędzeniu miejsca i pamięci, by zapewnić lepsze działanie, a nie na ułatwianiu tworzenia aplikacji i ułatwianiu życia programisty, zamiast czynić jedno i drugie łatwiejszym, wygodniejszym i bardziej spójnym logicznie. Dzisiaj coraz więcej aplikacji o znaczeniu krytycznym jest projektowanych w środowisku Windows, które zmieniło się z zabawki dla użytkowników komputerów domowych w profesjonalną platformę biznesową. Aktualnie nacisk kładzie się na niezawodność oraz strukturalne zabezpieczenia przed występowaniem błędów, nawet jeżeli kilka megabajtów musi zostać zmarnowane w tym procesie. Ostatecznie nowoczesne komputery posiadają spore zasoby do wykorzystania. Wszystko to sprowadza się do jednej zasady — programowanie w .NET jest programowaniem *nowoczesnym*.

Postrzeganie programów jako plików skompilowanych typu „assembly”

Jak już wcześniej wspominano, wszystkie aplikacje, które do tej pory tworzyliśmy, są prawdziwymi plikami skompilowanymi typu „assembly” środowiska .NET. Gdyby nimi nie były, wspólne środowisko uruchomieniowe (CLR) nie chciałoby ich wykonywać. Aby zobaczyć, jak nasze programy wyglądają jako pliki skompilowane typu „assembly”, można użyć interesującego programu o nazwie *ILDasm.exe* (IL Disassembler), który można znaleźć w katalogu takim jak *C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin*, w zależności od tego, gdzie zainstalowało się środowisko .NET Framework oraz jakiej wersji Visual Studio się używa. Najprostszym sposobem uruchomienia *ILDasm* jest przejście najpierw do wiersza poleceń Visual Studio (należy wybrać *Programs/Visual Studio 2005/Visual Studio Tools/Visual Studio 2005 Command Prompt* — czyli Programy/Visual Studio 2005/Narzędzia Visual Studio/Wiersz poleceń Visual Studio 2005), po czym wpisać `ildasm` w wierszu poleceń i nacisnąć *Enter*.

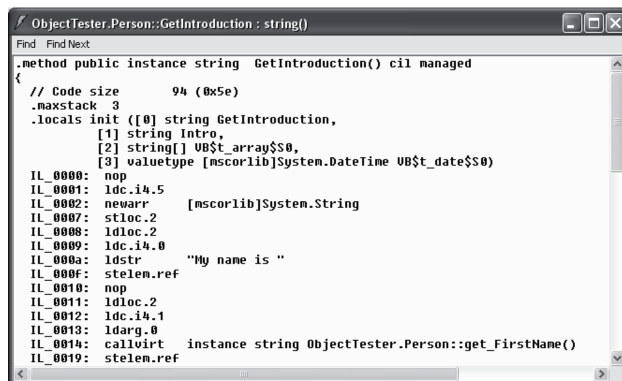
Kiedy już uruchomi się *ILDasm*, można otworzyć dowolny plik skompilowany typu „assembly” środowiska .NET (plik *.exe* lub *.dll*). Należy po prostu wybrać *File/Open* i wybrać właściwy plik. W następnym przykładzie pokazano działanie narzędzia *ObjectTester* opisanego w rozdziale 5.

ILDasm używa typowej, znanej struktury drzewiastej, by wyświetlić informacje dotyczące programu. Wszystkie typy, które są określone w naszych projektach, są automatycznie definiowane w metadanych znajdujących się w pliku skompilowanym typu „assembly” naszego programu. Powoduje to, że przeglądanie określonej definicji naszej klasy `Person` jest proste, co pokazano na rysunku 7.1.



Rysunek 7.1. Klasa Person rozłożona na czynniki pierwsze

Jeżeli dwukrotnie kliknie się metodę lub własność, ujrzy się listę związanych z nią instrukcji .NET, która została utworzona w oparciu o kod programu, co zostało zaprezentowane na rysunku 7.2.



Rysunek 7.2. Kod IL dla metody GetIntroduction()

Jeżeli kiedykolwiek wcześniej używałeś jakiegoś innego narzędzia desasemblującego, najprawdopodobniej zauważyłeś, że kod .NET wyraźnie się różni. Zwykle wszystko, na co można liczyć patrząc na skompilowany program, to znalezienie listy instrukcji w języku maszynowym niskiego poziomu. .NET dla odmiany kompiluje programy do specjalnego pośredniego języka o nazwie *IL*. Instrukcje IL nie wyglądają tak samo jak normalny kod Visual Basic 2005, lecz zachowują wystarczająco dużo podobieństw, by często można było zorientować się ogólnie, co się dzieje we fragmencie kodu.

Ze względu na to, że instrukcje IL zachowują tak wiele informacji na temat programu, wspólne środowisko uruchomieniowe (CLR) może dokonywać optymalizacji, chronić przed niedozwolonymi operacjami oraz chronić pamięć podczas działania aplikacji. Jednakże zachowywanie wszystkich tych informacji powoduje także, że stosunkowo łatwym dla innych programistów staje się zaglądanie do niektórych wewnętrznych detali dotyczących działania konkurencyjnego programu. Jest to problem, który przez jakiś czas nękał aplikacje Java, a rozwiązanie .NET jest podobne do tego zastosowanego w Javie. Mianowicie, jeżeli kod do odczytu jest istotny dla bezpieczeństwa, trzeba użyć narzędzia od postronnego dostawcy do tak zwanego *zaciemniania*, które umożliwia programiście zaszyfrowanie kodu w taki sposób, że staje się on trudny do zinterpretowania dla ludzi. (Na przykład jedną z technik jest nadanie wszystkim zmiennym nic nieznaczących identyfikatorów liczbowych w skompilowanym pliku programu). Pełna wersja Visual Studio oferuje pomniejszoną wersję jednego z popularnych narzędzi zaciemniających zrozumiałość kodu, o nazwie *Dotfuscator*.

Informacje o zależnościach

Spoglądając ponownie na rysunek 7.1, na szczyt drzewa *ILDasm*, zobaczysz element, który reprezentuje manifest Twojego pliku skompilowanego typu „assembly". Jeżeli dwukrotnie się go kliknie, ujrzy się informacje takie jak wersja pliku skompilowanego typu „assembly" i ustawienia lokalne. Jednak najważniejsze jest to, że można tu także zobaczyć informacje dotyczące zależności, występujące w postaci instrukcji `.assembly extern`, które wyglądają następująco:

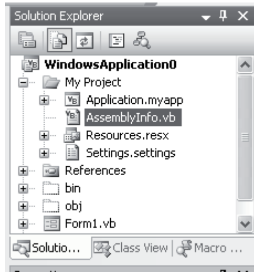
```
.assembly extern System.Data
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 ... )
  .ver 2:0:2411:0
}
```

Przykład ten wskazuje, iż by funkcjonować poprawnie, aktualny plik skompilowany typu „assembly" wymaga pliku assembly o nazwie *System.Data* zawartego w .NET. Występuje tutaj także dodatkowa informacja, która określa wymaganą wersję pliku skompilowanego typu „assembly" o nazwie *System.Data*: 2.0.2441.0. Numer ten występuje w formacie *major.minor.build.revision* (główny.drugorzędny.seria.weryfikacja). Kiedy uruchamia się taką aplikację, następuje poszukiwanie pliku z dokładnie takim numerem wersji. Domyślnie, jeżeli taka wersja nie istnieje, aplikacja po prostu odmawia uruchomienia. Późniejsze wersje nie będą używane automatycznie, jeżeli nie skonfiguruje się jawnie innej polityki wersji poprzez utworzenie pliku konfiguracyjnego (co omówione zostanie później w bieżącym rozdziale). Ryzyko załamania kodu przez komponenty, które zdają się być wstecznie kompatybilne, lecz w istocie nie są, jest po prostu zbyt wysokie.

Ustawianie informacji dotyczących pliku skompilowanego typu „assembly"

Informacje o zależnościach są dodawane automatycznie do manifestu podczas tworzenia pliku skompilowanego typu „assembly". Jednak co z innymi informacjami, takimi jak nazwa produktu oraz numer wersji? Wszystkie te szczegóły są określane w specjalnym pliku, który zawiera każdy projekt Visual Basic 2005. Nosi on nazwę *AssemblyInfo.vb*.

Normalnie plik *AssemblyInfo.vb* jest ukryty. Jednak jeżeli naprawdę zżera nas ciekawość, wybieramy *Project/Show All Files* (Projekt/Pokaż wszystkie pliki), po czym szukamy pliku w węźle *My Project* w wyszukiwarce rozwiązań Solution Explorer (patrz rysunek 7.3). Wewnątrz tego pliku zobaczyć można liczne atrybuty pliku skompilowanego typu „assembly".

