Visual Basic .NET. Bazy danych. Księga eksperta

Autor: Jeffrey McManus
Tłumaczenie: Mikołaj Szczepaniak (rozdz. 8 – 12),
Grzegorz Werner (przedmowa, rozdz. 1 – 7)
ISBN: 83-7197-803-0
Tytuł oryginału: Database Access with Visual Basic.NET
Format: B5, stron: 400
Przykłady na ftp: 1422 kB

Zaprojektuj i napisz wydajne aplikacje bazodanowe, korzystając z Visual Basic .NET

- Opracuj schemat bazy danych, stwórz więzy integralności i zastosuj język SQL do manipulowania danymi
- Poznaj możliwości systemu MS SQL Server 2000 i wykorzystaj go w swoich aplikacjach
- Zastosuj technologię ADO.NET i Web Services w aplikacjach napisanych w Visual Basic .NET

Bazy danych to podstawa większości aplikacji biznesowych. Jednak sama baza danych to nie wszystko — należy zapewnić osobom korzystającym ze zgromadzonych w niej informacji wygodny sposób dostępu do nich. W tym celu pisane są narzędzia służące do manipulacji danymi i wyświetlania ich. Dzięki technologii .NET i możliwościom oferowanym przez język Visual Basic .NET stworzenie aplikacji korzystającej z zasobów zgromadzonych w bazie danych nie nastręcza problemów, jednak mimo to należy podejść do tego zadania w odpowiedni sposób. Schemat bazy danych, odpowiednia konstrukcja zapytań, właściwe zastosowanie obiektów komunikujących się z bazą — to elementy, które należy wziąć pod uwagę, przystępując do pracy nad aplikacją.

Książka „Visual Basic .NET. Bazy danych. Księga eksperta" to kompendium wiedzy dla programistów wykorzystujących Visual Basic .NET do tworzenia aplikacji opartych na bazach danych. Przedstawia zasady projektowania i pisania aplikacji WinForms, stron WebForms oraz usług Web Services w oparciu o bazę danych MS SQL Server 2000. Opisuje zasady stosowania technologii ADO.NET do połączenia aplikacji z tabelami w bazie i manipulowania zgromadzonymi w nich danymi.

- Schemat bazy danych, relacje, więzy integralności i normalizacja danych
- Konstruowanie zapytań w języku SQL
- Dostęp do bazy z poziomu aplikacji WinForms
- Zastosowanie bazy MS SQL Server 2000 w aplikacjach bazodanowych
- Podstawy technologii ADO.NET
- Stosowanie obiektów DataSet i DataAdapter
- Korzystanie z języka XML
- Tworzenie usług WebServices

Jeśli chcesz szybko i efektywnie tworzyć aplikacje bazodanowe w oparciu o technologię .NET, w tej książce znajdziesz wszystkie niezbędne do tego informacje.

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

Spis treści

Rozdział 4.
ADO.NET — dostawcy danych

Czasem można odnieść wrażenie, że kiedy programista budzi się rano, już czeka na niego nowy (i odmienny) przygotowany przez Microsoft model dostępu do baz danych. W niniejszym rozdziale skupimy się na jego najnowszym wcieleniu — ADO.NET. Najpierw wyjaśnimy, dlaczego powstał ten model dostępowy i zastanowimy się, czy jego wprowadzenie było uzasadnione. Następnie przedstawimy przegląd modelu ADO.NET i jego ogólnej architektury.

Celem tego rozdziału jest przygotowanie Czytelników do pracy z ADO.NET. Musimy zatem omówić podstawy jego działania i dokładniej przyjrzeć się najważniejszym obiektom dostawcy danych ADO.NET — `Connection`, `Command`, `Parameter` i `DataReader`. W rozdziałach 5., 6. i 7. zajmiemy się bardziej zaawansowanymi i interesującymi obiektami, które oferuje ADO.NET. Wszystkie one współdziałają z obiektem `DataSet` — centralnym elementem technologii ADO.NET.

Przegląd technologii ADO.NET

Ci, którzy już od pewnego czasu tworzą aplikacje bazodanowe za pomocą Visual Basica, przywykli do tego, że Microsoft co kilka lat wprowadza nowy, ulepszony model dostępu do danych. Oprócz kolejnego trzyliterowego skrótu pojawia się nowy interfejs API i model obiektowy, które trzeba poznać i opanować. W ostatnich latach programiści uczyli się kolejno modeli ODBC, DAO, RDO i ADO, a obecnie mają do czynienia z ADO.NET. Po wprowadzeniu każdej nowej technologii programista musi przestudiować jej cele i założenia projektowe, a następnie zadać sobie pytanie: czy ja i mój zespół powinniśmy przestawić się na tę technologię? W większości przypadków odpowiedź brzmi twierdząco, chyba że programista pracuje nad projektem, którego obecne i przyszłe wymagania nie mają związku z nowymi cechami i funkcjami technologii. Zdarza się to rzadko, choć trzeba przyznać, że dostępność technologii RDO (*Remote Data Object*) rzeczywiście nie miała wpływu na projekty wykorzystujące mechanizm JET (MDB) (technologia DAO jest ciągle lepszą metodą dostępu do baz danych MDB).

Motywacja i filozofia

Zadajmy więc zasadnicze pytanie: do czego potrzebny jest nowy model dostępu do danych? Można na nie odpowiedzieć sloganem reklamowym Toyoty z lat 70.: „Dostaliście, o co prosiliście". ADO.NET oferuje wiele rzeczy, których programiści domagali się od czasu wprowadzenia technologii ADO. To prawda, że z czasem dodano do ADO funkcje takie jak rozłączone zestawy rekordów oraz obsługa XML. Problem polega na tym, że funkcje te są właśnie dodatkami. Uzupełniają one pierwotny projekt, więc są niekompletne i niewygodne w użyciu.

Klasyczna (oparta na modelu COM) technologia ADO zawiera pojedynczy obiekt, `recordset`, którego używa się do najróżniejszych celów. W zależności od różnych parametrów konfiguracyjnych — takich jak typ kursora, lokalizacja kursora i typ blokowania — obiekt `recordset` działa inaczej i pełni różne funkcje.

W ADO.NET poszczególne funkcje zostały przypisane oddzielnym obiektom, których można używać niezależnie albo w połączeniu z innymi. Dzięki temu programiści mogą korzystać z obiektów zoptymalizowanych pod kątem konkretnej funkcji i niezawierających dodatkowego „bagażu". Pomimo to różne obiekty dobrze ze sobą współpracują, zapewniając większą funkcjonalność.

Obsługa aplikacji rozproszonych i rozłączonych

ADO.NET oferuje doskonałą i elastyczną obsługę aplikacji rozproszonych między wieloma komputerami (serwerami baz danych, serwerami aplikacji i klienckimi stacjami roboczymi). Szczególnie godna uwagi jest obsługa aplikacji rozłączonych (trój- lub wielowarstwowych), która minimalizuje współbieżne obciążenie i blokowanie zasobów w serwerze baz danych. Rezultatem jest większa skalowalność — możliwość obsługi większej liczby użytkowników jednocześnie poprzez stopniową rozbudowę sprzętu. Jest to istotne zwłaszcza w aplikacjach WWW.

Pełna obsługa języka XML

Choć klasyczna technologia ADO może zapisywać i odczytywać dane w formacie XML, to rzeczywisty format jest raczej niezwykły i niewygodnie się z nim pracuje. Ponadto obsługę XML dodano do ADO na stosunkowo późnym etapie, więc jest ona dość ograniczona i nieelastyczna. W przypadku ADO.NET obsługa XML od początku stanowiła zasadniczy element projektu. Filozofia ADO.NET głosi, że „dane to dane" — bez względu na źródło ich pochodzenia można przetwarzać je jako dane relacyjne albo hierarchiczne w zależności od potrzeb albo w zależności od używanego narzędzia programistycznego.

Co więcej, języka XML używa się jako formatu transmisji danych między różnymi warstwami i komputerami. Nie tylko eliminuje to problem przepuszczania wywołań COM przez zapory sieciowe, ale również ułatwia współdzielenie danych z aplikacjami, które działają na platformach innych niż Windows (ponieważ każdy system może przetwarzać tekstowe dane XML).

