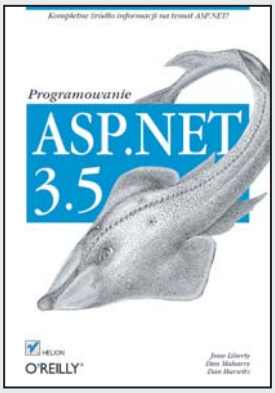
ASP.NET 3.5. Programowanie



Autorzy: Jesse Liberty, Dan Maharry, Dan Hurwitz
Tłumaczenie: Robert Górczyński
ISBN: 978-83-246-2212-2
Tytuł oryginału: Programming ASP.NET 3.5
Format: 168×237, stron: 1088

Kompletne źródło informacji na temat ASP.NET!

- Jak maksymalnie wykorzystać możliwości Visual Studio?
- Jakie tajemnice kryje język LINQ?
- Jak tworzyć bezpieczne aplikacje internetowe?

Aplikacje internetowe cieszą się wciąż rosnącą popularnością. Na rynku narzędzi do ich tworzenia można znaleźć wiele rozwiązań, a wśród nich jedno wyjątkowe – platformę .NET. Pozwala ona na wykorzystanie dowolnego obsługiwanego przez nią języka programowania do tworzenia dynamicznych, interaktywnych i atrakcyjnych rozwiązań internetowych. Wybierając platformę .NET, otrzymasz dostęp do wielu dodatkowych narzędzi i – co najważniejsze – do wiedzy zgromadzonej przez całą używającą jej społeczność. Niezliczona liczba stron, artykułów i osób chętnych do pomocy sprawia, że rozwiązanie nawet najbardziej skomplikowanego problemu staje się łatwiejsze.

Dzięki tej książce zdobędziesz wiedzę pozwalającą Ci na swobodne poruszanie się w świecie aplikacji internetowych opartych o .NET. Nauczysz się w maksymalny sposób wykorzystywać możliwości środowiska Visual Studio 2008, poznasz dostępne kontrolki oraz sprawdzisz, do czego może Ci się przydać ADO.NET. Ponadto odkryjesz tajemnice języka LINQ i zasady, których przestrzeganie zapewni bezpieczeństwo Twojej aplikacji. W kolejnych rozdziałach autorzy przedstawią Ci metody tworzenia usług sieciowych, zwiększania wydajności poprzez buforowanie oraz konfiguracji serwera IIS 7.0. Książka ta pozwoli Ci w łatwy sposób wykonać pierwszy krok w świat dynamicznych stron WWW, tworzonych z wykorzystaniem ASP.NET.

- Praca w zintegrowanym środowisku programistycznym Visual Studio 2008
- Podstawowe kontrolki oraz kontrolki pozwalające na dostęp do danych
- Dostęp do baz danych z wykorzystaniem ADO.NET
- Zastosowanie języka LINQ
- Gwarancja poprawności danych
- Zapewnienie bezpieczeństwa aplikacji internetowej
- Tworzenie stron wzorcowych
- Przygotowanie usług sieciowych
- Protokoły i standardy usług sieciowych
- Poprawa wydajności poprzez zastosowanie buforowania
- Konfiguracja serwera IIS 7.0
- Debugowanie kodu i śledzenie jego wykonania
- Wdrażanie aplikacji w środowisku lokalnym i globalnym
- Przydatne skróty klawiaturowe

Poznaj możliwości jednej z najpopularniejszych platform do tworzenia dynamicznych stron WWW!

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

Spis treści

ROZDZIAŁ 10.

Prezentacja LINQ

Jednym z głównych dodatków do wydania 3.5 platformy .NET jest LINQ (Language Integrated Query), czyli nowy *interfejs programowania aplikacji* (API), będący w zasadzie zbiorem przestrzeni nazw oraz klas służących jednemu celowi: pobieraniu danych z dowolnych źródeł.

Czytelnik może się zastanawiać, dlaczego firma Microsoft zdecydowała się na dostarczenie kolejnego sposobu pracy z obiektami źródeł danych, skoro technologia ADO.NET doskonale sprawdza się na tym polu. Czy wprowadzenie LINQ nie jest bezcelowe? Otóż nie. W rozdziale 7. pokazano, jak obiekty .NET `DataSource` zapewniają możliwość współpracy z danymi pochodzącymi z różnych źródeł — obiektów `Business`, pliku XML lub bazy danych. Ponadto w rozdziale 9. pokazano, że technologia ADO.NET oferuje znacznie dokładniejszą kontrolę nad dostępem do bazy danych. Cofnijmy się jednak o krok i zastanówmy nad sposobem codziennej pracy z danymi:

- Bardzo rzadko zdarza się tak, że wszystkie wymagane dane znajdują się w tym samym źródle. Niektóre mogą znajdować się w bazie danych, kolejne w obiektach `Business`, a jeszcze inne w punkcie końcowym usługi sieciowej itd.
- Łatwość, z jaką można uzyskać dostęp do danych, jest całkowicie uzależniona od miejsca ich przechowywania. Uzyskanie dostępu do obiektów umieszczonych w pamięci okazuje się znacznie łatwiejsze niż uzyskanie dostępu do danych przechowywanych w pliku XML bądź bazie danych.
- Same nieprzetworzone dane bardzo często nie stanowią produktu końcowego. Po ich zebraniu potrzeba sortowania, modyfikacji, grupowania, zmiany kolejności, zapętlenia, połączenia w pojedynczą pulę itd. Warto zatem spojrzeć na poniższy fragment kodu:

```
List<Book> books = GetBooks();
// Sortowanie.
books.SortByPrice(delegate(Book first, Book second))
{
   return ((double)(second.Price - first.Price));
}
// Zapętlenie oraz agregacja.
double totalIncome = 0;
books.ForEach(delegate(Book book))
{
   totalIncome += (book.Price * book.TotalSales);
}
```

- Powyżej w sześciu krótkich wierszach kodu przeprowadzono sortowanie, zapętlenie i agregację. Przyjęto jednak założenie, że cena nie jest pobierana poprzez odczyt oddzielnych źródeł, takich jak arkusz kalkulacyjny, usługa sieciowa lub plik XML.

Powstaje więc pytanie, dlaczego nie skorzystać z lepszego API służącego do pobierania danych. Takie API mogłoby oferować łatwy dostęp do wszystkich źródeł danych, a także możliwość łączenia danych pochodzących z wielu źródeł. Następnie na połączonych danych można przeprowadzać standardowe operacje, to wszystko w pojedynczym wierszu kodu. Przykładowo, pojedyncza operacja sprawdzałaby, czy wszystkie pola danych są ściśle określone, więc odpada konieczność rzutowania obiektu na właściwy rodzaj podczas pobierania obiektu z bazy danych. Inny przykład to ułatwienie programistom tworzenia dostawców dostępu do danych, które nie są jeszcze obsługiwane. Takie możliwości daje LINQ, którego kod jest podobny do poniższego:

```
var query = from book in Books
   where book.QuarterlySales > 0
   select book => {Name, (Price * QuarterlySales) as QuarterlyIncome}
   orderby QuarterlySales;
```

LINQ używa wielu nowych funkcji C# 3.0 w celu przedstawienia składni znanej z SQL, którą można zastosować na dowolnej liczbie odmiennych źródeł danych w celu wykonywania zapytań i przetwarzania otrzymanych danych. LINQ to API o naprawdę potężnych możliwościach.

W rozdziale zostanie omówione działanie LINQ, znajdzie się tu także wyjaśnienie, dlaczego działa tak dobrze. Będzie mowa również o sposobach integracji LINQ z tworzonymi stronami ASP.NET. W szczególności przyjrzymy się używaniu LINQ z bazą danych SQL Server oraz obsługą wbudowaną w Visual Studio 2008 (VS2008), dzięki której stosowanie nowego API jest niemal banalne. Zapoznamy się także z `LinqDataSource`, czyli nową kontrolkę `DataSource` stosującą w swoich poleceniach wyrażenia LINQ.

LINQ to obszerny temat, na tyle duży, że można by poświęcić mu oddzielną książkę, podobnie jak technologiom używającym LINQ. Dokładniejsze omówienie LINQ można znaleźć w książkach *LINQ in Action* (autor Fabrice Marguerie i inni, wydawnictwo Manning) oraz *Pro LINQ* (autor Joseph C. Rattz, Jr., wydawnictwo Apress). Warto także zapoznać się z ponad pięciuset przykładowymi fragmentami kodu umieszczonymi na stronie MSDN Code Gallery pod adresem *http://code.msdn.microsoft.com/csharpsamples*.

Budowa LINQ

Przejdźmy od razu do kodu i zobaczmy bardzo proste wyrażenie LINQ w działaniu. Po uruchomieniu VS2008 należy utworzyć nową witrynę internetową o nazwie *C10_LINQ* przeznaczoną dla wszystkich przykładów omówionych w rozdziale. Pracę rozpoczynamy od utworzenia i uruchomienia kilku zapytań względem znajdującej się w pamięci listy książek. Dzięki temu poznamy podstawową składnię zapytań oferowaną przez LINQ.

W VS2008 trzeba kliknąć menu *Website/Add New Item*, a następnie wskazać *Class* jako rodzaj pliku dodawanego do witryny internetowej. Nowej klasie należy nadać nazwę *Book.cs*, ustawić język jako C# i kliknąć przycisk *OK*. W pliku klasy trzeba umieścić kod przedstawiony na listingu 10.1.