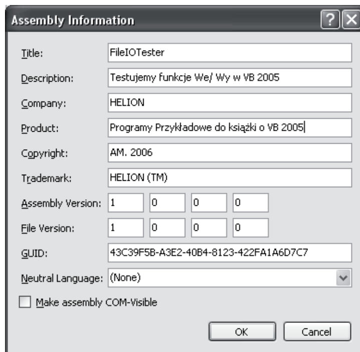


Rysunek 7.3. Ukryty plik AssemblyInfo.vb

Każdy z atrybutów w pliku *AssemblyInfo.vb* wbudowuje do skompilowanego pliku „assembly" pojedynczy fragment informacji. Poniżej pokazano, jak wyglądają atrybuty (z dodanymi kilkoma przykładowymi informacjami):

```
<Assembly: AssemblyTitle("FileIOTester")>
<Assembly: AssemblyDescription("Test VB 2005 I/O features.")>
<Assembly: AssemblyCompany("No Starch Press, Inc.")>
<Assembly: AssemblyProduct("The No Starch VB 2005 Examples")>
<Assembly: AssemblyCopyright("Copyright 2006")>
<Assembly: AssemblyTrademark("No Starch(TM)")>
```

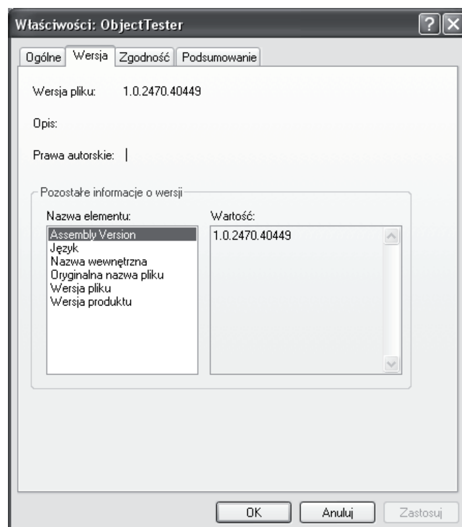
Można ręcznie zmienić wszystkie te detale, jednak nie ma takiej potrzeby; te same informacje są dostępne w kreatorze Visual Studio. Wystarczy po prostu kliknąć dwukrotnie węzeł *My Project* w wyszukiwarce rozwiązań Solution Explorer, wybrać etykietę *Application*, po czym kliknąć przycisk *Assembly Information*. Rysunek 7.4 przedstawia, co wtedy zobaczymy.



Rysunek 7.4. Modyfikowanie w prosty sposób metadanych pliku skompilowanego typu „assembly"

Podsumowując, opcje te zastępują własności projektu używane w klasycznych wersjach Visual Basic. Jeżeli chciałoby się zobaczyć, gdzie informacje te trafiają, należy po prostu skompilować aplikację i uruchomić Eksplorator Win-

dows. Należy przejść do odpowiedniego pliku, kliknąć go prawym klawiszem myszy oraz wybrać *Properties* (właściwości). Ujrzy się okno takie, jak to pokazane na rysunku 7.5.



Rysunek 7.5. Informacje o pliku skompilowanym „assembly”

Jeżeli przewinie się do dołu ekran pliku *AssemblyInfo.vb* (lub spojrzy na dół okna dialogowego *Assembly Information* na rysunku 7.4), znajdzie się najważniejszy fragment informacji. Jest to numer wersji, a w nowym projekcie domyślnie wygląda on następująco:

```
<Assembly: AssemblyVersion("1.0.0.0")>
<Assembly: AssemblyFileVersion("1.0.0.0")>
```

Postarajmy się zignorować fakt, że numer wersji jest w zasadzie określany dwukrotnie. Powodem występowania tego dziwactwa jest to, że pierwszy z atrybutów (atrybut `AssemblyVersion`) nadaje oficjalny numer wersji .NET, podczas gdy drugi z nich (atrybut `AssemblyFileVersion`) nadaje wersję, którą starsze aplikacje Windows, wliczając w to Eksplorator Windows, będą widzieć w pliku. Jeżeli tylko nie planujesz naprawdę dziwnego żartu (lub nie masz jakiegoś problemu ze wsteczną kompatybilnością, który próbujesz rozwiązać), te numery wersji powinny zawsze pozostawać takie same. W świecie .NET pierwszy numer wersji jest najważniejszy.

Jeżeli nic się nie zrobi, wersja naszej aplikacji będzie stale ustawiona jako 1.0.0.0, bez względu na to, ile razy się ją przekompiluje. Jeżeli chce się utworzyć nową wersję, trzeba otworzyć plik *AssemblyInfo.vb* i zmienić tę wartość na jakąś inną. Na przykład po utworzeniu nowej wersji ze stosunkowo niewielką ilością zmian można zmienić numer wersji na 1.1.0.0.

Nikt nie lubi zmieniania numeru wersji dla każdej kompilacji i konsolidacji, czy to poprzez edytowanie pliku *AssemblyInfo.vb*, czy używanie okna dialogowego *Assembly Information*. Jest to po prostu zbyt niewygodne (i zbyt łatwo jest zapomnieć to zrobić, co prowadzi do powstania różnych wersji aplikacji o takim samym numerze wersji). Większość programistów preferuje posiadanie jakiegoś autoinkrementującego numeru. Dzięki temu każdy plik skompilowany typu „assembly" posiada niepowtarzalny numer wersji.

W .NET łatwo jest używać takiego autoinkrementującego numeru wersji. Należy po prostu użyć gwiazdki (*) przy numerze wersji, co pokazano poniżej:

```
<Assembly: AssemblyVersion("1.0.*")>
<Assembly: AssemblyFileVersion("1.0.*")>
```

Użycie gwiazdki informuje Visual Studio, że ta aplikacja zawsze będzie wersją 1.0 (są to komponenty główny i drugorzędny numeru wersji), jednak Visual Studio powinno inkrementować trzeci i czwarty komponent numeru wersji podczas każdej kompilacji i tworzenia nowego pliku wykonywalnego *.exe* (nawet jeżeli robi się to tylko w celach testowych, klikając przycisk uruchamiania w IDE). Warto zwrócić uwagę, że numery te są inkrementowane o więcej niż jeden. Typowy numer wersji powstały przy użyciu tego systemu może wyglądać mniej więcej tak jak ten: 1.0.594.21583.

Po zakończeniu testowania zwykle zmienia się główny i drugorzędny komponent numeru wersji, by utworzyć plik, który będzie wyraźnie rozróżnialny jako nowy produkt. Jednak w trakcie testowania numery kompilacji i konsolidacji (trzeci i czwarty komponent) oraz weryfikacji mogą pomagać w rozróżnianiu różnych wersji.

Pobieranie informacji o pliku skompilowanym typu „assembly"

Czasami przydatna jest możliwość pobrania programowo informacji o pliku skompilowanym typu „assembly". Najbardziej oczywistym przykładem jest okienko *About*, w którym programy wyświetlają informacje takie jak ich aktualna wersja. Informacja taka nie może być zakodowana na sztywno w programie, ponieważ zachodziłaby potrzeba jej zmiany przy każdej kompilacji i konsolidacji oraz nie zapewniałaby ona dokładności. Zamiast tego informacja może zostać pobrana poprzez użycie klasy `System.Windows.Forms.Application`, która jest podobna do obiektu `App` obecnego w poprzednich wersjach Visual Basic. (Ta sama informacja jest także dostępna poprzez użycie obiektu `My`).

```
lblProductName.Text = Application.ProductName
lblProductVersion.Text = Application.ProductVersion
lblPath.Text = Application.ExecutablePath
```

W większości przypadków klasa ta dostarcza podstawowych cech, które są potrzebne. Jednakże można także zagłębić się w bardziej interesujący obszar za pomocą klasy `System.Reflection.Assembly`. Klasa ta używa refleksji w celu pobrania informacji na temat pliku skompilowanego typu „assembly". *Refleksja* (ang. *Reflection*) jest cechą, która umożliwia użytkownikowi wejrzenie w plik skompilowany typu „assembly" w ruchu programu, i pobranie informacji dotyczących struktury kodu znajdującego się wewnątrz. Na przykład można użyć refleksji, by dowiedzieć się, jakie klasy znajdują się w pliku skompilowanym typu „assembly" lub jakie metody znajdują się w jakiejś klasie, chociaż nie można pobrać rzeczywistego kodu. Na pierwszy rzut oka refleksja jest trochę zawiłą sztuczką, polegającą na badaniu jednego fragmentu kodu za pomocą innego fragmentu kodu.