Integracja z .NET Framework

ADO.NET nie jest po prostu następną wersją ADO. Technologia ta została zaprojektowana i zaimplementowana jako element .NET Framework. Oznacza to, że działa jako kod zarządzany (ang. *Managed Code*), a wszystkie jej obiekty zachowują się zgodnie z tym, czego przywykliśmy oczekiwać od obiektów .NET. Jest także częścią standardowego pakietu .NET Framework, co pozwala uniknąć kłopotów z wersjami, z którymi programiści ADO musieli zmagać się w przeszłości.

Programowanie w stylu ADO

Choć technologię ADO.NET zaprojektowano i zaimplementowano jako część .NET Framework, wiele jej elementów powinno wyglądać znajomo dla programistów klasycznej technologii ADO. Nawet w przypadku funkcji nowych lub zrealizowanych w odmienny sposób doświadczony programista ADO będzie mógł wykorzystać nabytą wcześniej wiedzę, aby szybko zrozumieć i zastosować obiekty ADO.NET.

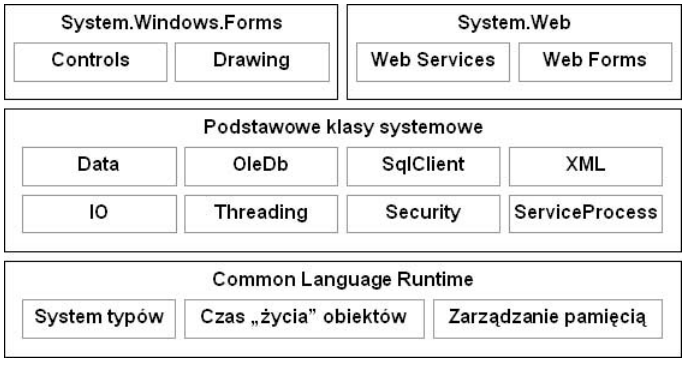
Porównanie ADO.NET z klasyczną technologią ADO (ADO 2.X)

Różnice między ADO.NET a klasyczną technologią ADO można streścić w następujący sposób:

- Klasyczna technologia ADO została zaprojektowana pod kątem dostępu z połączeniem i jest związana z fizycznym modelem danych. Technologia ADO.NET została zaprojektowana z myślą o dostępie bez połączenia i pozwala na logiczne modelowanie danych.
- W ADO.NET istnieje wyraźne rozróżnienie między modelem dostępu z połączeniem a modelem dostępu bez połączenia.
- W ADO.NET nie ma właściwości `CursorType`, `CursorLocation` ani `LockType`. Technologia ta używa tylko statycznych kursorów klienckich i blokowania optymistycznego.
- Zamiast używać pojedynczego, uniwersalnego obiektu, w ADO.NET funkcje klasycznego obiektu `recordset` podzielono na mniejsze, bardziej wyspecjalizowane obiekty, takie jak `DataReader`, `DataSet` i `DataTable`.
- ADO.NET umożliwia manipulowanie danymi XML, a nie tylko wykorzystywanie języka XML jako formatu wejścia-wyjścia.
- ADO.NET zapewnia ścisłą kontrolę typów w obiektach `DataSet` zamiast używania typu `Variant` do obsługi wszystkich pól. Ułatwia to wykrywanie błędów w czasie pisania kodu i zwiększa wydajność aplikacji.

Obiekty ADO.NET w ogólnej strukturze .NET Framework

Na rysunku 4.1 przedstawiono miejsce klas ADO.NET w ogólnej strukturze .NET Framework. Na dole znajduje się *Common Language Framework* (CLR), czyli infrastruktura



Rysunek 4.1. *Klasy ADO.NET w strukturze .NET Framework*

uruchomieniowa wszystkich aplikacji .NET (bez względu na język, w którym zostały napisane). Środowisko CLR zapewnia wspólny system plików, zarządzanie pamięcią i czasem „życia" obiektów.

Następna warstwa logiczna, która korzysta z usług CLR, to zbiór podstawowych klas systemowych. To właśnie te klasy zapewniają bogaty zestaw funkcji, z których mogą korzystać aplikacje .NET. Na rysunku 4.1 pokazano tylko niektóre klasy z biblioteki .NET Framework. W praktyce warstwa ta jest nowym interfejsem programistycznym (ang. *application programming interface*, API) systemu Windows. W przeszłości funkcje systemu Windows były dostępne za pośrednictwem interfejsu Windows API, który składał się z wielu niespójnych wywołań systemowych. W technologii .NET dostęp do tych (i nowych) funkcji odbywa się za pośrednictwem właściwości i metod eksponowanych przez podstawowe klasy systemowe. Dzięki takiemu podejściu pisanie aplikacji Windows jest bardziej spójne i komfortowe bez względu na to, czy są to aplikacje stacjonarne, internetowe czy usługi Web Services.

Warstwa ta zawiera kilka przestrzeni nazw (grup klas i innych definicji) związanych z dostępem do danych: `System.Data`, `System.Data.OleDB` oraz `System.Data.SqlClient`. W dalszej części niniejszego rozdziału — oraz w rozdziałach 5., 6. i 7. — przyjrzymy się bliżej wielu klasom i definicjom zawartym w tych przestrzeniach nazw.

Interfejsy aplikacyjne

Na najwyższym poziomie pojawia się zróżnicowanie między typami aplikacji, które mogą budować programiści. Istnieją klasy i kontrolki do budowania (klasycznych) aplikacji Windows opartych na formularzach (Windows Forms), inne klasy i kontrolki do budowania aplikacji internetowych opartych na przeglądarce (Web Forms), i kolejny zbiór klas do budowania usług Web Services. Wszystkie one korzystają jednak ze wspólnej biblioteki klas logiki aplikacyjnej — *podstawowych klas systemowych*.

Wiemy już mniej więcej, jakie miejsce zajmują klasy ADO.NET w ogólnej strukturze .NET Framework, więc przyjrzyjmy się bliżej obiektom ADO.NET.

Przegląd obiektów dostawczych .NET

Pomimo nacisku na model dostępu bez połączenia nadal konieczne jest łączenie się z fizyczną bazą danych w celu pobrania, zaktualizowania, wstawienia i (lub) usunięcia danych. Oprogramowanie, które łączy się i komunikuje z fizyczną bazą danych, w terminologii ADO.NET określa się mianem dostawcy danych .NET (ang *.NET Data Provider*). Dostawca danych .NET to odpowiednik dostawcy OLEDB albo sterownika ODBC. Dostawca danych składa się z kilku obiektów realizujących funkcje zdefiniowane przez klasy i interfejsy, z których się wywodzą.

Obecnie dostępnych jest trzech dostawców danych ADO.NET, każdy zdefiniowany we własnej przestrzeni nazw. W nazwach obiektów należących do tych przestrzeni nazw używa się przedrostków `OleDb`, `Sql` i `Odbc`. Kiedy piszemy o tych obiektach w sposób ogólny, używamy nazwy obiektu bez żadnego przedrostka.

SqlClient

Dostawca danych `SqlClient` jest zoptymalizowany pod kątem współpracy z SQL Serverem 7.0 i jego nowszymi wersjami. Osiąga większą wydajność, ponieważ (1) komunikuje się z bazą danych za pośrednictwem natywnego protokołu Tabular Data Stream (TDS), a nie protokołu OLEDB, który musi odwzorowywać interfejs OLEDB na protokół TDS; (2) eliminuje dodatkowe koszty związane z usługami zgodności COM; (3) nie oferuje funkcji, które nie są obsługiwane przez SQL Server. Obiekty tego dostawcy są zawarte w przestrzeni nazw `System.Data.SqlClient`.

OleDb

Dostawca danych `OleDb` uzyskuje dostęp do danych za pośrednictwem istniejącego natywnego dostawcy OLEDB (COM) oraz usług zgodności COM. Używa się go do obsługi baz danych innych niż SQL Server 7.0 i jego nowsze wersje. Zapewnia dostęp do każdej bazy danych, dla której napisano dostawcę OLEDB. Obiekty tego dostawcy są zawarte w przestrzeni nazw `System.Data.OleDb`.