Listing 10.1. Pełny kod pliku klasy Books.cs

```
using System;
using System.Collections.Generic;
public class Book
{
   public string ISBN { get; set; }
   public string Title { get; set; }
   public decimal Price { get; set; }
   public DateTime ReleaseDate { get; set; }
   public static List<Book> GetBookList()
   {
      List<Book> list = new List<Book>();
      list.Add(new Book { ISBN = "0596529562",
         ReleaseDate = Convert.ToDateTime("2008-07-15"),
         Price = 30.0m, Title = "Programming ASP.NET 3.5" });
      list.Add(new Book { ISBN = "059652756X",
         ReleaseDate = Convert.ToDateTime("2008-06-15"),
         Price = 26.0m, Title = "Programming .NET 3.5" });
      list.Add(new Book { ISBN = "0596518455",
         ReleaseDate = Convert.ToDateTime("2008-07-15"),
         Price = 28.0m, Title = "Learning ASP.NET 3.5" });
      list.Add(new Book { ISBN = "0596518439",
         ReleaseDate = Convert.ToDateTime("2008-03-15"),
         Price = 25.0m, Title = "Programming Visual Basic 2008" });
      list.Add(new Book { ISBN = "0596527438",
         ReleaseDate = Convert.ToDateTime("2008-01-15"),
         Price = 31.0m, Title = "Programming C# 3.0" });
      return list;
   }
}
```

Jak można zauważyć, klasa Book zawiera cztery właściwości oraz jedną metodę statyczną, która zwraca listę pięciu książek każdej stronie potrzebującej choć jednej. Klasa pokazuje również jedną z nowych funkcji języka w C# 3.0 — inicjalizatory obiektu — za pomocą której można konstruować egzemplarz obiektu bez konieczności używania wcześniej zdefiniowanego konstruktora.

Więcej informacji na temat inicjalizatorów obiektu oraz innych nowych funkcji C# 3.0 Czytelnik znajdzie na kolejnych stronach rozdziału.

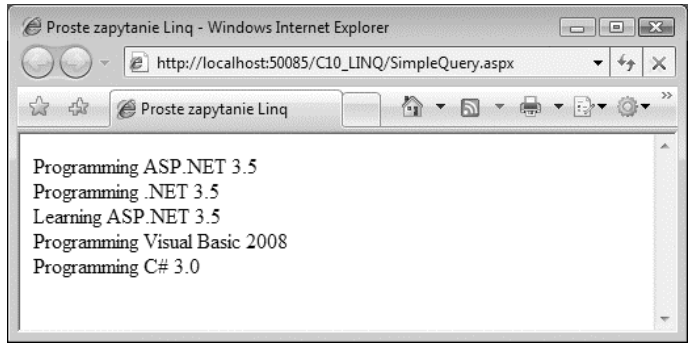
Teraz do witryny dodajemy nową stronę internetową o nazwie *SimpleQuery.aspx*, a na stronie umieszczamy kontrolkę `Label` nazwaną `lblBooks`. Po przejściu do pliku ukrytego kodu należy dodać kod przedstawiony na listingu 10.2.

Listing 10.2. Plik ukrytego kodu SimpleQuery.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.UI;
public partial class SimpleQuery : Page
{
   protected void Page_Load(object sender, EventArgs e)
   {
      List<Book> books = Book.GetBookList();
      // Używanie dołączania wewnętrznego.
      var bookTitles =
```

```
            from b in books
            select b.Title;
        foreach (var title in bookTitles)
        {
            lblBooks.Text += String.Format("{0}<br />", title);
        }
    }
}
```

Po zapisaniu i uruchomieniu strony można się przekonać, że kontrolka `Label` po prostu wyświetla tytuły książek z listy, tak jak pokazano na rysunku 10.1.



Rysunek 10.1. Strona SimpleQuery.aspx w działaniu

Można zadawać sobie pytanie, w jaki sposób to wszystko działa. Dowolne zapytanie LINQ można wykonać względem dowolnej klasy danych dziedziczącej po `IEnumerable<T>`, na przykład `List<Book>` użytej w powyższym przykładzie. Aby zagwarantować, że zapytanie będzie mogło być wykonane względem starszych zbiorów .NET v1.x, list itd., przestrzeń nazw `System.Linq` zawierająca implementację dla wszystkich operatorów zapytania (`select`, `from` itd.) ma także metodę `OfType<T>`. Wymienioną metodę można zastosować w dowolnej klasie dziedziczącej po `IEnumerable` w celu jej konwersji na jedną z dziedziczących po `IEnumerable<T>` (jeżeli programista nie chce przeprowadzać rzutowania klasy), która także będzie mogła zostać użyta wraz z LINQ.

Jeżeli Czytelnik zastanawia się co oznacza przyrostek `<T>` w nazwie klasy, wyjaśniamy, że to jest sposób deklarowania *Generics*, czyli funkcji języka wprowadzonej w C# 2.0. W języku C# 1.0 można było zadeklarować obiekt `List`, ale jego treść zawsze była traktowana jako podstawowe obiekty C#, a nie jako obiekt `Book` lub `Customer`. Ogólne rodzaje, metody i interfejsy wprowadzone w C# 2.0 pozwalają na zastąpienie znaku `T` w ich deklaracji dowolną nazwą rodzaju, która będzie zachowywana podczas operacji ogólnych — stąd `List<Book>` lub `OfType<Customer>`. Więcej informacji na temat Generics można znaleźć w książce *Programming C# 3.0*, autorstwa Jessiego Liberty'ego i Donalda Xie (wydawnictwo O'Reilly).

Rzeczywiste zapytanie LINQ jest bardzo proste i może być zinterpretowane bardziej jak polecenie SQL, chociaż z klauzulami w nieco odmiennej kolejności:

```
var bookTitles =
    from b in books
    select b.Title;
```

Zapytania LINQ mogą być umieszczane w pojedynczym wierszu, ale znacznie bardziej czytelne będzie rozbicie ich na kilka wierszy, podobnie zresztą jak w przypadku SQL. Nie należy jednak zapominać, że podobieństwo między LINQ i SQL dotyczy jedynie słów kluczowych, ale już nie samego przetwarzania. Jest to odzwierciedlone przez kolejność klauzul w zapytaniu.

Zapytanie przechodzi przez listę książek (`Books`) i zwraca zbiór implementujący `IEnumera➥ble<T>`, gdzie `T` oznacza rodzaj obiektu wynikowego. Każdy element zbioru jest tytułem książki w postaci ciągu tekstowego, tak więc `bookTitles` jest typu `IEnumerable<String>` (obiekt `StringCollection`). Ponieważ jednak można użyć nowej funkcji C# 3.0, czyli typu anonimowego, to nie trzeba podawać typu przed wykonaniem zapytania. W takim przypadku kompilator samodzielnie określi odpowiedni typ. Uff!

Wcześniejsze zapytanie można zapisać także w poniższej postaci:

```
var bookTitles = books.Select(b => b.Title);
```

Chociaż jest nieco trudniejsze w odczycie, pokazuje drugą nową funkcję C# 3.0 stosowaną przez to proste zapytanie. Wyrażenia Lambda oznaczone operatorem `=>` pobierają obiekt lub zestaw obiektów i zwracają (projekt) niektórych właściwości metodzie `Select` (lub innemu operatorowi zapytania) do użycia w innym miejscu. Wyrażenia Lambda zostaną szczegółowo omówione w dalszej części rozdziału.

Wiedząc, że zapytania LINQ zwracają zbiór pewnego rodzaju, oraz znając zawartość tego zbioru, ostatni wiersz kodu przechodzi przez wynik zapytania i umieszcza tytuły wszystkich książek we właściwości `Text` kontrolki `Label`:

```
foreach (var title in bookTitles)
{
   lblBooks.Text += String.Format("{0}<br />", title);
}
```

Ponownie w pętli `foreach` można wykorzystać typowanie anonimowe, aby uprościć sobie pracę z wynikami zapytań LINQ. Słowo kluczowe `var` nadal jest ściśle określone — po prostu sugerowane przez kompilator — i nie określa rodzaju jak słowo kluczowe `var` znane użytkownikom Visual Basic.

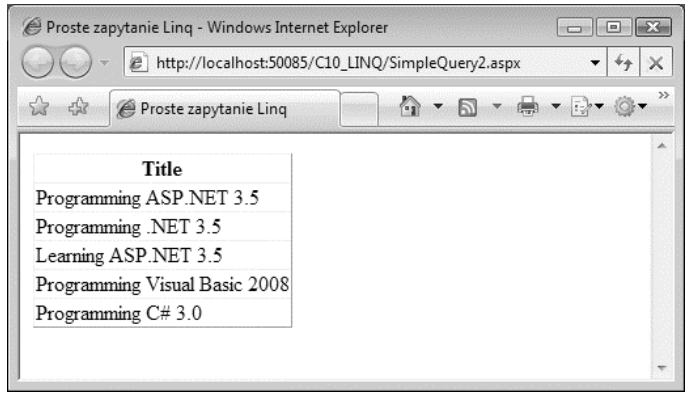
Powstaje pytanie, czy można ustawić dla właściwości `DataSource` jednej z kontrolek źródeł danych przedstawionych w rozdziale 8. wynik zapytania LINQ. Dowiedzmy się tego. Do witryny internetowej należy dodać nową stronę o nazwie *SimpleQuery2.aspx*, a na stronie trzeba umieścić kontrolkę `GridView` nazwaną `gvwBooks`. Po przejściu do pliku ukrytego kodu należy umieścić w nim procedurę obsługi zdarzeń `Page_Load` i polecenia `using` z listingu 10.2. Jedyna zmiana, którą trzeba wprowadzić, to usunięcie pętli `foreach` i zastąpienie jej poniższym przypisaniem `bookTitles` do właściwości `DataSource` kontrolki `GridView`:

```
gvwBooks.DataSource = bookTitles;
gvwBooks.DataBind();
```

Po uruchomieniu strony widzimy, że kontrolka `GridView` została wypełniona tytułami książek, tak jak pokazano na rysunku 10.2. Warto przy tym zwrócić uwagę, że nagłówek kolumny jest opisany jako „Item”.

Należy ponownie spojrzeć na wyrażenie LINQ używane do pobrania tytułów książek z listy:

```
var bookTitles =
   from b in books
   select b.Title;
```



Rysunek 10.2. Wynik zapytania LINQ użyty jako źródło danych dla kontrolki GridView

W przeciwieństwie do polecenia SQL takiego jak:

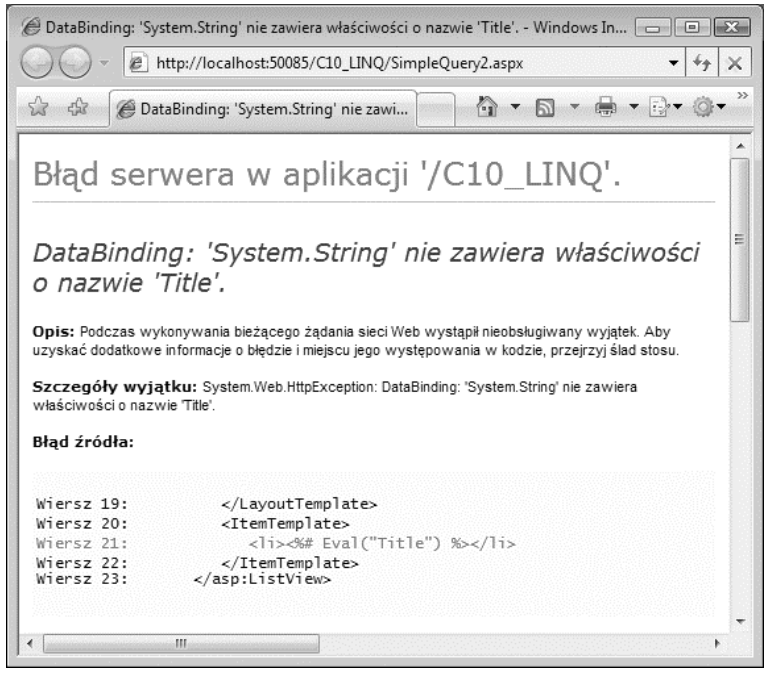
```
SELECT title FROM Books
```

wyniki polecenia LINQ są zbiorem anonimowych wartości, dlatego kontrolka `GridView` nadała kolumnie nazwę `Item`. Kontrolka wie, że w zbiorze wynikowym znajdują się wartości, ale nie wie, jakie są nazwy właściwości lub pól, gdyż nie zostały nazwane. Nie będzie to dużym problemem w przypadku kontrolki `GridView`, ale po zastąpieniu kontrolki `GridView` kontrolką używającą szablonów, na przykład `ListView`, problem stanie się istotny. Zmieńmy więc kontrolkę `GridView` na kontrolkę używającą szablonów. W kodzie strony *SimpleQuery2.aspx* usuwamy kontrolkę `GridView` i dodajemy `ListView`, jak przedstawiono na listingu 10.3.

Listing 10.3. Kod źródłowy strony SimpleQuery2.aspx wraz z kontrolką ListView

```
<%@ Page Language="C#" AutoEventWireup="true"
   CodeFile="SimpleQuery2.aspx.cs" Inherits="SimpleQuery" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title>Proste zapytanie Linq</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
      <asp:ListView runat="server" ID="lvwBooks">
         <LayoutTemplate>
            <ul>
               <asp:PlaceHolder runat="server" ID="itemPlaceholder" />
            </ul>
         </LayoutTemplate>
         <ItemTemplate>
            <li><%# Eval("Title") %></li>
         </ItemTemplate>
      </asp:ListView>
   </div>
   </form>
</body>
</html>
```

Największy problem stanowi nazwa pola dołączana do wiersza przedstawionego pogrubioną czcionką. Wybierane jest pole `b.Title`, więc prawdopodobnie można je nazwać „Title". Jednak po zapisaniu i uruchomieniu kodu zobaczymy komunikat błędu, co pokazano na rysunku 10.3.



Rysunek 10.3. Problemy z dołączaniem wartości anonimowych pobranych przez LINQ

Rozwiązaniem jest nadanie każdej pobieranej wartości nazwy, której następnie można użyć podczas operacji dołączania. Kod przedstawiony na listingu 10.4 pokazuje rozwiązanie omówionego problemu.

Listing 10.4. Nadawanie nazw wybranym właściwościom

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.UI;
public partial class SimpleQuery : Page
{
   protected void Page_Load(object sender, EventArgs e)
   {
      List<Book> books = Book.GetBookList();
      // Używanie właściwości DataSource.
      var bookTitles =
         from b in books
         select new { Title = b.Title };
      lvwBooks.DataSource = bookTitles;
      this.DataBind();
   }
}
```

Podobnie jak na wcześniejszym listingu 10.1, właściwościom zbioru wynikowego `bookTitles` można nadać nazwy, które kontrolkom używającym szablonów pozwolą na prawidłowe dołączenie danych. Po zapisaniu i ponownym uruchomieniu strony zauważymy, że kontrolka `ListView` zgodnie z oczekiwaniami dołączyła wyniki zapytania.

W omówionym przykładzie zapytanie tworzy nowy *typ anonimowy* z pojedynczą właściwością o nazwie `Title` dla każdej książki wymienionej na liście. Następnie rzutuje tytuł książki do właściwości `Title` nowego typu.

```
select new {Title = b.Title}
```

Ponieważ typ jest anonimowy, taka operacja nosi nazwę *rzutowania anonimowego*. Nowy nazwany typ można utworzyć także „w locie", jeśli zachodzi taka potrzeba:

```
select new CatalogItem {Title = b.Title}
```

W powyższym przykładzie zapytanie wykonuje *rzutowanie nieanonimowe*.

Składnia LINQ

Jak dotąd w zapytaniach LINQ widzieliśmy jedynie operatory `from` i `select`. W rzeczywistości dostępnych jest znacznie więcej operatorów, które implementują wszystkie najczęściej stosowane klauzule w zapytaniu (zobacz tabela 10.1).