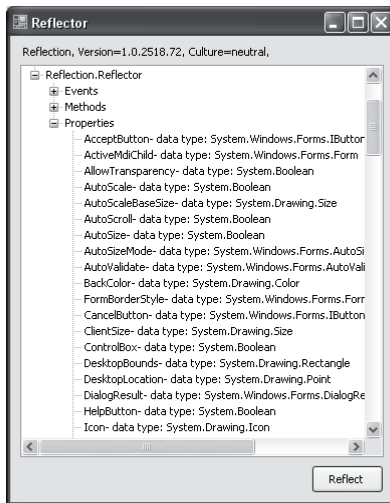
Na początek można użyć metody `GetExecutingAssembly()`, by zwrócić referencje do obecnego pliku skompilowanego typu „assembly" dla projektu. Nasz następny przykład używa pliku skompilowanego typu „assembly" i pobiera wszystkie zdefiniowane typy (wliczając w to klasy i inne konstrukcje, takie jak wyliczenia). Następnie kod przeszukuje każdą klasę w celu znalezienia listy jej metod, zdarzeń oraz własności. Podczas tego procesu konstruowany jest obiekt `TreeView`.

```
Dim MyAssembly As System.Reflection.Assembly
MyAssembly = System.Reflection.Assembly.GetExecutingAssembly()
lblAssemblyInfo.Text = MyAssembly.FullName

' Definiujemy zmienne stosowane do poruszania się w strukturze programu.
Dim MyTypes(), MyType As Type
Dim MyEvents(), MyEvent As System.Reflection.EventInfo
Dim MyMethods(), MyMethod As System.Reflection.MethodInfo
Dim MyProperties(), MyProperty As System.Reflection.PropertyInfo
' Wykonujemy iterację poprzez klasy programu
MyTypes = MyAssembly.GetTypes()
For Each MyType In MyTypes
    Dim nodeParent As TreeNode = treeTypes.Nodes.Add(MyType.FullName)
' Wykonujemy iterację poprzez zdarzenia w obrębie każdej spośród klas
    Dim node As TreeNode = nodeParent.Nodes.Add("Events")
    MyEvents = MyType.GetEvents
    For Each MyEvent In MyEvents
        node.Nodes.Add(MyEvent.Name)
    Next
' Wykonujemy iterację poprzez metody w obrębie każdej spośród klas
    node = nodeParent.Nodes.Add("Methods")
    MyMethods = MyType.GetMethods()
    For Each MyMethod In MyMethods
        node.Nodes.Add(MyMethod.Name)
    Next
' Wykonujemy iterację poprzez własności w obrębie każdej spośród klas
    node = nodeParent.Nodes.Add("Properties")
    MyProperties = MyType.GetProperties
```

```
    For Each MyProperty In MyProperties
        node.Nodes.Add(MyProperty.Name)
    Next
Next
```

Efektem końcowym jest powstanie obiektu `TreeView`, który odwzorowuje zgrubny obraz struktury programu, prezentując każdą klasę znajdującą się w aplikacji wraz z jej elementami (rysunek 7.6). W tym przypadku znaleźć można kilka automatycznie wygenerowanych klas (używanych do obsługi obiektów `My`) oraz indywidualną klasę formularza. Projekt ten (noszący nazwę *Reflection*) można znaleźć wśród przykładów odnoszących się do tego rozdziału, zamieszczonych na stronie internetowej niniejszej książki.



Rysunek 7.6. Informacje zawarte w oknie Reflection

Refleksja jest przydatna w niezliczonej ilości nietypowych scenariuszy, jednak zwykle nie jest stosowana jako część codziennego projektowania. Czasami ciekawie jest poeksperymentować z refleksją tylko po to, by lepiej zrozumieć, w jaki sposób działa mechanizm .NET, a także w jaki sposób klasyfikuje i organizuje on typy oraz metadane.

Można także dojść do wniosku, że warto będzie bardziej dokładnie przyjrzeć się refleksji. Należy rozpocząć od wykorzystania klasy `Assembly` i pozwolić jej na poprowadzenie się do innych klas „informacyjnych" znajdujących się w przestrzeni nazw `System.Reflection`. Każda z tych klas jest zindywidualizowana tak, by dostarczać informacji na temat specjalnego typu konstrukcji kodu i jej metadanych (na przykład przedstawiony wyżej kod używa klasy *EventInfo*, która dostarcza szczegółowych informacji na temat sygnatur poszczególnych zdarzeń).

Tworzenie komponentu .NET

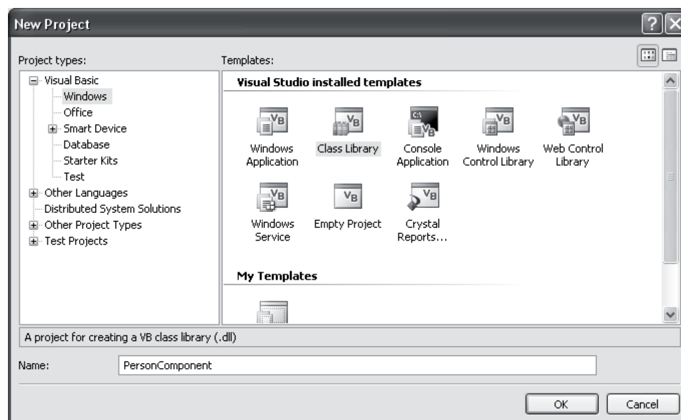
Dobrym sposobem, by zrozumieć pliki skompilowane typu „assembly”, jest samodzielne utworzenie prostego komponentu. Tak jak i we wcześniejszych wersjach Visual Basic *komponent* jest kolekcją składającą się z jednej lub więcej klas, które zawierają zestaw powiązanych ze sobą funkcji i cech. Klasy te są dostarczane przez plik *.dll*, a klient może tworzyć obiekty oparte na tych klasach tak, jakby definicja klasy była częścią aktualnego projektu.

UWAGA *W wielu przypadkach projektanci .NET używają terminów komponent i plik skompilowany typu assembly przemiennie. Technicznie rzecz biorąc, plik skompilowany typu „assembly” jest skompilowanym komponentem .NET. Jednakże określenie komponent jest używane także w luźniejszym, mniej formalnym znaczeniu, w odniesieniu do dowolnego pakietu odpowiedniego kodu.*

Tworzenie projektu biblioteki klas

Klasy `Person` oraz `NuclearFamily`, które widzieliśmy w rozdziałach 5. i 6., mogłyby stanowić podstawę logicznego komponentu. Aktualnie te definicje klas są umieszczane wewnątrz projektu formularzy Windows, który został utworzony w celu ich testowania. Poprzez rozpakowanie tych klas do oddzielnego komponentu uzyskuje się oddzielnie rozprowadzany, współdzielny komponent, który może być używany w dowolnym typie aplikacji, wliczając w to witrynę WWW ASP.NET oraz usługi sieciowe (patrz rozdziały 12. i 13., by uzyskać więcej informacji na temat tych typów projektów). W projektach na małą skalę ten szablon — gdzie jakaś klasa jest projektowana wewnątrz projektu, po czym przetwarzana w oddzielny komponent — jest typowy.

Aby utworzyć komponent, należy wybrać z menu Visual Studio *File/New/Project* (Plik/Nowy/Projekt). Następnie należy wybrać typ projektu *Class Library* (biblioteka klas) z grupy projektów Visual Basic (patrz rysunek 7.7).



Rysunek 7.7. Tworzenie biblioteki klas

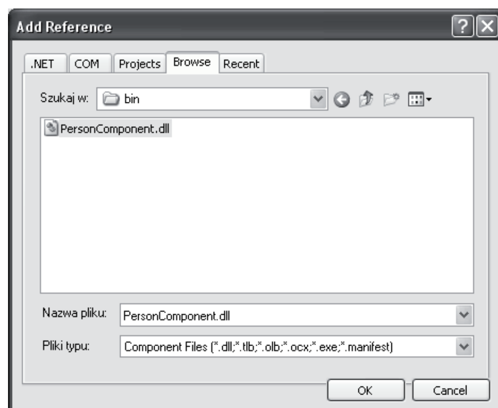
Zostanie utworzony zwykły projekt VB 2005, nieposiadający żadnych komponentów graficznych, takich jak formularze Windows. Możesz użyć tylu plików, ilu zechcesz, do utworzenia klas. W naszym przykładzie biblioteka klas będzie używała kodu, który już wcześniej został opracowany w ramach innego projektu. W celu przetransferowania kodu można importować do projektu istniejące pliki *.vb* lub można otworzyć inny egzemplarz Visual Studio, otworzyć projekt źródłowy, a następnie wyciąć i wkleić odpowiednie definicje klas dla klas `NuclearFamily` oraz `Person`.

Kiedy kod jest skopiowany, trzeba już tylko zbudować projekt. Można zbudować go, klikając standardowy przycisk *Start*. Plik *.dll* z nazwą projektu zostanie skompilowany w katalogu *bin*, a my otrzymamy ostrzeżenie informujące, że projekt może być tylko kompilowany, bez możliwości wykonania. W przeciwieństwie do niezależnych aplikacji komponenty potrzebują klienta, który ich używa, i nie mogą one wykonywać czegokolwiek samodzielnie. Aby pominąć komunikat o błędzie, należy po prostu kliknąć prawym klawiszem myszy projekt w wyszukiwarce rozwiązań Solution Explorer, po czym kliknąć przycisk *Build* za każdym razem, gdy chcemy wygenerować plik skompilowany typu „assembly".