Odbc

Dostawcy danych `Odbc` używa się do obsługi baz danych, które nie mają własnego dostawcy .NET, ani dostawcy OLEDB (COM). Ponadto w przypadku niektórych baz danych sterownik ODBC może zapewniać większą wydajność niż sterownik OLEDB, więc warto przeprowadzić testy, aby ustalić, który z nich lepiej odpowiada wymaganiom konkretnej aplikacji. Obiekty tego dostawcy są zawarte w przestrzeni nazw `Microsoft.Data.Odbc`.

Pisanie dostawcy danych ODBC opóźniło się nieco w stosunku do reszty .NET Framework i Visual Studia .NET. Nie został on więc dołączony do pierwotnego wydania Visual Studia .NET, ale można go pobrać z witryny WWW Microsoftu. Warto też sprawdzać, czy nie pojawili się nowi dostawcy danych .NET.

Obecnie w witrynie WWW dostępny jest również dostawca danych Oracle .NET. Dostawcy ODBC i Oracle są dołączeni do wersji 1.1 .NET Framework dostarczanej wraz z Visual Studiem .NET 2003. W rezultacie przestrzeń nazw dostawcy ODBC zmieni się z:

```
Microsoft.Data.Odbc
```

na:

```
System.Data.ODBC
```

W przykładach zamieszczonych w niniejszym rozdziale używana jest wersja 1.0 dostawcy ODBC. Ci, którzy korzystają już z wersji 1.1, powinni zmienić przestrzeń nazw w pokazany wyżej sposób.

Kluczowe obiekty

Każdy dostawca danych zawiera cztery kluczowe obiekty, które są wymienione w tabeli 4.1.

Tabela 4.1. *Kluczowe obiekty dostawców danych*

Obiekt	Krótki opis
`Connection`	Nawiązuje połączenie ze specyficznym źródłem danych
`Command`	Wykonuje polecenie w źródle danych; eksponuje kolekcję obiektów `Parameter` oraz metody do wykonywania różnych typów poleceń
`DataReader`	Zwraca jednokierunkowy, przeznaczony tylko do odczytu strumień danych ze źródła danych
`DataAdapter`	Tworzy pomost między obiektem `DataSet` a źródłem danych, aby umożliwić pobieranie i zapisywanie danych

Każdy obiekt wywodzi się z uniwersalnej klasy bazowej i implementuje uniwersalne interfejsy, ale zapewnia własną, specyficzną implementację. Na przykład obiekty `SqlData➥Adapter`, `OleDbDataAdapter` i `OdbcDataAdapter` wywodzą się z klasy `DbDataAdapter` i implementują te same interfejsy. Jednakże każdy z nich implementuje je inaczej, na użytek odpowiedniego źródła danych.

Przestrzeń nazw `System.Data.OleDb` zawiera następujące obiekty:

- `OleDbConnection`,
- `OleDbCommand`,
- `OleDbDataReader`,
- `OleDbDataAdapter`.

Przestrzeń nazw `System.Data.SqlClient` zawiera następujące obiekty:

- `SqlConnection`,
- `SqlCommand`,
- `SqlDataReader`,
- `SqlDataAdapter`.

Wreszcie przestrzeń nazw `Microsoft.Data.Odbc` zawiera następujące obiekty:

- `OdbcConnection`,
- `OdbcCommand`,
- `OdbcDataReader`,
- `OdbcDataAdapter`.

Wszyscy przyszli dostawcy danych będą mieć własne przestrzenie nazw i będą implementować wymagane obiekty w taki sam sposób.

Obiekt Connection

Obiekt ADO.NET `Connection` przypomina znany i lubiany obiekt `Connection` z klasycznej technologii ADO. Służy do nawiązywania połączenia z konkretnym źródłem danych z wykorzystaniem nazwy użytkownika i hasła określonych przez łańcuch połączeniowy (ang. *connection string*). Można dostosować połączenie do własnych potrzeb, określając odpowiednie parametry i wartości w łańcuchu połączeniowym. Obiekt `Command` (albo `DataAdapter`) może następnie użyć tego połączenia, aby wykonać żądane operacje na źródle danych.

W przeciwieństwie do obiektu `Connection` z ADO 2.X, obiekt `Connection` z ADO.NET nie ma metod `Execute` ani `OpenSchema`. Polecenia SQL można wykonywać tylko za pomocą obiektów `Command` albo `DataAdapter`. Odpowiednikiem metody `OpenSchema` jest metoda `GetOleDbSchemaTable` obiektu `OleDbConnection`.

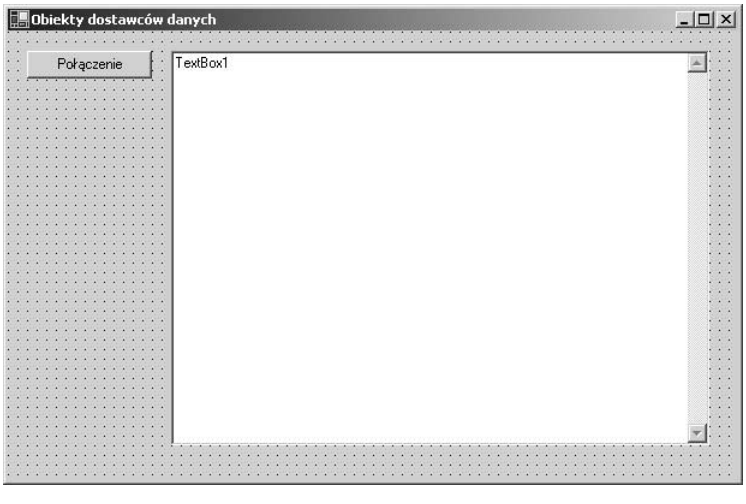
Choć pochodne obiekty `OleDbConnection`, `SqlConnection` i `OdbcConnection` implementują te same interfejsy, istnieją między nimi różnice. Odmienne są, na przykład, formaty łańcuchów połączeniowych. Format łańcucha `OleDbConnection` zaprojektowano tak, aby odpowiadał (z niewielkimi wyjątkami) standardowemu łańcuchowi połączeniowemu OLEDB. Format łańcucha `OdbcConnection` jest zbliżony do standardowego formatu łańcucha połączeniowego ODBC, choć nie identyczny. Format łańcucha `SqlConnection` różni się od dwóch pozostałych, ponieważ zawiera parametry dotyczące tylko SQL Servera 7.0 i jego nowszych wersji.

Co więcej, niektóre obiekty zawierają dodatkowe właściwości. Na przykład obiekt `OleDbConnection` ma właściwość `Provider`, za pomocą której określa się dostawcę OLEDB, a obiekt `OdbcConnection` ma właściwość `Driver`, za pomocą której określa się sterownik ODBC. Obiekt `SqlConnection` nie ma żadnej z tych właściwości, ponieważ jego źródło danych jest z góry określone (SQL Server). Ma natomiast właściwości `PacketSize` i `WorkstationID` specyficzne dla SQL Servera i nieobsługiwane przez dwa pozostałe typy połączeń.

Dość teorii; przejdźmy wreszcie do pisania kodu! Wszystkie kluczowe obiekty dostawców danych zostaną zaprezentowane na prostych, konkretnych przykładach. Zaczniemy od poniższego przykładowego programu i będziemy stopniowo go rozwijać.

1. Uruchomić Visual Studio .NET.
2. Utworzyć nowy projekt Visual Basica — *Windows Application*.
3. Nadać projektowi nazwę `ObiektyDostawcyDanych`.
4. Określić ścieżkę do folderu, w którym zostaną zapisane pliki projektu.
5. Powiększyć formularz `Form1`.
6. W oknie właściwości formularza `Form1` ustawić jego właściwość `Text` na `Obiekty dostawcy danych`.
7. Przeciągnąć przycisk z zakładki *Windows Forms* przybornika do górnego lewego rogu formularza.
8. W oknie właściwości przycisku ustawić właściwość `Name` na `cmdConnection`, a właściwość `Text` na `Połączenie`.
9. Przeciągnąć pole tekstowe z zakładki *Windows Forms* przybornika na prawą stronę formularza.
10. W oknie właściwości pola tekstowego ustawić właściwość `Name` na `txtResults`, właściwość `Multiline` na `True`, a właściwość `ScrollBars` na `Both`.
11. Powiększyć pole tekstowe tak, aby pokrywało mniej więcej 80 procent obszaru formularza.