Tabela 10.1. Najczęściej stosowane klauzule zapytania LINQ wymienione w kolejności ich wykonywania

Słowo kluczowe	Opis
`from`	Definiuje zakres (zestaw) początkowy danych, do których następuje zapytanie.
`join`, `on`	Definiuje dodatkowy zestaw danych, które mogą być wciągnięte do zapytania. Ponadto definiuje ich powiązanie z pierwszym zestawem danych opisanym w klauzuli `from`.
`let`	Definiuje zmienną używaną w grupowaniu bądź filtrowaniu.
`where`	Filtruje zestaw danych za pomocą pewnego warunku Boolean.
`orderby`, `orderbydescending`	Definiuje kolejność sortowania wyników zapytania.
`select`	Definiuje wartości zwracane z elementów znajdujących się w zakresie danych (bardzo często jako właściwości anonimowo określonych obiektów).
`group`	Określa, w jaki sposób zakres danych powinien być grupowany wokół zmiennej zdefiniowanej przez klauzulę `let` lub jednej z właściwości w danych.

Operatorom wymienionym w tabeli przyjrzymy się po kolei, a następnie ogólnie zapoznamy się z pełnym zestawem operatorów implementowanych przez LINQ.

Klauzula from

Pierwszą klauzulą w zapytaniu LINQ zawsze jest `from`:

```
from book in Books
```

Definiuje ona główne źródło danych w zapytaniu, które musi implementować `IEnumerable<T>`. Jeżeli typ zmiennej `Books` użytej w powyższym fragmencie kodu implementuje jedynie `IEnume`
`↪rable`, to można użyć metody LINQ `OfType<T>` w celu konwersji typu:

```
from book in Books.OfType<Book>()
```

Na listingu 10.5 przedstawiono użycie metody `OfType<T>` w ten sposób do przeprowadzenia konwersji tablicy obiektów `BookStats` — których za chwilę użyjemy podczas demonstracji klauzuli `join` — na typ dziedziczący po `IEnumerable<BookStat>`. Do katalogu *App_Code* witryny należy dodać nowy plik klasy o nazwie *BookStats.cs*, a następnie umieścić w nim kod przedstawiony na listingu 10.5.

Listing 10.5. Plik BookStts.cs z użyciem OfType<T>

```
using System.Collections.Generic;
using System.Linq;
public class BookStats
{
   public int Sales { get; set; }
   public int Pages { get; set; }
   public int Rank { get; set; }
   public string ISBN { get; set; }
   public static IEnumerable<BookStats> GetBookStats()
   {
      BookStats[] stats = {
         new BookStats { ISBN = "0596529562", Pages=904,
                         Rank=1, Sales=109000},
         new BookStats { ISBN = "0596527438", Pages=607,
                         Rank=2, Sales=58000},
         new BookStats { ISBN = "059652756X", Pages=704,
                         Rank=3, Sales=75000},
         new BookStats { ISBN = "0596518455", Pages=552,
                         Rank=4, Sales=120000},
         new BookStats { ISBN = "0596518439", Pages=752,
                         Rank=5, Sales=37500}
      };
      return stats.OfType<BookStats>();
   }
}
```

Klauzula join

Jeżeli w zapytaniu mają być wykorzystane dodatkowe źródła danych, należy użyć klauzuli `join` oraz słowa kluczowego `on` w celu zdefiniowania sposobu powiązania dodatkowych danych z już wymienionymi w zapytaniu. Przykładowo, aby połączyć przedstawione na listingu 10.5 obiekty `BookStats` z listą książek przedstawioną na listingu 10.1, można użyć poniższego kodu:

```
IEnumerable<Book> books = Book.GetBookList();
IEnumerable<BookStats> stats = BookStats.GetBookStats();
   var bookTitles =
      from b in books
      join s in stats on b.ISBN equals s.ISBN
      select new { Name = b.Title, Pages = s.Pages };
```

W omawianym przykładzie kod spowoduje połączenie dwóch zbiorów danych, bazując jedynie na współdzielonych przez nie informacjach — czyli numerze ISBN książki. Wynik zapytania będzie więc zawierał zarówno dane książek, jak i zbiór danych statystycznych. W zasadzie działa to dokładnie tak samo jak polecenie SQL `INNER JOIN`. Podobnie jak w SQL, klauzula `join` może być użyta wielokrotnie do połączenia wszystkich wymaganych oddzielnych źródeł danych w pojedynczym zapytaniu.

Zapytanie w działaniu zostało pokazane na stronie *SimpleJoin.aspx*, która znajduje się w materiałach dołączonych do książki.