Tworzenie klienta

W celu utworzenia klienta należy rozpocząć działanie albo od otwarcia istniejącego projektu, albo od rozpoczęcia projektu nowej aplikacji z formularzem Windows. Następnie klikamy prawym klawiszem myszy projekt w wyszukiwarce rozwiązań Solution Explorer i wybieramy opcję *Add Reference* (Dodaj referencje).

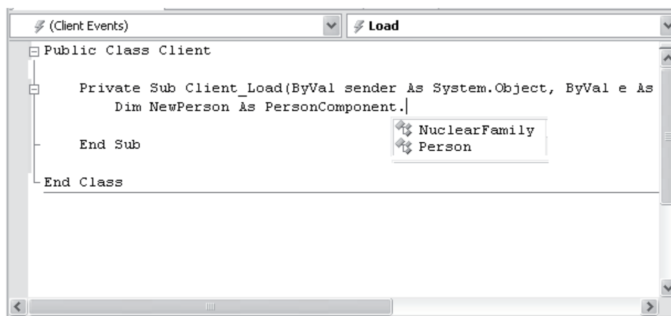
Aby dodać referencje, należy kliknąć zakładkę *Browse* (Przeglądaj), a następnie wybrać plik *.dll* z własnego projektu biblioteki klas, tak jak to pokazano na rysunku 7.8. Plik *.dll* znajdziemy w katalogu *bin*, wewnątrz folderu projektu (na przykład *PersonComponent*). Wybieramy plik, po czym klikamy *OK*.



Rysunek 7.8. Dodawanie referencji do pliku PersonComponent.dll

WSKAZÓWKA *Technika, która została tu przedstawiona (dodawanie referencji przy użyciu pliku skompilowanego), jest dobrym sposobem, by dowiedzieć się więcej na temat komponentów .NET, ponieważ dokładnie widać, co się dzieje. Jeśli jednak planuje się projektowanie klienta oraz komponentu równocześnie, istnieje jeszcze prostszy sposób, by debugować oba z nich. Należy rozpocząć od dodania obu projektów do tego samego rozwiązania. (Trzeba upewnić się, że ustawiło się klienta jako projekt otwierany przy uruchamianiu, klikając go prawym klawiszem myszy w wyszukiwarce rozwiązań Solution Explorer i wybierając opcję Set As StartUp Project). Powoduje to, że obydwa własne kody są łatwo dostępne do edycji. Następnie, gdy referencja jest już dodana, wybieramy ją na zakładce Projects. W ten sposób za każdym razem, gdy zmienimy cokolwiek w komponencie, nowa wersja zostanie skompilowana dla klienta automatycznie.*

Kiedy już wybrany zostanie właściwy plik, trzeba wcisnąć *OK*, by móc kontynuować. Visual Studio skopiuje plik *.dll* do katalogu *bin* aktualnie używanego projektu. Klasy zdefiniowane w projekcie biblioteki klas będą teraz automatycznie dostępne w aktualnie otwartym projekcie, dokładnie tak, jakby były zdefiniowane w tym projekcie. Jedyną różnicą jest to, że trzeba używać przestrzeni nazw (namespace) biblioteki danego projektu, by uzyskać dostęp do jej klas, tak jak pokazano to na rysunku 7.9.



Rysunek 7.9. Używanie pliku PersonComponent.dll

WSKAZÓWKA *W celu upewnienia się, że komponent został dodany, tak jak się spodziewaliśmy, można wybrać z menu Project/Show All Files (Projekt/Pokaż wszystkie pliki) oraz sprawdzić grupę References (Referencje) w wyszukiwarce rozwiązań Solution Explorer. Nasz komponent powinien znajdować się na liście jako jedna z referencji dodanych do naszego projektu.*

Oto najlepszy przykład skuteczności współdzielenia kodu, które z łatwością przełamuje wszelkie bariery pomiędzy językami. Można pracować z klasami C# w VB 2005 i vice versa, nie mając nawet pojęcia o różnicy. Można nawet dziedziczyć od klasy i rozszerzać ją w pliku *.dll*, używając dowolnego języka .NET. Rzeczywisty plik skompilowany „assembly" *.dll* jest obojętny językowo, a jego kod jest przechowywany w specjalnych instrukcjach IL, jak te, które widzieliśmy w programie *ILDasm*.

Kiedy decydujemy się zainstalować nasz nowy projekt klienta, jeszcze raz przekonujemy się, jak drastyczne są różnice pomiędzy COM i .NET. Podczas projektowania w COM najpierw trzeba było skopiować indywidualny komponent do katalogu systemowego Windows na nowym komputerze, zarejestrować go, po czym użyć programu-klienta. Podczas projektowania w .NET nie zachodzi potrzeba rejestrowania. Pliki *.dll* biblioteki klas oraz pliki *.exe* klienta mogą być kopiowane do dowolnego katalogu na nowym komputerze i będą działały automatycznie. Plik *.exe* posiada w swoim manifeście wszystkie informacje o zależnościach, których potrzebuje .NET. .NET będzie lokalizowało automatycznie pliki *.dll* biblioteki klas tak długo, dopóki będą się znajdować w tym samym katalogu.

Globalny bufor plików skompilowanych GAC

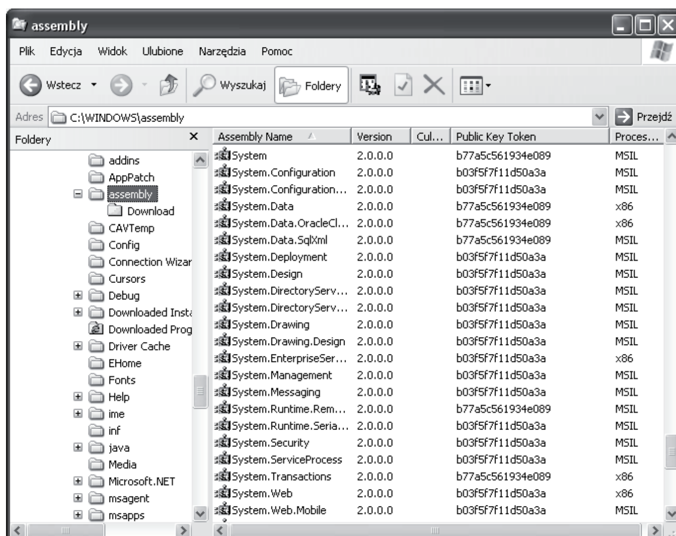
Nie wszystkie pliki skompilowane typu „assembly" są prywatne. Na przykład nikt nie chciałby instalować w katalogu oddzielnej wersji biblioteki klas .NET dla każdej aplikacji .NET. Aby uniknąć takich sytuacji, Microsoft nadal dodaje centralne repozytorium komponentów. Tym razem zamiast wrzucać pliki do katalogu systemowego, dzielonego ze sterownikami i innymi narzędziami systemowymi, Windows wykorzystuje specjalnie im poświęcony obszar, noszący nazwę Globalnego bufora plików skompilowanych (ang. *Global Assembly Cache — GAC*). GAC różni się od katalogu systemowego Windows także pod wieloma innymi istotnymi względami. Na przykład zezwala, a właściwie zachęca do instalowania wielu wersji tego samego komponentu. Nie blokuje plików w taki sposób, żeby nie można było ich aktualizować podczas ich używania; przeciwnie, zachowuje starsze wersje dla dotychczasowych użytkowników, a jednocześnie zapewnia nową, zaktualizowaną wersję dla nowych użytkowników (lub wtedy, gdy program zostanie uruchomiony ponownie). I w końcu umożliwia użytkownikowi zastosowanie zindywidualizowanej polityki używania różnych wersji, która decyduje, jakiej dokładnie wersji komponentu będzie używała dana aplikacja.

Jednak nawet pomimo wszystkich tych zalet powinno się próbować oprzeć pokusie, by zacząć wrzucać wszystkie komponenty do GAC. Wiele z czynników, które wymuszały globalne składowanie komponentów w przeszłości, po prostu nie odnosi się do świata .NET. W COM komponent musiał być częścią globalnego rejestru, by można było go używać. W przypadku .NET tak nie jest. Jeżeli tworzy się własny komponent, nie zachodzi potrzeba używania GAC. W zasadzie nawet jeżeli zakupi się komponent od producenta postronnego, by używać go w swoich aplikacjach, umieszczanie go w GAC może okazać się bezcelowe. Prawdopodobnie bardziej opłaci się po prostu skopiować dany plik do katalogu aplikacji. Należy pamiętać, że filozofia .NET ceni sobie wyżej łatwe, bezproblemowe wdrażanie, niż zaoszczędzenie dodatkowych kilkudziesięciu megabajtów miejsca na twardym dysku. Jeżeli ma się jakieś wątpliwości, można użyć lokalnego, prywatnego pliku skompilowanego typu „assembly".