Po wykonaniu powyższych czynności formularz powinien wyglądać tak jak na rysunku 4.2.



Rysunek 4.2. *Formularz Form1 w projekcie ObiektyDostawcowDanych*

Teraz należy przełączyć się do widoku kodu formularza i dodać poniższe wiersze na górze pliku. Powodują one zaimportowanie przestrzeni nazw, których będziemy używać podczas pisania przykładowych aplikacji w niniejszym rozdziale:

```
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.OleDb
Imports Microsoft.Data.Odbc
```

Zwróćmy uwagę na uniwersalne klasy i definicje ADO.NET oraz oddzielną przestrzeń nazw każdego dostawcy danych.

Edytor Visual Studia może nie rozpoznać przestrzeni nazw `Microsoft.Data.Odbc`, ponieważ w rzeczywistości jest to dodatek do podstawowego wydania produktu. W takim przypadku należy wykonać poniższe czynności:

1. Pobrać plik instalacyjny dostawcy danych `Odbc` z witryny WWW Microsoftu i zainstalować go w komputerze.
2. W oknie *Solution Explorer* kliknąć prawym przyciskiem myszy węzeł *References* projektu `ObiektyDostawcowDanych`.
3. Z podręcznego menu wybrać polecenie *Add Reference*.
4. Na karcie *.NET* okna dialogowego *Add Reference* przewinąć w dół listę komponentów i odszukać pozycję *Microsoft.Data.Odbc.dll*.
5. Kliknąć dwukrotnie pozycję *Microsoft.Data.Odbc.dll*, aby dodać ją do listy *Selected Components* na dole okna dialogowego.
6. Kliknąć przycisk *OK*, aby zamknąć okno dialogowe.

Jeśli z jakichś przyczyn nie zostanie rozpoznana któraś z pozostałych przestrzeni nazw, trzeba będzie dodać odwołanie do biblioteki *System.Data.dll*. W tym celu należy wykonać czynności 2 – 6, a w punkcie 4. zamienić nazwę *Microsoft.Data.Odbc.dll* na *System.Data.dll*[1].

Teraz w procedurze obsługi zdarzenia `btnConnection_Click` należy wpisać kod przedstawiony na listingu 4.1, który otwiera połączenie z bazą danych `pubs` w SQL Serverze. Kod ten wyświetla stan połączenia przed próbą nawiązania połączenia i po niej.

Listing 4.1. *Kod, który otwiera połączenie z bazą danych i wyświetla jego stan*

```
Private Sub btnConnection_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles _
  btnConnection.Click
    ' Tworzenieinstancji obiektu Connection
    Dim cnn As SqlConnection = New SqlConnection()

    ' Ustawianie łańcucha połączeniowego
    cnn.ConnectionString = "server=localhost;uid=sa;database=pubs"
    txtResults.Clear()

    ' Wyświetlanie stanu połączenia
    If (cnn.State = System.Data.ConnectionState.Open) Then
        txtResults.Text = txtResults.Text & "Połączenie jest otwarte"
    Else
        txtResults.Text = txtResults.Text & "Połączenie jest zamknięte"
    End If
    txtResults.Text = txtResults.Text & ControlChars.CrLf & ControlChars.CrLf

    ' Otwieranie połączenia
    txtResults.Text = txtResults.Text & "Otwieranie połączenia z bazą danych..." _
      & ControlChars.CrLf & ControlChars.CrLf
    cnn.Open()
```

[1] Użytkownicy Visual Studia .NET 2003 powinni zamienić przestrzeń nazw `Microsoft.Data.Odbc` na `System.Data.Odbc` — *przyp. tłum.*

```
    ' Wyświetlanie stanu połączenia
    If (cnn.State = System.Data.ConnectionState.Open) Then
        txtResults.Text = txtResults.Text & "Połączenie jest otwarte"
    Else
        txtResults.Text = txtResults.Text & "Połączenie jest zamknięte"
    End If
    txtResults.Text = txtResults.Text & ControlChars.CrLf
End Sub
```

Nową przydatną funkcją VB.NET jest możliwość automatycznego uzyskiwania tekstowej reprezentacji wartości wyliczeniowej (`enum`) zamiast pisania procedury, która wykonuje instrukcję `select-case` na wszystkich możliwych wartościach wyliczenia. Typy wyliczane są obiektami i dziedziczą metodę `ToString`, która zwraca łańcuch odpowiadający bieżącej wartości wyliczenia.

Na listingu 4.1 można zastąpić jednym wierszem instrukcje `If-Else`, które wyświetlają stan połączenia. Zamiast:

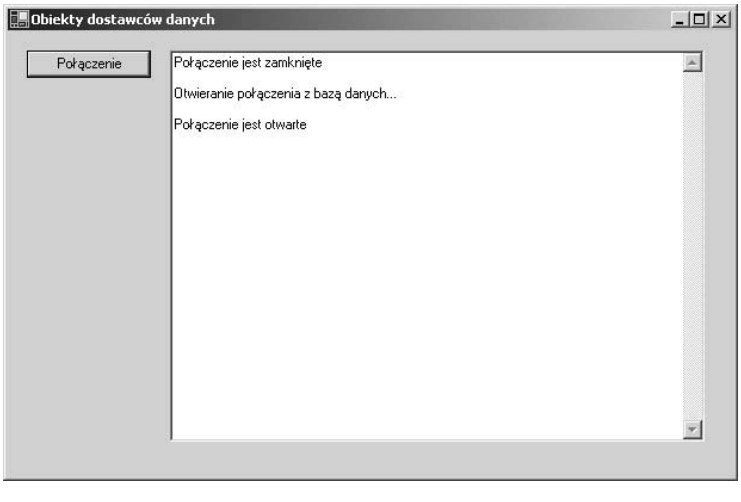
```
' Wyświetlanie stanu połączenia
If (cnn.State = System.Data.ConnectionState.Open) Then
    txtResults.Text = txtResults.Text & "Połączenie jest otwarte"
Else
    txtResults.Text = txtResults.Text & "Połączenie jest zamknięte"
End If
```

można napisać:

```
' Wyświetlanie stanu połączenia
txtResults.Text = txtResults.Text & "Połączenie jest w stanie " & _
   cnn.State.ToString & ControlChars.CrLf
```

Po uruchomieniu projektu `ObiektyDostawcowDanych` i kliknięciu przycisku *Połączenie* w polu tekstowym powinny ukazać się komunikaty, że połączenie jest zamknięte, otwierane, a wreszcie otwarte (zobacz rysunek 4.3).

Rysunek 4.3. *Wyniki otwierania połączenia za pomocą kodu z listingu 4.3.*



Podczas pisania kodu prawdziwych aplikacji trzeba wybrać i wdrożyć strategię obsługi błędów w poszczególnych procedurach i operacjach. Taka strategia zwykle opiera się na bloku `Try-Catch`. Zwykle nie dołączamy tego kodu do przykładów, ponieważ skupiamy się na programowaniu baz danych, a nie na ogólnych aspektach programowania w VB.NET.

Obiekt Command

Obiekt ADO.NET `Command` również powinien wydawać się znajomy doświadczonym programistom ADO 2.X. Podobnie jak obiekt ADO.NET `Connection`, przypomina on swojego poprzednika z ADO 2.X. Obiekt ten umożliwia wydawanie poleceń źródłu danych oraz pobieranie danych i (lub) wyników (jeśli polecenie zwraca jakieś wyniki).

Zgodnie z oczekiwaniami obiekt `Command` ma właściwości `CommandText` i `CommandType`, które definiują tekst i typ polecenia, właściwość `Connection`, która określa połączenie używane do wykonania polecenia, oraz właściwość `CommandTimeout`, która określa, jak długo polecenie może być wykonywane, zanim zostanie wygenerowany błąd. Ma również właściwość `Parameters`, która jest kolekcją parametrów przekazywanych wykonywanemu poleceniu. Wreszcie, w przeciwieństwie do klasycznego obiektu ADO `Command`, właściwość `Transaction` określa transakcję, w ramach której wykonywane jest polecenie.