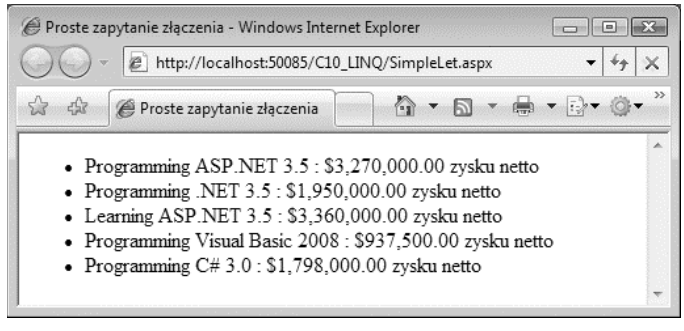
Klauzula let

Klauzula `let` pozwala na zdefiniowanie wartości przeznaczonej do użycia w kolejnych częściach zapytania. Klauzulę można więc stosować w taki sam sposób jak zmienną lokalną w metodzie. O ile zmienna ma zasięg w trakcie wykonywania metody, o tyle zasięg klauzuli `let` ma długość jednej iteracji w źródle danych podczas wykonywania zapytania.

Załóżmy na przykład, że trzeba obliczyć zysk netto ze sprzedaży książek w zbiorze. W tym celu można użyć poniższego zapytania:

```
IEnumerable<Book> books = Book.GetBookList();
IEnumerable<BookStats> stats = BookStats.GetBookStats();
var bookTitles =
   from b in books
   join s in stats on b.ISBN equals s.ISBN
   let profit = (b.Price * s.Sales)
   select new { Name = b.Title, GrossProfit = profit };
```

Jeżeli wartości te zostaną dołączone do kontrolki `ListView` lub innej kontrolki danych, zysk netto będzie prawidłowo obliczony i wyświetlony po kolei dla każdej książki, jak pokazano na rysunku 10.4.



Rysunek 10.4. Klauzula let w działaniu

Warto pamiętać, że w zapytaniu można umieścić dowolną liczbę klauzul `let`.

Zapytanie w działaniu zostało pokazane na stronie *SimpleLet.aspx*, która znajduje się w materiałach dołączonych do książki.

Klauzula where

Klauzula `where` pozwala na stosowanie filtrów warunkowych na zbiorze danych, względem którego jest wykonywane zapytanie. Jeżeli filtr przyjmie wartość `true` dla obiektu aktualnie przetwarzanego w zbiorze, obiekt ten będzie dołączony do zbioru wynikowego. Jeśli filtr przyjmie wartość `false`, bieżący obiekt nie zostanie umieszczony w zbiorze wynikowym.

Przykładowo, celem zapytania może być pobranie listy książek, których sprzedaż przekroczyła 60 000 sztuk (to musi być szczęśliwy dzień!). Tego rodzaju zapytanie można zbudować następująco:

```
IEnumerable<Book> books = Book.GetBookList();
IEnumerable<BookStats> stats = BookStats.GetBookStats();
var bookTitles =
   from b in books
   join s in stats on b.ISBN equals s.ISBN
   where s.Sales > 60000
   select new { Name = b.Title, Sales = s.Sales};
```

Istnieje również możliwość jednoczesnego zastosowania wielu filtrów. Przykładowo, jeżeli zachodzi potrzeba pobrania listy książek, które nie zostały jeszcze wydane i mają ponad 700 stron, w zapytaniu można umieścić dwie klauzule `where`:

```
var bookTitles =
   from b in books
   join s in stats on b.ISBN equals s.ISBN
   where b.ReleaseDate > DateTime.Now
   where s.Pages > 700
   select new {Name = b.Title, ReleaseDate = b.ReleaseDate, Pages = s.Pages};
```

Dopóki klauzula `where` zwraca wartość Boolean, dopóty można zastosować ją wewnątrz innej klauzuli `where`.

Zapytanie w działaniu zostało pokazane na stronie *SimpleWhere.aspx*, która znajduje się w materiałach dołączonych do książki.

Klauzule orderby i orderbydescending

Klauzule `orderby` i `orderbydescending` pozwalają na sortowanie wyniku zapytania w kolejności wskazanej na podstawie wartości jednej lub większej liczby właściwości w zbiorze wynikowym. Przykładowo, przedstawione poniżej zapytanie zwróci listę wszystkich książek posortowaną w kolejności wydania od najstarszej do najnowszej. Jeżeli więcej niż jedna książka będzie miała taką samą datę wydania, to zostaną posortowane względem liczby stron:

```
IEnumerable<Book> books = Book.GetBookList();
IEnumerable<BookStats> stats = BookStats.GetBookStats();
var bookTitles =
   from b in books
   join s in stats on b.ISBN equals s.ISBN
   orderby b.ReleaseDate, s.Pages
   select new {Name = b.Title, Pages = s.Pages, ReleaseDate = b.ReleaseDate};
```

Użycie klauzuli `orderbydescending` zamiast `orderby` powoduje odwrócenie kolejności sortowania.

Zapytanie w działaniu zostało pokazane na stronie *SimpleOrderBy.aspx*, która znajduje się w materiałach dołączonych do książki.

Klauzula select

Ostatnią częścią zapytania LINQ zawsze musi być klauzula `select` — albo sama klauzula, albo jako część klauzuli `groupby`, która zostanie omówiona jako kolejna. Wymienione klauzule definiują informacje pobierane przez zapytanie. Jak już wcześniej pokazano, klauzulę `select` można wykorzystać w następujących celach:

- *pobrania* pojedynczego fragmentu informacji typu anonimowego lub nazwanego;
- *rzutowania* wielu fragmentów informacji na typ anonimowy bądź nazwany;
- *pobrania* całego obiektu, względem którego jest wykonywane zapytanie.