Z drugiej strony, jeżeli tworzy się własne komponenty w celu udostępniania ich innym programistom, by mogli ich używać w swoich aplikacjach, lub tworzy się jakiś inny rodzaj plików skompilowanych typu „assembly", które mają być ogólnie dostępne w całym systemie, GAC może się okazać dokładnie tym, czego potrzebujemy. Aby przyjrzeć się GAC, należy przejść do katalogu *Windows* w Eksploratorze Windows, po czym skierować się do podkatalogu *Assembly*. Dzięki utworzonej w tym celu wtyczce programowej wewnątrz Eksploratora Windows można zobaczyć pliki skompilowane typu „assembly" wraz z ich nazwami, *kulturą* (ang. *culture*) oraz informacjami dotyczącymi wersji. Rysunek 7.10 prezentuje częściową listę tego, co zobaczy się na ekranie komputera, który ma zainstalowane zarówno środowisko .NET 1.0, jak i .NET 2.0.

UWAGA *Kultura jest terminem technicznym określającym grupę ustawień, która odnosi się do poszczególnego regionu lub języka. Na przykład może łączyć w grupy informacje takie jak konwencje formatowania daty i czasu, język, zasady sortowania i wyświetlania tekstu, i tym podobne. Przykładem kultury jest zh-TW, które reprezentuje język chiński (Tajwan). Obsługę .NET dla lokalizacji można znaleźć w przestrzeni nazw* `System.Globalization`.

GAC jest trochę mniej przerażający niż świat COM, ponieważ posiada mniejszą ilość komponentów. Więcej komponentów oraz obiektów sterujących jest upakowanych w pojedynczym pliku skompilowanym typu „assembly". Kolejną rzeczą jest to, że jedyne pliki skompilowane typu „assembly", które będą dostępne początkowo, będą częścią środowiska .NET Framework.



Rysunek 7.10. Globalny bufor pliku skompilowanego

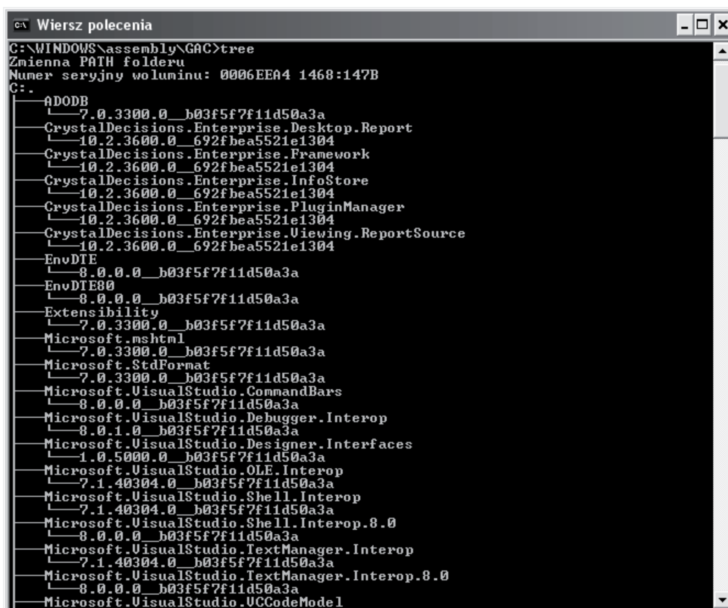
GAC „pod maską”

Wygląd pojedynczej listy plików w GAC jest trochę zwodniczy. Gdyby to, co się widzi, było naprawdę tym, co się dostaje, ta prosta technika powodowałaby dwa oczywiste problemy:

Kolidowanie nazw
: Co się stanie, jeżeli ktoś zainstaluje inny plik skompilowany typu „assembly” o identycznej nazwie jak już istniejący plik skompilowany typu „assembly”?

Problemy z wersjami
: Jedną z największych zalet .NET jest jego umiejętność przechowywania więcej niż jednej wersji danego pliku skompilowanego typu „assembly” w GAC, co powoduje, że każdy program używa dokładnie takiej wersji, dla jakiej został zaprojektowany. Ale jak jest to możliwe, skoro pliki skompilowane typu „assembly” używają takiej samej nazwy?

Rzeczywistość GAC jest trochę bardziej zaskakująca. To, co można zobaczyć w Eksploratorze Windows, jest produktem zmyślnej wtyczki programowej Eksploratora. Jeżeli użyje się do przeglądania katalogów narzędzia niskiego poziomu lub listingu wiersza poleceń, ujrzy się całkowicie inny obraz GAC (patrz rysunek 7.11).



Rysunek 7.11. Częściowy listing katalogów GAC

W odróżnieniu od prostej listy plików skompilowanych typu „assembly” GAC jest naprawdę złożoną strukturą katalogów. Struktura ta umożliwia istnienie wielu wersji każdego pliku skompilowanego typu „assembly” oraz używa spe-

cjalnych, unikalnych nazw, by upewnić się, że kolidowanie nazw jest niemożliwe. Faktyczny plik skompilowany typu „assembly" otrzymuje nazwę, którą widać we wtyczce programowej GAC, jednak jest przechowywany w specjalnym katalogu, używającym numeru wersji oraz niepowtarzalnego wygenerowanego ID (takiego jak *C:\[WinDir]\Assembly\GAC\System.Web\2.0.2411.0_b03f5f7f11d50a3a*, które mogłoby być użyte do przechowywania wersji pliku skompilowanego typu „assembly" *System.Web.dll*).

Tworzenie współdzielonego pliku skompilowanego typu „assembly"

Teraz, kiedy znasz już prawdę o GAC, nie zaskoczy Cię prawdopodobnie odkrycie, że nie można skopiować prywatnego pliku skompilowanego typu „assembly" bezpośrednio do GAC. (W Eksploratorze Windows polecenie *Wklej* (ang. *Paste*) jest nieaktywne). Zamiast tego trzeba utworzyć *współdzielony plik skompilowany typu assembly* (ang. *shared assembly*).

Zanim będzie można skopiować plik do GAC, należy utworzyć dla niego unikalną nazwę *strong name*. Te specjalne nazwy *strong name* to szczególna, nowa koncepcja, wprowadzona przez .NET w celu zapewnienia, że nigdy nie padnie się ofiarą piekła DLL. Myślą przewodnią jest to, że wszystkie pliki skompilowane typu „assembly", które się tworzy, są oznaczane za pomocą specjalnego klucza, który posiada tylko dany użytkownik. Taki system sprawia, że niemożliwe dla kogokolwiek innego staje się utworzenie pliku skompilowanego typu „assembly", który udaje nową wersję Twojego komponentu. Taka nazwa-klucz *strong name*, której używa się dla współdzielonego pliku skompilowanego typu „assembly", zawiera także ID posiadające gwarancję, że jest statystycznie niepowtarzalne (tak jak GUID).

Aby nieco skrócić tę długą historię, powiedzmy konkretnie: musisz wykonać cztery kroki.

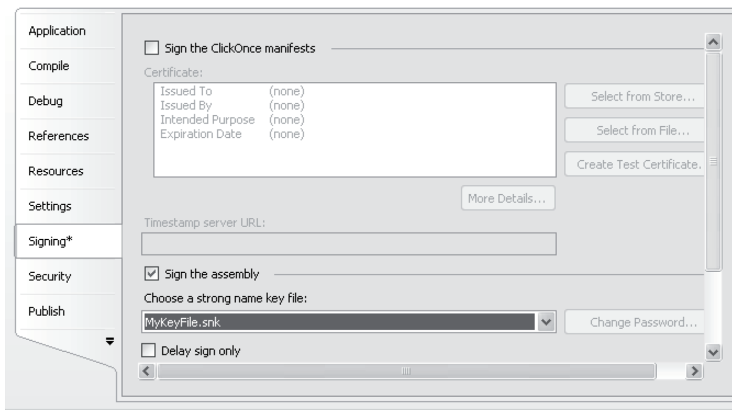
1. Utworzyć klucz.
2. Dodać klucz do danego pliku skompilowanego typu „assembly".
3. Skompilować dany plik „assembly".
4. Zainstalować ten plik skompilowany typu „assembly" w GAC.

We wcześniejszych wersjach .NET trzeba było zastosować narzędzia z wiersza poleceń, by wykonać to zadanie. W Visual Basic 2005 jednakże można wykonać dwa pierwsze kroki wewnątrz środowiska projektowego.

Tworzenie klucza

Pierwszym krokiem jest dwukrotne kliknięcie węzła *My Projects* w wyszukiwarce rozwiązań Solution Explorer. Następnie należy kliknąć zakładkę *Signing* (oznaczanie). Zakładka ta jest trochę odstraszająca, ponieważ jej opcje obejmują zarówno nadawanie tych unikalnych nazw, które interesują nas w danym momencie, jak i możliwość oznakowania dla *ClickOnce* (który omawiany jest w rozdziale 14.). W chwili obecnej zajmijmy się nadawaniem unikalnych nazw i zaznaczmy pole

wyboru *Sign The Assembly* (Oznacz plik skompilowany typu „assembly"). Następnie na liście *Choose A Strong Name Key File* (Wybierz plik klucza dla unikalnej nazwy), należy wybrać opcję *New*, podać nazwę pliku oraz (opcjonalnie) hasło (ang. *password*), po czym kliknąć *OK* (patrz rysunek 7.12). W wyszukiwarce rozwiązań Solution Explorer pojawi się plik klucza.