Wszystkie trzy wersje obiektu `Command` (`OleDb`, `Sql` i `Odbc`) mają identyczne właściwości i metody z jednym wyjątkiem. Obiekt `SqlCommand` ma jedną dodatkową metodę — `ExecuteXmlReader`. Metoda ta wykorzystuje mechanizm SQL Servera, który automatycznie zwraca dane w formacie XML (kiedy do kwerendy SQL `SELECT` zostanie dodana klauzula `FOR XML`).

Inną różnicą między poszczególnymi wersjami obiektu `Command` są wartości właściwości `CommandType`. Wszystkie trzy wersje obsługują wartości `Text` i `StoredProcedure`, ale obiekt `OleDbCommand` obsługuje również trzecią wartość — `TableDirect`. Pozwala ona efektywnie wczytać zawartość całej tabeli poprzez ustawienie właściwości `CommandType` na `TableDirect`, a właściwości `CommandText` na nazwę tabeli.

Teraz należy rozbudować przygotowany wcześniej formularz (zobacz rysunek 4.1):

1. Dodać kolejny przycisk tuż pod przyciskiem `btnConnection` poprzez przeciągnięcie go z zakładki *Windows Forms* przybornika.
2. W oknie właściwości przycisku ustawić właściwość `Name` na `btnCommand`, a właściwość `Text` na `Polecenie`.
3. Wpisać kod obsługi przycisku `btnCommand` (zobacz listing 4.2).

Listing 4.2. *Kod, który otwiera połączenie z bazą danych i przygotowuje obiekt polecenia*

```
Private Sub btnCommand_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnCommand.Click

    ' Tworzenie instancji obiektu Connection
    Dim cnn As SqlConnection = New SqlConnection( _
      "server=localhost;uid=sa;database=pubs")
    ' Tworzenie instancji obiektów Command i Parameter
    Dim cmd As SqlCommand = New SqlCommand
    txtResults.Clear()
    ' Otwieranie połączenia
    cnn.Open()
    ' Ustawianie połączenia i tekstu polecenia w obiekcie Command
    cmd.Connection = cnn
    cmd.CommandType = CommandType.Text
```

```
    cmd.CommandText = "Select au_lname, state from authors"
    ' Wypisywanie łańcucha polecenia
    txtResults.Text = "Łańcuch polecenia:" & ControlChars.CrLf
    txtResults.Text = txtResults.Text & ControlChars.Tab & _
      cmd.CommandText() & ControlChars.CrLf
End Sub
```

Po uruchomieniu projektu ObiektyDostawcowDanych i kliknięciu przycisku Command w polu tekstowym powinna pojawić się instrukcja SQL, którą przypisano właściwości CommandText obiektu SqlCommand: Select au_lname, state from authors.

Wiele klas .NET Framework, a także klas pisanych przez innych programistów, zawiera przeciążone konstruktory obiektów. Innymi słowy, istnieje kilka różnych metod tworzenia nowych instancji klasy, za pomocą konstruktorów przyjmujących różną liczbę argumentów. Można wybrać wersję, która najlepiej odpowiada bieżącym potrzebom.

Konstruktor obiektu SqlConnection z listingu 4.2 różni się od konstruktora z listingu 4.1. W tamtym przykładzie użyliśmy konstruktora domyślnego, który nie przyjmuje żadnych argumentów. Następnie przypisaliśmy łańcuch połączeniowy obiektowi SqlConnection, ustawiając właściwość ConnectionString:

```
' Tworzenie instancji obiektu Connection
Dim cnn As SqlConnection = New SqlConnection()

' Ustawianie łańcucha połączeniowego
cnn.ConnectionString = "server=localhost;uid=sa;database=pubs"
```

Na listingu 4.2 użyliśmy konstruktora, który przyjmuje łańcuch połączeniowy jako argument. Dzięki temu mogliśmy utworzyć obiekt i przypisać mu łańcuch połączeniowy za pomocą jednego wiersza kodu:

```
' Tworzenie instancji obiektu Connection
Dim cnn As SqlConnection = New SqlConnection( _
  "server=localhost;uid=sa;database=pubs")
```

Używanie obiektu Command w połączeniu z parametrami i procedurami składowanymi

Podczas przesyłania kwerend lub poleceń do źródła danych często trzeba przekazywać wartości parametrów. Jest to niemal regułą podczas wykonywania poleceń akcji (UPDATE, INSERT lub DELETE) i wywoływania procedur składowanych. Aby spełnić ten wymóg, obiekt Command zawiera właściwość Parameters, która jest obiektem typu ParameterCollection i zawiera kolekcję obiektów Parameter. Mechanizm ten również jest bardzo podobny do swojego odpowiednika z ADO 2.X.

Obiekt Parametr (i ParameterCollection) jest ściśle związany z konkretnym dostawcą danych, więc jest jednym z obiektów, które muszą być zaimplementowane w każdym dostawcy danych ADO.NET. Programowanie obiektu SqlParameterCollection znacznie się różni od programowania obiektów OdbcParameterCollection i OleDbParameter ➥Collection. Pierwszy obsługuje parametry nazwane, a pozostałe dwa — parametry pozycyjne. Różnica ta wpływa na sposób definiowania kwerend i parametrów.

Zacznijmy od prostej kwerendy parametrycznej na tabeli `authors` w bazie danych `pubs`. Przypuśćmy, że chcemy pobierać nazwiska wszystkich autorów z określonego stanu. Gdybyśmy używali dostawcy danych `OleDb` lub `Odbc`, kwerenda wyglądałaby tak:

```
Select state, au_fname, au_lname from authors where state = ?
```

Miejscem na parametr jest tutaj znak zapytania. Gdyby kwerenda miała inne parametry, również zastąpiono by je znakami zapytania. Poszczególne parametry są odróżniane według pozycji; oznacza to, że kolejność dodawania parametrów do obiektu `Parameter` ➥`Collection` musi dokładnie odpowiadać ich kolejności w kwerendzie albo procedurze składowanej.

Gdybyśmy natomiast używali dostawcy danych `SqlClient`, kwerenda wyglądałaby tak:

```
Select state, au_fname, au_lname from authors where state = @MyParam
```

Miejscem na parametr jest tu nazwa specyficznego parametru; dodatkowe parametry również byłyby określone przez nazwy. Ponieważ parametry są odróżniane według nazw, można je dodawać do obiektu `ParameterCollection` w dowolnej kolejności.

Obiekt `Parameter` można utworzyć jawnie, za pomocą jego konstruktora (to znaczy za pomocą operatora `New`), albo poprzez przekazanie wymaganych parametrów do metody `Add` obiektu `ParameterCollection` (właściwości `Parameters` obiektu `Command`). Warto też zauważyć, że zarówno konstruktor `Parameter`, jak i metoda `Add` mają kilka przeciążonych wersji.

Oto sposób dodawania parametru polecenia poprzez jawne utworzenie obiektu parametru:

```
Dim myParameter As New OdbcParameter("@MyParam", OdbcType.Char, 2)
myParameter.Direction = ParameterDirection.Input
myParameter.Value = "CA"
cmd.Parameters.Add(myParameter)
```

A oto sposób dodawania parametru polecenia poprzez przekazanie odpowiednich argumentów do metody `Add`:

```
cmd.Parameters.Add("@MyParam", OdbcType.Char, 2)
cmd.Parameters(@MyParam).Direction = ParameterDirection.Input
cmd.Parameters(@MyParam).Value = "CA"
```

Drugi sposób jest krótszy i zwykle preferowany, chyba że obiekt `Parameter` będzie używany wielokrotnie.

W wywołaniu metody `Add` trzeba określić nazwę parametru, jego typ i (w razie potrzeby) długość. Następnie można ustawić „kierunek" parametru — wejściowy (`Input`), wyjściowy (`Output`), wejściowo-wyjściowy (`InputOutput`) albo wartość zwrotna (`ReturnValue`). Domyślny kierunek to `Input`. Wreszcie, aby określić wartość parametru, należy przypisać ją właściwości `Value` obiektu `Parameter`. Można ustawić kilka dodatkowych właściwości tego obiektu takich jak `Scale`, `Precision` i `IsNullable`.

Gdybyśmy używali dostawcy danych `SqlClient`, kod byłby niemal identyczny. Zastąpilibyśmy tylko przedrostki `Odbc` przedrostkami `Sql` i użyli typu wyliczeniowego `SqlDbType`:

```
Dim myParameter As New SqlParameter("@MyParam", SqlDbType.Char, 2)
myParameter.Direction = ParameterDirection.Input
myParameter.Value = "CA"
cmd.Parameters.Add(myParameter)
```

lub:

```
cmd.Parameters.Add("@MyParam", SqlDbType.Char, 2)
cmd.Parameters(@MyParam).Direction = ParameterDirection.Input
cmd.Parameters(@MyParam).Value = "CA"
```

Aby przypisać parametrowi wartość `Null`, należy użyć właściwości `Value` obiektu `DBNull`. Oto odpowiedni wiersz kodu:

```
cmd.Parameters("@MyParam").Value = DBNull.Value
```

Zmodyfikujmy kod przycisku `btnCommand` w sposób pokazany na listingu 4.3. Po uruchomieniu programu i kliknięciu przycisku *Polecenie* wyświetlony zostanie tekst kwerendy oraz nazwa i wartość parametru.

Listing 4.3. *Kod, który przygotowuje obiekty polecenia i parametru oraz wyświetla ich wartości*

```
Private Sub btnCommand_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnCommand.Click
    ' Tworzenie instancji obiektu Connection
    Dim cnn As SqlConnection = New SqlConnection( _
      "server=localhost;uid=sa;database=pubs")

    ' Tworzenie instancji obiektu Command i obiektu Parameter
    Dim cmd As SqlCommand = New SqlCommand
    Dim prm As SqlParameter = New SqlParameter
    txtResults.Clear()
    ' Otwieranie połączenia
    cnn.Open()
    ' Ustawianie połączenia i tekstu polecenia w obiekcie Command
    cmd.Connection = cnn
    cmd.CommandType = CommandType.Text
    cmd.CommandText = _
      "Select au_lname,state from authors where state = @MyParam"
    'Tworzenie parametru i ustawianie wartości
    cmd.Parameters.Add(New SqlParameter("@MyParam", _
      SqlDbType.Char, 2))
    cmd.Parameters("@MyParam").Value = "CA"
    ' Wypisywanie łańcucha polecenia
    txtResults.Text = "Łańcuch polecenia:" & ControlChars.CrLf
    txtResults.Text = txtResults.Text & ControlChars.Tab & _
      cmd.CommandText() & ControlChars.CrLf
    ' Wypisywanie parametrów polecenia i ich wartości
    txtResults.Text = txtResults.Text & "Parametry polecenia:" & _
      ControlChars.CrLf
    For Each prm In cmd.Parameters
        txtResults.Text = txtResults.Text & ControlChars.Tab & _
          prm.ParameterName & " = " & prm.Value & ControlChars.CrLf
    Next
End Sub
```

Procedury składowane wywołuje się w ten sam sposób z tym, że typem polecenia jest `CommandType.StoredProcedure`, a nie `CommandType.Text`. Nazwę procedury składowanej należy przypisać właściwości `CommandText`. Zatem wywołanie procedury składowanej o nazwie `GetAuthorsFromState`, która przyjmuje dwuznakowy parametr, wyglądałoby tak:

```
cmd.CommandType = CommandType.StoredProcedure
cmd.CommandText = "GetAuthorsFromState"
cmd.Parameters.Add("@MyParam", SqlDbType.Char, 2)
cmd.Parameters("@MyParam").Direction = ParameterDirection.Input
cmd.Parameters("@MyParam").Value = "CA"
```

W razie wywoływania procedury składowanej za pośrednictwem obiektu `OdbcCommand` należy używać standardowych sekwencji unikowych ODBC, zamiast po prostu podawać nazwę procedury składowanej we właściwości `CommandText`. W sekwencji unikowej znaki zapytania zastępują wartości parametrów. Odpowiednik ODBC powyższego kodu wyglądałby tak:

```
cmd.CommandType = CommandType.StoredProcedure
cmd.CommandText = "{GetAuthorsFromState ?}"
cmd.Parameters.Add("@MyParam", OdbcType.Char, 2)
cmd.Parameters("@MyParam").Direction = ParameterDirection.Input
cmd.Parameters("@MyParam").Value = "CA"
```

Jeśli procedura składowana ma wartość zwrotną, należy poprzedzić nazwę procedury znakami `? =`, jak w poniższym przykładzie:

```
cmd.CommandText = "{? = GetAuthorsFromState ?}"
```

Jeśli procedura składowana zwraca wyniki, należy określić kierunek `Output` i odczytać właściwość `Value` parametru po wywołaniu procedury. W poniższym przykładzie określono też, że procedura ma wartość zwrotną. Ponieważ użyto typu `Int` SQL Servera, nie trzeba określać długości parametru, gdyż z definicji składa się on z czterech bajtów:

```
cmd.Parameters.Add(New SqlParameter("result", SqlDbType.Int))
cmd.Parameters("result").Direction = ParameterDirection.ReturnValue
cmd.Parameters.Add(New SqlParameter("@MyParam", SqlDbType.Int))
cmd.Parameters("@MyParam").Direction = ParameterDirection.Output
' Tutaj należy wywołać procedurę składowaną
MsgBox(cmd.Parameters("@MyParam").Value)
```

Jeśli parametr jest wartością zwrotną (`ReturnValue`) procedury składowanej, powinien być dodawany jako pierwszy do kolekcji `Parameters`. Jest to wymagane w przypadku parametrów `OleDb` i `Odbc`, ponieważ — jak już wspomniano — mają one charakter pozycyjny, a wartość zwrotna jest przekazywana na pierwszej pozycji. Natomiast w przypadku parametrów `Sql` wartość zwrotna może natomiast zajmować dowolną pozycję, ponieważ są one rozróżniane według nazw.

Niebawem wyjaśnimy, jak wykonywać te polecenia, a wówczas przedstawimy dodatkowe przykłady użycia parametrów.

Wykonywanie poleceń

Dotychczas ustawialiśmy różne właściwości i parametry obiektu `Command`, ale jeszcze nie wykonaliśmy żadnego polecenia! Czas to zrobić. Istnieją trzy standardowe sposoby wykonywania poleceń zdefiniowanych w obiekcie `Command` oraz jeden sposób dodatkowy, dostępny tylko w przypadku obiektów `SqlCommand`:

- `ExecuteNonQuery` — wykonuje polecenie SQL, które nie zwraca żadnych rekordów,
- `ExecuteScalar` — wykonuje polecenie SQL i zwraca pierwszą kolumnę pierwszego wiersza,
- `ExecuteReader` — wykonuje polecenie SQL i zwraca wynikowy zestaw rekordów za pośrednictwem obiektu `DataReader`,
- `ExecuteXmlReader` — wykonuje polecenie SQL i zwraca wynikowy zestaw rekordów w formacie XML za pośrednictwem obiektu `XMLReader`.

Poniżej omówimy trzy pierwsze metody wykonywania poleceń. W rozdziale 10. zajmiemy się metodą `ExecuteXmlReader`, kiedy będziemy omawiać zastosowania języka XML w ADO.NET.

ExecuteNonQuery

Metoda `ExecuteNonQuery` to prawdopodobnie najłatwiejszy sposób wykonywania poleceń na źródle danych. Metoda ta umożliwia wykonywanie poleceń, które nie zwracają wyników (zestawów rekordów albo wartości skalarnych) poza wartością, która określa sukces lub niepowodzenie. Jest to również najbardziej wydajna metoda wykonywania poleceń. Można wykonywać te instrukcje SQL lub procedury składowane, które: (1) są poleceniami `CATALOG` lub Data Definition Language (DDL) służącymi do tworzenia lub modyfikowania struktur bazy danych takich jak tabele, widoki lub procedury składowane, (2) są poleceniami aktualizacyjnymi (`UPDATE`, `INSERT` lub `DELETE`) służącymi do modyfikowania danych.