Rysunek 7.12. Ustawianie pliku klucza

WSKAZÓWKA *Zasadniczo .NET dostarcza dwóch rodzajów plików kluczy: zwykłych plików kluczy, które posiadają rozszerzenie .snk, oraz chronionych hasłem plików kluczy, posiadających rozszerzenie .pfx. Jak już zapewne domyśliłeś się, Visual Studio decyduje, który z tych kluczy utworzyć, w oparciu o to, czy podało się hasło podczas tworzenia pliku klucza. Chociaż taka ochrona nie jest niezłomna, pliki .pfx zapewniają dodatkową warstwę ochronną, ponieważ inni programiści nie będą mogli ich używać, by oznaczać aplikacje, jeżeli nie podadzą hasła.*

Każdy z plików klucza zawiera kombinację prywatnego i publicznego klucza. Klucz publiczny jest zwyczajowo dostępny dla świata zewnętrznego. Klucz prywatny jest bacznie strzeżony i nigdy nie powinien być wyjawiany więcej niż kilku wybranym osobom z określonej organizacji. Prywatne oraz publiczne klucze zapewniają specjalnego typu kodowanie. Coś zakodowane kluczem prywatnym może być odczytywane jedynie za pomocą odpowiadającego mu klucza publicznego. Każda rzecz zakodowana za pomocą klucza publicznego może być odczytywana jedynie przy użyciu odpowiadającego mu klucza prywatnego. Jest to uświęcony tradycją system kodowania, używany w e-mailach oraz innych aplikacjach internetowych.

W przypadku .NET prywatny klucz używany jest podczas tworzenia pliku skompilowanego typu „assembly", a klucz publiczny jest przechowywany w manifeście tego pliku. Kiedy ktoś uruchamia plik skompilowany typu „assembly" z GAC, wspólne środowisko uruchomieniowe (CLR) używa klucza publicznego, by odkodować informacje z manifestu. Jeżeli w celu oznaczenia użyty został inny klucz, operacja się nie powiedzie. Innymi słowy, dzięki użyciu pary kluczy

można uzyskać pewność, że jedynie osoba posiadająca dostęp do pliku klucza może utworzyć plik skompilowany typu „assembly” z użyciem Twojego identyfikatora. Z drugiej strony, każdy może uruchomić właściwie oznaczony plik dzięki zawartemu w nim kluczowi publicznemu.

WSKAZÓWKA *Kiedy posiadamy już plik klucza, możemy ponownie go używać w dowolnej ilości projektów. Można nawet utworzyć plik klucza poza Visual Studio, poprzez użycie narzędzia sn.exe, które dostarczane jest wraz z .NET. Magicznym wierszem polecenia służącym do tworzenia nowego pliku klucza za pomocą sn.exe jest wiersz* `sn –k KeyFileName.snk`.

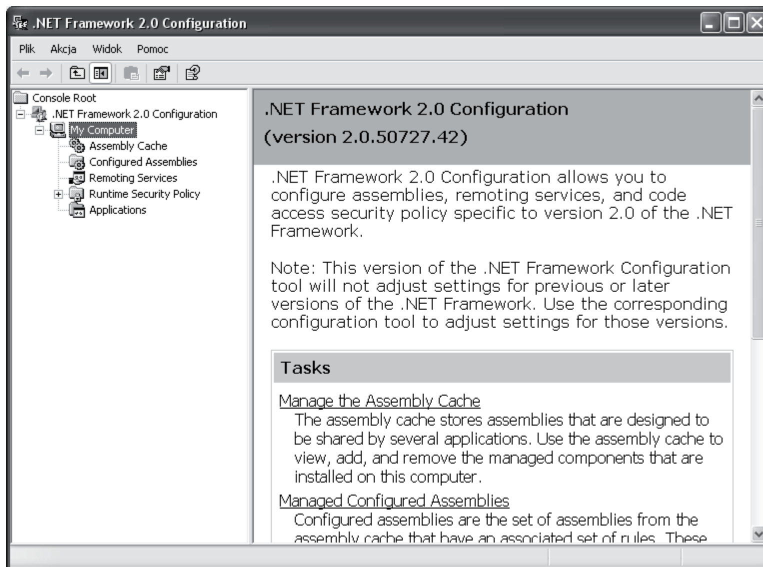
Instalowanie pliku assembly w GAC

Teraz posiadamy już różnorakie opcje dotyczące instalacji pliku skompilowanego typu „assembly” w GAC. Można wykorzystać w tym celu specjalnie zaprojektowany program instalacyjny (patrz rozdział 14.) lub narzędzie *GACUtil.exe*, lub ewentualnie przeciągnąć i upuścić plik skompilowany typu „assembly” za pomocą wtyczki programowej do GAC w Eksploratorze Windows. Bez względu na to, jaką metodę się wybierze, struktura katalogu tworzona jest automatycznie, plik skompilowany typu „assembly” jest kopiowany, a komponent pojawia się na liście GAC.

Pliki strategii

Jedną z najbardziej fascynujących cech plików skompilowanych typu „assembly” z unikalnymi nazwami jest to, że można konfigurować ich ustawienia dotyczące wersji. Dokonuje się tego poprzez utworzenie pliku o tej samej nazwie, co odpowiadający mu plik *.dll* lub *.exe*, oraz dodanie do niego rozszerzenia *.config*. Taki plik będzie automatycznie badany przez wspólne środowisko uruchomieniowe Common Language Runtime w poszukiwaniu dodatkowych informacji dotyczących tego, gdzie szukać plików skompilowanych typu „assembly” oraz które z ich wersji są dopuszczalne.

Informacja zawarta w pliku *.config* jest przechowywana w nadającym się do czytania formacie XML. Jednak nie będziemy teraz omawiali formatu pliku *.config*. Jeżeli jesteś ciekaw, jak on wygląda, możesz przeszukać plik pomocy Visual Studio. Godnym polecenia jest użycie wygodnego modułu dodatkowego do zarządzania, zapewnionego przez Microsoft, a zaprezentowanego na rysunku 7.13. W celu uruchomienia narzędzia *.NET Framework Configuration* (Konfiguracja środowiska .NET Framework) należy wybrać z menu *Start*: *Settings/Control Panel/Administrative Tools/Microsoft .NET Framework 2.0 Configuration* (Ustawienia/Panel sterowania/Narzędzia administracyjne/Konfiguracja środowiska Microsoft .NET Framework 2.0).



Rysunek 7.13. Narzędzie .NET Framework Configuration

Narzędzie do konfiguracji wypełnione jest zawrotną ilością opcji służących do ustawiania wszystkiego, poczynając od bezpieczeństwa dla całego obszaru komputera, a kończąc na podglądzie zależności plików skompilowanych typu „assembly”. Teraz zajmiemy się rozważaniami, w jaki sposób używać narzędzia konfiguracyjnego w jednym, konkretnym celu: do ustawiania strategii numeracji wersji.

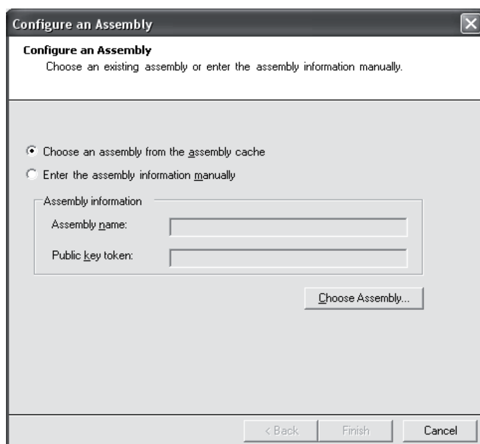
Aby zrozumieć, dlaczego jest to niezbędne, rozważmy następujący scenariusz. Zainstalowaliśmy nowszy plik skompilowany typu „assembly” w GAC, który jest nadal wstecznie kompatybilny i poprawia wydajność. Ponieważ jednak używa on nowego numeru wersji, nie będzie używany przez istniejące aplikacje. Rozwiązanie? Wchodząc we wszystkie aplikacje po kolei, można zawrzeć w programach instrukcje umożliwiające im używanie nowej wersji.

Tworzenie strategii wersji

Oto sposób na zmianę strategii wersji dla pliku skompilowanego typu „assembly”. Na początek należy przejść do znajdującego się na drzewie węzła *Configured Assemblies* (skonfigurowane pliki skompilowane typu „assembly”) oraz kliknąć opcję *Configure An Assembly* (skonfiguruj plik skompilowany typu „assembly”), która wywołuje okno zaprezentowane na rysunku 7.14.

Teraz należy kliknąć przycisk *Choose Assembly* (wybierz plik skompilowany typu „assembly”) oraz wybrać odpowiedni plik z GAC. Po tym, jak potwierdzi się dokonanie wyboru, wyświetlone zostanie okno z zakładkami o nazwie *Properties* (własności), które umożliwia użytkownikowi zmianę opcji pliku skompilowanego typu „assembly”:

- Zakładka *General* (podstawowe) dostarcza podstawowych informacji dotyczących pliku skompilowanego typu „assembly”.