Metoda `ExecuteNonQuery` zwraca pojedynczą wartość całkowitą. Znaczenie tej wartości zależy od typu wykonywanego polecenia.

Jeśli używane są polecenia `CATALOG` lub DDL, które modyfikują strukturę bazy danych, wartość zwrotna metody jest równa –1, jeśli operacja przebiegła pomyślnie. Jeśli używane są połączenia `UPDATE`, `INSERT` lub `DELETE`, które aktualizują dane, wartość zwrotna jest równa liczbie wierszy poddanych operacji. W obu przypadkach wartość zwrotna wynosi 0, jeśli operacja zakończy się niepowodzeniem.

Kontynuując projekt `ObiektyDostawcowDanych`, użyjemy obiektów z przestrzeni nazw `OleDb` i będziemy pracować z bazą danych `pubs`. Za pomocą odpowiedniego polecenia DDL utworzymy nową tabelę w tej bazie. Tabela będzie zawierać odwzorowania między kodami pocztowymi a stanami. Definicje pól tej tabeli będą dopasowane do używanych w bazie danych `pubs` (innych niż pola w bazie danych `Nowinki`). Tabela będzie zawierać dwa pola — jedno na kod pocztowy, a drugie na odpowiadający mu stan. Oto instrukcja SQL, która tworzy tę tabelę:

```
CREATE TABLE tblStateZipCode (
  ZipCode char (5) NOT NULL,
  State char (2) NOT NULL )
```

Formularz `Form1` należy zmodyfikować w następujący sposób:

1. Otworzyć formularz `Form1` w środowisku programistycznym Visual Studio.
2. W lewym górny rogu formularza dodać poprzez przeciągnięcie go z zakładki *Windows Forms* przybornika.

3. W oknie właściwości przycisku ustawić właściwość `Name` na `btnNonQuery`, a właściwość `Text` — na `Wykonanie`.

Następnie do procedury obsługi zdarzenia `Click` nowego przycisku należy dodać kod przedstawiony na listingu 4.4.

Listing 4.4. *Kod, który tworzy tabelę bazy danych za pomocą obiektów z przestrzeni nazw OleDb*

```
Private Sub btnNonQuery_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnNonQuery.Click
    ' Tworzenie instancji obiektu Connection
    Dim cnn As OleDbConnection = New OleDbConnection( _
      "provider=SQLOLEDB;server=localhost;uid=sa;database=pubs")
    Dim sql As String
    Dim result As Integer
   ' Tworzenie instancji obiektu Command
    Dim cmd As OleDbCommand = New OleDbCommand()
    ' Określanie połączenia i tekstu polecenia
    cmd.Connection = cnn
    cmd.CommandType = CommandType.Text
    ' Przypisywanie instrukcji SQL, która tworzy nową tabelę
    sql = "CREATE TABLE  tblStateZipCodes ( " & _
      "ZipCode char (5)  NOT NULL, " & _
      "State char (2)  NOT NULL )"
    cmd.CommandText = sql
    ' Otwieranie połączenia przed wywołaniem metody ExecuteNonQuery()
    cnn.Open()
    ' Kod trzeba umieścić w bloku Try-Catch, ponieważ nieudane
    ' wykonanie polecenia powoduje błąd czasu wykonania
    Try
        result = cmd.ExecuteNonQuery()
    Catch ex As Exception
        ' Wyświetlanie komunikatu o błędzie
        MessageBox.Show(ex.Message)
    End Try
    ' Wyświetlanie wyników polecenia
    If result = -1 Then
        MessageBox.Show("Polecenie zostało wykonane pomyślnie")
    Else
        MessageBox.Show("Wykonanie polecenia nie powiodło się")
    End If
    cnn.Close()
End Sub
```

Kiedy projekt `ObiektyDostawcowDanych` zostanie uruchomiony, a przycisk *Wykonanie* kliknięty po raz pierwszy, powinno pojawić się okno z informacją o pomyślnym wykonaniu polecenia. Aby sprawdzić, czy tabela została utworzona, można obejrzeć listę tabel w bazie danych `pubs` za pomocą narzędzia *Server Explorer* w Visual Studio (zobacz rozdział 1.) albo za pomocą programu SQL Server Enterprise Manager (zobacz rozdział 3.).

Jeśli przycisk *Wykonanie* zostanie kliknięty ponownie, pojawią się dwa okna z komunikatami. Pierwsze okno zawiera tekst pochodzący z wygenerowanego wyjątku; jest ono wyświetlane przez blok `Catch` i podaje przyczynę wystąpienia błędu. W tym przypadku polecenie zostało odrzucone, ponieważ w bazie danych znajduje się już tabela o takiej nazwie. Następnie zostanie wyświetlone drugie okno z komunikatem o niepowodzeniu podczas wykonywania polecenia.

W ten sam sposób można utworzyć widok albo procedurę składowaną. Aby utworzyć widok o nazwie EmployeeJobs_view, który zwraca listę stanowisk oraz nazwisk pracowników (posortowaną według stanowiska), należy zmienić instrukcję SQL z listingu 4.3 w następujący sposób:

```
sql = "CREATE VIEW EmployeeJobs_view AS " & _
  "SELECT TOP 100 PERCENT jobs.job_desc, " & _
  "employee.fname, employee.lname " & _
  "FROM jobs INNER JOIN " & _
  "employee ON jobs.job_id = employee.job_id " & _
  "ORDER BY jobs.job_desc "
```

Aby dołączyć do definicji widoku klauzulę ORDER BY w celu posortowania wyników, trzeba dodać klauzulę TOP w instrukcji SELECT.

Aby utworzyć procedurę składowaną, która przyjmuje jeden parametr i zwraca pojedynczą wartość, należy zmienić instrukcję SQL w sposób pokazany na listingu 4.5.

Listing 4.5. *Kod zawierający instrukcję SQL, która tworzy procedurę składowaną AuthorsInState1*

```
sql = "CREATE PROCEDURE AuthorsInState1 @State char(2)" & _
  " AS DECLARE @result int" & _
  " SELECT @result = COUNT (*) FROM authors " & _
  " WHERE state = @State" & _
  " RETURN (@result)"
```

Choć metoda ExecuteNonQuery zwraca tylko jedną wartość, jeśli zdefiniujemy parametry typu Output lub ReturnValue, zostaną one prawidłowo wypełnione danymi parametru. Jest to bardziej wydajne niż polecenie, które zwraca zbiór wyników albo wartość skalarną.

Rozważmy teraz drugi typ polecenia niekwerendowego — polecenie aktualizujące bazę danych, czyli instrukcję UPDATE, DELETE lub INSERT. Polecenia te zwykle wymagają określenia parametrów, zwłaszcza jeśli są wykonywane z wykorzystaniem procedur składowanych (a powinny ze względu na wydajność).

Przypuśćmy, że wydawca, który zaimplementował bazę danych pubs, jest w dobrym nastroju i postanowił zwiększyć tantiemy wypłacane autorom. Jeśli do formularza Form1 w projekcie ObiektyDostawcowDanych dodamy kolejny przycisk i pole tekstowe, główny księgowy wydawcy będzie mógł wprowadzić odsetek, o jaki zostaną zwiększone tantiemy, jako parametr polecenia UPDATE. W tym celu należy wykonać poniższe czynności:

1. Dodać kolejny przycisk tuż pod przyciskiem btnNonQuery.
2. W oknie właściwości przycisku ustawić właściwość Name na btnUpdate, a właściwość Text na Aktualizacja.
3. Pod nowym przyciskiem dodać pole tekstowe poprzez przeciągnięcie go z zakładki *Windows Forms* przybornika.
4. W oknie właściwości pola tekstowego ustawić właściwość Name na txtParam1, a właściwość Text na 0. Dzięki ustawieniu wartości pola tekstowego na 0 użytkownik, który uruchomi program i zapomni o ustawieniu tej wartości przed kliknięciem przycisku *Aktualizacja*, nie spowoduje uszkodzenia danych albo błędu czasu wykonania.

5. Dodać kod nowego przycisku (zobacz listing 4.6).

Listing 4.6. *Kod, który aktualizuje bazę danych, wykorzystując instrukcję SQL z parametrem*

```
Private Sub btnUpdate_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles btnUpdate.Click
    Dim result As Integer
    ' Tworzenie instancji obiektu Connection
    Dim cnn As SqlConnection = New SqlConnection( _
      "server=localhost;uid=sa;database=pubs")
    ' Tworzenie instancji obiektu Command
    Dim cmd As SqlCommand = New SqlCommand()
    txtResults.Clear()
    ' Określanie połączenia i tekstu polecenia
    cmd.Connection = cnn
    cmd.CommandType = CommandType.Text
    cmd.CommandText = "UPDATE roysched SET royalty = royalty + @param1"
    ' Tworzenie parametru i ustawianie jego wartości
    cmd.Parameters.Add(New SqlParameter("@param1", SqlDbType.Int))
    cmd.Parameters("@param1").Direction = ParameterDirection.Input
    cmd.Parameters("@param1").Value = Val(txtParam1.Text)
    ' Otwieranie połączenia przed wywołaniem funkcji ExecuteNonQuery()
    cnn.Open()
    result = cmd.ExecuteNonQuery()
    MessageBox.Show("Zaktualizowano " & result _
      & " rekordów", "Obiekty dostawców danych")
    cnn.Close()
End Sub
```

Aby zaktualizować tabelę tantiem, należy uruchomić projekt `ObiektyDostawcowDanych`, wpisać wartość całkowitą w polu tekstowym parametru i kliknąć przycisk *Aktualizuj*. Powinno pojawić się okno komunikatu z informacją o liczbie zaktualizowanych rekordów. Można zweryfikować ten wynik poprzez uruchomienie programu SQL Server Enterprise Manager i wyświetlenie danych z tabeli `roysched` przed wykonaniem tego przykładowego programu i po nim.

Tę samą aktualizację można wykonać za pomocą procedury składowanej. Zaletą procedur składowanych jest to, że są one bardziej wydajne i znajdują się w centralnej lokalizacji. Mają one jednak tę wadę, że w skład zespołu programistów musi wchodzić administrator baz danych, a przynajmniej ktoś, kto umie pisać procedury składowane. W dużych organizacjach może upłynąć sporo czasu, zanim uda się namówić administratora baz danych do zmodyfikowania procedury składowanej. Tym, którzy mogą zrobić to sami, nie zajmie to więcej niż minutę. Procedurę składowaną można dodać za pomocą programów SQL Server Enterprise Manager albo SQL Query Analyzer w sposób opisany w rozdziale 3. Można również użyć projektu `ObiektyDostawcowDanych` i zmienić instrukcję SQL, jak zrobiliśmy to w jednym z poprzednich przykładów.

Procedura składowana powinna wyglądać tak:

```
CREATE PROCEDURE UpdateRoyalties
@param int
AS
UPDATE roysched SET royalty = royalty + @param
```

Aby wywołać procedurę składowaną, na listingu 4.6 trzeba zmienić właściwości `Command➥Type` i `CommandText` obiektu `Command`. Odpowiednie wiersze kodu powinny wyglądać tak:

```
cmd.CommandType = CommandType.StoredProcedure
cmd.CommandText = "UpdateRoyalties"
```

Uruchomienie zmodyfikowanego programu powinno dać te same wyniki. Teraz aktualizacja jest wykonywana przez procedurę składowaną, a nie przez instrukcję SQL zapisaną w kodzie aplikacji.

ExecuteScalar

Czasem trzeba wykonać polecenie bazy danych, które zwraca wartość *skalarną* — to znaczy pojedynczą. Typowym przykładem takich poleceń są instrukcje SQL, które wykonują funkcję agregacyjną, takie jak `SUM` lub `COUNT`. Innym przykładem są tabele wyszukiwania, które zwracają pojedynczą wartość, a także polecenia, które zwracają wartość logiczną. Metoda `ExecuteScalar` wykonuje polecenie i zwraca pierwszą kolumnę pierwszego wiersza w zbiorze wyników. Pozostałe kolumny i wiersze są ignorowane.

Dodajmy poniższą procedurę składowaną do bazy danych `pubs`:

```
CREATE PROCEDURE AuthorsInState2
@param1 char(2)
AS
SELECT COUNT(*) FROM authors WHERE state = @param1
```

Procedura `AuthorsInState2` przyjmuje parametr w postaci dwuliterowego kodu nazwy stanu i zwraca liczbę autorów mieszkających w określonym stanie. Jest ona funkcjonalnie równoważna procedurze `AuthorsInState1` pokazanej na listingu 4.5, ale zwraca zbiór wyników zamiast pojedynczej wartości.

Kiedy używamy metody `ExecuteScalar` zamiast `ExecuteNonQuery` i przekazujemy wartość skalarną jako parametr typu `ReturnValue`, wydajność aplikacji nieco się zmniejsza. Po co więc używać metody `ExecuteScalar`? Jest ona prostsza i mniej pracochłonna, ponieważ nie trzeba definiować parametrów zarówno w poleceniu, jak i w wywołującym je kodzie.

Aby wywołać tę procedurę za pomocą obiektów dostawcy danych `Odbc`, należy wykonać poniższe czynności:

1. Dodać kolejny przycisk tuż pod polem tekstowym `txtParam1`.
2. W oknie właściwości przycisku ustawić właściwość `Name` na `btnExecuteScalar`, a właściwość `Text` na *Wartość skalarna*.
3. Dodać kod nowego przycisku (zobacz listing 4.7).

Listing 4.7. *Kod, który pobiera wartość skalarną z procedury składowanej, wykorzystując obiekty dostawcy danych Odbc*

```
Private Sub btnExecuteScalar_Click(ByVal sender As System.Object,
  ByVal e As System.EventArgs) Handles btnExecuteScalar.Click
    Dim result As Integer
```

```
    ' Tworzenie instancji obiektu Connection
    Dim cnn As OdbcConnection = New OdbcConnection( _
      "DRIVER={SQL Server};server=localhost;uid=sa;database=pubs")
    ' Tworzenie instancji obiektu Command
    Dim cmd As OdbcCommand = New OdbcCommand()
    txtResults.Clear()
    ' Określanie połączenia i tekstu polecenia
    cmd.Connection = cnn
    cmd.CommandType = CommandType.StoredProcedure
    cmd.CommandText = "{call AuthorsInState2(?) }"
    ' Tworzenie parametru i ustawianie jego wartości
    cmd.Parameters.Add("@param1", OdbcType.Char, 2)
    cmd.Parameters("@param1").Value = txtParam1.Text
    ' Otwieranie połączenia przed wywołaniem metody ExecuteScalar()
    cnn.Open()

    result = cmd.ExecuteScalar()
    MessageBox.Show("Liczba autorów: " & result, "Obiekty Dostawców danych")
    cnn.Close()
End Sub
```

Teraz należy uruchomić aplikację i wprowadzić dwuznakowy kod nazwy stanu w polu tekstowym parametru. Po kliknięciu przycisku *Wartość skalarna* powinno pojawić się okno komunikatu z informacją o liczbie autorów mieszkających w danym stanie. Można zweryfikować ten wynik za pomocą programu SQL Server Enterprise Manager poprzez wyświetlenie danych z tabeli `authors`.

Domyślne dane w tabeli `pubs` powinny dać wynik 2 w przypadku stanu UT i 15 w przypadku stanu CA.

ExecuteReader

Najlepsze (i najważniejsze) zostawiliśmy na koniec. Metodę `ExecuteReader` wywołuje się po to, aby wykonać polecenie, które zwraca zestaw wierszy (rekordów). W większości aplikacji bazodanowych właśnie ta metoda wykonywania poleceń jest używana najczęściej. Wykonuje ona polecenie, które zwraca wiersze danych za pośrednictwem obiektu `DataReader`. Wiersze skanuje się kolejno, zaczynając od pierwszego. W następnym podrozdziale podamy więcej informacji o obiekcie `DataReader` i podamy odpowiednie przykłady.