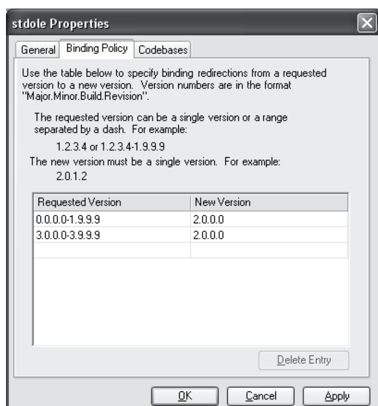
Rysunek 7.14. Konfiguracja pliku skompilowanego typu „assembly”

- Zakładka *Binding Policy* (polityka powiązań) umożliwia użytkownikowi konfigurację strategii numeracji wersji, na czym skoncentrujemy się w tym przykładzie.
- Zakładka *Codebases* (bazy kodu) umożliwia określenie ścieżki automatycznego pobierania i aktualizowania plików skompilowanych typu „assembly”.

Aby ustawić nową strategię wersji, należy dodać wpisy do zakładki *Binding Policy*. Każdy wpis łączy wymaganą wersję (lub zestaw wymaganych wersji) z nowym numerem wersji. Wymagana wersja to ta, do której próbuje uzyskać dostęp aplikacja klienta. Nową wersją jest plik skompilowany typu „assembly”, którego wspólne środowisko uruchomieniowe (CLR) decyduje się używać zamiast wersji wymaganej. Zasadniczo .NET przesyła zapotrzebowanie na plik skompilowany typu „assembly” do innej wersji tego samego pliku.

W poniższej konfiguracji przykładowej (rysunek 7.15) komponent będzie używał wersji 2.0.0.0 w momencie, gdy otrzyma informację o zapotrzebowaniu na wersję z zakresu od 0.0.0.0 do 1.9.9.9. Wiadomo, że taka wersja jest wstecznie kompatybilna i zapewnia najlepszą wydajność. Podobna specyfikacja dokonywana jest dla wersji z zakresu od 3.0.0.0 do 3.9.9.9. Jednakże żądanie jakiejkolwiek innej wersji (takiej jak na przykład 2.5.0.2) nie będzie przekazane, a używana będzie wymagana wersja pliku skompilowanego typu „assembly”, jeśli taka istnieje.

UWAGA *Strategia powiązań odnosi się jedynie do plików skompilowanych typu „assembly” z unikalnymi nazwami i ma sens jedynie wtedy, gdy ma się do czynienia ze współdzielonymi plikami skompilowanymi typu „assembly” w GAC (gdzie może koegzystować równocześnie więcej niż jedna wersja takiego pliku). Kiedy tworzy się zwykły, prywatny plik skompilowany typu „assembly”, nieposiadający unikalnej nazwy, aplikacja klienta będzie po prostu używać dowolnej wersji komponentu, która będzie obecna w aktualnym katalogu.*



Rysunek 7.15. Tworzenie strategii wersji

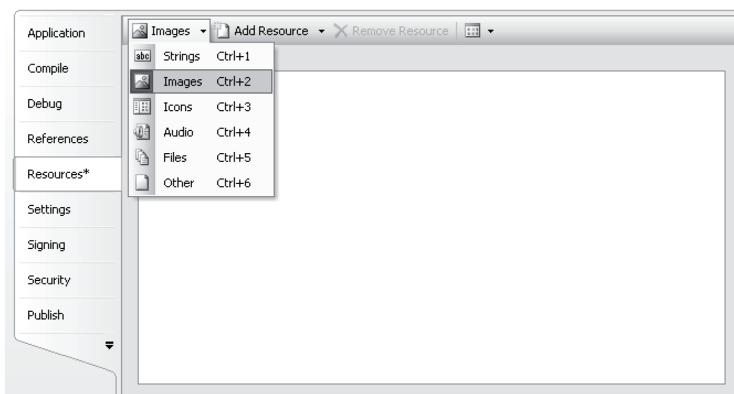
Zasoby

Wiele aplikacji posiada powiązane ze sobą zasoby — składniki takie jak pliki dźwiękowe, mapy bitowe oraz pokaźne bloki tekstu dotyczącego pomocy (które niekiedy tłumaczone są na wiele różnych języków). Można by przechowywać wszystkie te informacje w oddzielnych plikach, jednak skomplikowałoby to rozpowszechnianie Twojej aplikacji oraz otworzyło drzwi dla wszelkich rodzajów podstępnych problemów, takich jak dziwne błędy pojawiające się, gdy pliki te zaginą gdzieś lub zostaną zmodyfikowane poza danym programem.

.NET zawsze posiadało rozwiązanie dla tego typu problemów, jednakże dopiero w Visual Basic 2005 otrzymało ono wygodny sposób rozwiązywania problemów efektywnie podczas projektowania. Podstawowym zamysłem jest tutaj, że zbierze się te oddzielne pliki i zamieści się je jako dane binarne bezpośrednio wewnątrz skompilowanego pliku „assembly” (plik *.dll* lub *.exe*). Spowoduje to, że będą bezpiecznie ukryte przed wścibskimi spojrzeniami oraz manipulacją. Kiedy program użytkownika potrzebuje zasobu, może wyciągnąć go z pliku skompilowanego typu „assembly” i przekazać do programu w postaci ciągu bajtów (lub jakiegoś bardziej dogodnego typu danych).

Dodawanie zasobów

W celu wypróbowania, jak to działa, należy utworzyć nową aplikację Windows. Następnie dwukrotnie klikamy węzeł *My Project* w wyszukiwarce rozwiązań Solution Explorer i wybieramy zakładkę *Resources* (zasoby). Zakładka *Resources* jest centralnym miejscem, w którym można dodawać zasoby, usuwać je oraz zarządzać wszystkimi swoimi zasobami. Zasoby są pogrupowane w oddzielnych kategoriach, opartych na typie zawartości — na przykład istnieje oddzielna sekcja dla łańcuchów tekstowych (które można ręcznie edytować), obrazów, ikon, zawartości audio oraz wszelkich innych typów plików. Typ zasobów, który chce się zobaczyć, wybierany jest z rozwijanej listy *Category* (patrz rysunek 7.16).

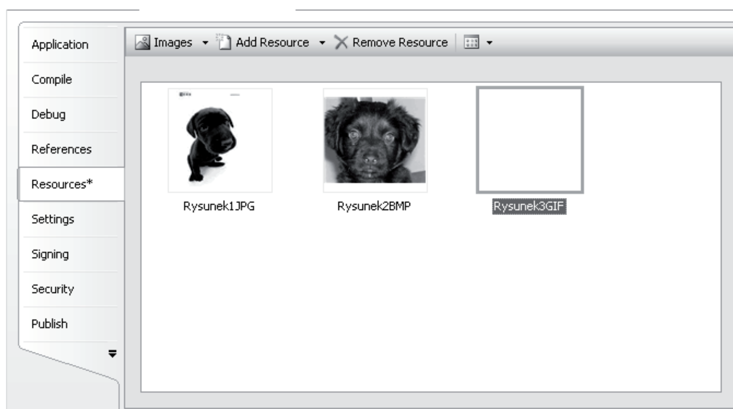


Rysunek 7.16. Przeglądanie typów zasobów

W tym przypadku jesteśmy zainteresowani obrazami, tak więc wybieramy kategorię *Images* (obrazy). Każda z kategorii pokazywana jest w trochę inny sposób — kategoria *Images* prezentuje miniatury wszystkich obrazów. Oczywiście w nowej aplikacji pracę zaczyna się, nie mając żadnych plików z obrazami.

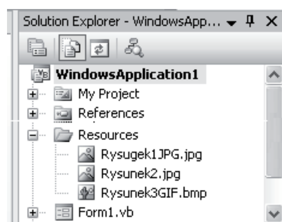
Aby dodać nowy obraz, należy w przyborniku wybrać opcję *Add Resource/Add Existing File* (Dodaj zasoby/Dodaj istniejący plik). Teraz trzeba przejść do właściwego pliku z obrazem, wybrać go i kliknąć *OK*. Jeżeli nie posiada się łatwo dostępnego pliku z obrazem, można spróbować wykorzystać któryś ze znajdujących się w katalogu Windows.

Kiedy doda się nowe zasoby, Visual Studio nada im taką samą nazwę, jaką noszą ich bieżące pliki (minus rozszerzenie oraz z niewielkimi zmianami, jeżeli nazwa nie jest odpowiednia dla nazwy zmiennej VB). Można jednak zmienić nazwę jakiegoś zasobu po dodaniu go, klikając prawym klawiszem myszy jego etykietę i wybierając opcję *Rename* (zmień nazwę). Rysunek 7.17 prezentuje sytuację, w której dwa przykładowe obrazy zostały dodane jako zasoby.



Rysunek 7.17. Dodawanie zasobów z obrazami

Gdy dodasz zasoby, dzieje się coś interesującego. Visual Studio tworzy podkatalog *Resources* w katalogu Twojego projektu i kopiuje plik z zasobami do tego nowego podkatalogu. W oknie przeglądarki *Solution Explorer* możesz zobaczyć ten podkatalog wraz ze wszystkimi zasobami, które zawiera (patrz rysunek 7.18).



Rysunek 7.18. Pliki zasobów w naszym projekcie

Innymi słowy, pliki źródłowe zasobów są przechowywane w pobliżu, jako część naszego projektu. Gdy jednak kompilujemy naszą aplikację, wszystkie te zasoby zostają osadzone wewnątrz pliku skompilowanego *assembly*. Urok takiego modelu polega na łatwości aktualizacji zasobów. Wystarczy tylko zastąpić nowszą wersją odpowiedni plik w podkatalogu *Resources* i potem dokonać rekompilacji naszej aplikacji. Innymi słowy, możesz bezpiecznie używać zasobu w dziesiątkach różnych plików kodu i wciąż aktualizować dane zasoby poprzez wykonanie pojedynczej, prostej operacji „kopiuj-wstaw". To podejście znacznie lepsze, niż występujące w poprzednich środowiskach programistycznych. Dla przykładu: programiści posługujący się Visual Basic 6 często przechowywali zasoby aplikacji w odrębnych plikach, co powodowało, że ich aplikacje były wrażliwe i podatne na błędy powodowane ewentualnym brakiem takich zewnętrznych plików zasobów bądź naruszeniem ich integralności poprzez zewnętrzną (także nieautoryzowaną) interwencję.

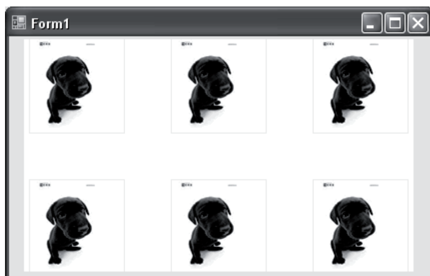
Używanie zasobów

Pozostał nam już tylko ostatni logiczny krok — zastosowanie zasobów w naszym projekcie. Skoro zasoby zostają podczas kompilacji osadzone wewnątrz pliku typu *assembly*, nie ma potrzeby ani sensu, byś wyciągał je z tego pliku. W istocie żaden plik nie musi nawet być rozpowszechniany wraz z Twoją aplikacją. Przeciwnie. Tu będzie Ci potrzebna i pomocna specjalnie dedykowana do tego celu klasa .NET o nazwie `ResourceManager`.

Nie, nie musisz używać klasy `ResourceManager` bezpośrednio. Visual Basic wyeksponuje wszystkie Twoje zasoby jako dedykowane własności (ang. *dedicated properties*) obiektu `My.Resources`. Dla przykładu: wyobraźmy sobie, że dodałeś zasób `Crayons` (dosł. ołówki) tak, jak pokazano to na rysunku 7.17. Możesz pobrać ten zasób jako obiekt klasy `Image` (obraz) z `My.Resources`, posługując się wygenerowaną automatycznie własnością `My.Resources.Crayons`. Oto przykład, jak możesz pobrać obraz i przypisać go do tła swojego formularza (ang. *form background*).

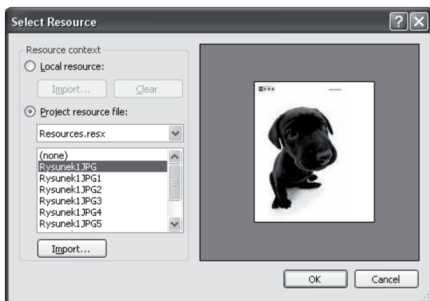
```
Private Sub Form1_Load (ByVal tender As System.Object, _
  ByVal e As System.EventArgs) Handles Sybase.Load
  Me.BackgroundImage = My.Resources.Crayons
  Me.BackgroundImageLayout = ImageLayout.Title
End Sub
```

Na rysunku 7.19 pokazano rezultat działania tego kodu.



Rysunek 7.19. Wyświetlanie obrazu z zasobów

W znakomitej większości przypadków nie musisz uciekać się do kodu, by posługiwać się zasobami, chyba że zamierzasz wykonać jakiś szczególny, indywidualny rysunek lub wykorzystać efekt dynamiczny (powiedzmy taki, że oto obraz pojawia się w ruchu programu, np. wtedy, gdy przesuniesz mysz ponad pewnym fragmentem formularza). Kreator Visual Studio daje Ci do dyspozycji szeroki zestaw możliwości dołączania zasobów graficznych do innych komponentów sterujących. Dla przykładu: gdy w trybie projektowania pobierzemy z listwy narzędziowej *Toolbox* i upuścimy na nasz formularz obiekt `PictureBox`, wystarczy kliknąć myszą znak wielokropka `(...)` w oknie własności *Properties* obok własności `Image`, by przywołać kreator (*designer*), którego okno pokazano na rysunku 7.20. W tym oknie możesz wybrać obraz spośród wszystkich zasobów w swoim projekcie.



Rysunek 7.20. Dodanie obrazu do zasobów aplikacji w trybie projektowania

UWAGA *Możesz posłużyć się opcją Local Resorce (zasoby lokalne) dostępną w górnej części tego okna, by dokonać importu obrazka i dodać ten obrazek jako zasoby do bieżącego formularza. To przypomina sposób zachowania się w podobnej sytuacji wcześniejszych wersji Visual Basic. Ma to z naszego punktu widzenia taką zaletę, że zaimportowany w ten sposób obraz może zostać dodany do pliku skompilowanego (assembly), ma jednak i poważną wadę. Mianowicie taki plik nie zostaje skopiowany do podkatalogu zasobów Resources, nie ma więc łatwego sposobu, by go później aktualizować. Nie będziesz również mógł w podobnie łatwy sposób wykorzystywać tego samego obrazu do wielu formularzy.*

Jeśli, przypadkowo, zżera Cię ciekawość, możesz zobaczyć, jak wygląda wygenerowany automatycznie kod dla klasy `My.Resources` (pamiętaj jednak, że zdecydowanie nie powinieneś próbować modyfikować tego kodu). Aby podejrzeć ten kod, wybierz polecenie *Project / Show All Files* (Pokaż wszystkie pliki projektu) i znajdź plik *Resources.Designer.vb* po rozwinięciu węzła *MyProject* w oknie przeglądarki *Solution Explorer*.

W środowisku .NET zasoby są stosowane również do innych zadań. Możesz, przykładowo, poeksperymentować z zasobami różnych typów, np. takimi jak pliki audio lub takie pliki, z których w swoim programie będziesz mógł wczytywać dane w postaci macierzy bajtowej (ang. *byte array*) lub w postaci strumienia obiektów (ang. *stream object*). Zasoby pozwalają nam również na *lokalizację* naszych formularzy. Lokalizacja to taki proces, w którym np. dla poszczególnych komponentów sterujących definiujemy tekst w różnych językach i odpowiednia wersja językowa zostaje zastosowana automatycznie, w zależności od ustawień lokalnych na danym komputerze, na którym zostaje uruchomiona aplikacja. Znacznie więcej informacji o lokalizacji aplikacji znajdziesz w systemie elektronicznej pomocy Visual Studio Help.

I co dalej?

W tym rozdziale przeanalizowaliśmy serce środowiska .NET, czyli mechanizm śledzenia wersji, wyjaśniając, jak dalece mechanizmy te różnią się zdecydowanie w .NET od znanych uprzednio z technologii COM. Zapoznałeś się także z tym, jak możesz osadzać zasoby (to takie przydatne nam bity spośród binarnych informacji) wprost we własnych plikach skompilowanych typu „assembly”. Dzięki temu Twoje komponenty już nigdy nie będą musiały szukać danych, które są im potrzebne.

Spora część tych różnych, a opisanych tu złożoności nie musi Cię dotyczyć. Dzięki planowaniu przez .NET rozpowszechnianie naszych aplikacji staje się często tak proste, jak zwyczajne skopiowanie katalogu. Jeśli jednak chcesz głębiej zrozumieć specyfikę środowiska .NET, możesz dalej samodzielnie poeksperymentować z takimi narzędziami jak *refleksje*, *ILDasm* czy *.NET Framework Configuration*. Z wszystkimi tymi narzędziami zapoznałeś się w tym rozdziale.

Najogólniej rzecz ujmując, pliki typu *assembly* w środowisku .NET tworzą swoiste systemowe fundamenty zupełnie nowej metody rozpowszechniania naszych aplikacji. Wiesz już teraz dostatecznie dużo, by aplikacja .NET mogła zadziałać na innym komputerze. W końcu, dopóki mamy zainstalowane środowisko .NET Framework, wszystko sprowadza się do zwykłego przenoszenia plików. W rozdziale 14. pójdziemy o krok dalej i dowiesz się, jak można utworzyć indywidualnie dostosowany program instalacyjny (ang. *customized setup*) ze znacznie większą liczbą bajerów.