Ponadto właściwości wymienionych typów anonimowych lub nazwanych będą miały nadane nazwy, choć muszą być wyraźnie zarejestrowane, gdy dane będą dołączane do kontrolek serwerowych ASP.NET. Jeżeli nazwa nie zostanie wyraźnie ustawiona, zastosowana będzie taka sama nazwa, jaką ma właściwość wskazywana w pierwszej kolejności.

Przykładowo, poniższe zapytanie zwraca zbiór egzemplarzy typów anonimowych zawierających właściwość o nazwie `Title`:

```
var bookTitles =
   from b in books
   select b.Title;
```

Z kolei poniższe zapytanie zwraca zbiór obiektów `CategoryItem`, z których każdy ma dwie właściwości o nazwach `Title` oraz `BookId`:

```
var bookTitles =
   from b in books
   select new CategoryItem { b.Title, BookId = b.ISBN };
```

Wreszcie kolejne zapytanie zwraca zbiór obiektów przechowywanych w `bookTitles`. Obiekty te zachowają własne nazwy oraz właściwości, jeżeli nie będą typami anonimowymi:

```
var bookTitles =
   from b in books
   where b.ReleaseDate > DateTime.Now
   select b;
```

Klauzuli `select` można użyć także do przekształcenia wyników na postać ułatwiającą pracę:

```
var bookTitles =
   from b in books
   select new { ISBN = b.ISBN, ISBN13 = "978-" + b.ISBN  };
var bookTitles =
   from b in books
   select new { ISBN = b.ISBN,
      Released = (b.ReleaseDate < DateTime.Now ? "Niedostępna" : "Już wkrótce")};
```

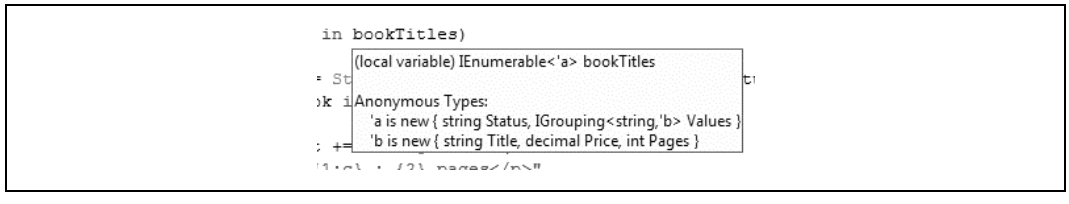
Zapytanie w działaniu zostało pokazane na stronie *SimpleSelect.aspx*, która znajduje się w materiałach dołączonych do książki.

Klauzula group

Klauzula `group` definiuje sposób, w jaki wyniki zapytania powinny być zwracane w postaci grup, oraz właściwość kluczową, na której ma bazować grupowanie. Przykładowo, jeżeli zachodzi potrzeba pobrania listy książek pogrupowanych na podstawie tego, czy zostały już wydane, można użyć poniższego zapytania:

```
var bookTitles =
   from b in books
   join s in stats on b.ISBN equals s.ISBN
   let outYet = (b.ReleaseDate < DateTime.Now ? "Niedostępna" : "Już wkrótce")
   orderby s.Rank
   group new { Title = b.Title, Price = b.Price, Pages = s.Pages }
   by outYet
      into groupedBooks
      select new
      {
         Status = groupedBooks.Key,
         Values = groupedBooks
      };
```

Jak można zauważyć, grupowanie powoduje zwiększenie poziomu skomplikowania zapytania, ale warto zastanowić się na otrzymanymi wynikami. Zamiast pojedynczego zbioru wynikowego (`IEnumerable<Results>`) zapytanie podzieli zbiór na kilka oddzielnych na podstawie wartości kluczowej. Dlatego też zapytanie obecnie zwraca *kolekcję* (zbiór wyników oraz wartość właściwości, względem której wyniki zostały pogrupowane). W rzeczywistości, po umieszczeniu kursora myszy nad `bookTitles` w przedstawionym kodzie, lista *IntelliSense* pokazuje prawdziwą strukturę wyników (zobacz rysunek 10.5).



Rysunek 10.5. Prawdziwa struktura pogrupowanych danych

Wracamy do zapytania. Klauzula `group` przedstawiona pogrubioną czcionką w powyższym fragmencie kodu składa się z dwóch oddzielnych części. W wierszu pierwszym zdefiniowano rzeczywiste informacje, które powinny być pobrane dla każdej książki (wystarczy po prostu zastąpić słowo kluczowe `group` słowem `select`, aby poznać sens tego wiersza):

```
group new { Title = b.Title, Price = b.Price, Pages = s.Pages }
```

Pozostała część definiuje sposób, w jaki rzeczywiste informacje będą podzielone na grupy. Zdefiniowano zmienną lokalną o nazwie `outYet`, która może przyjąć jedną z dwóch wartości. Informacje o książce zostaną więc podzielone na dwie grupy w zależności od wartości zmiennej `outYet` dla każdej książki:

```
by outYet
```

Każda grupa (do której lokalnie się odnosimy, używając nazwy znajdującej się po słowie kluczowym `into`) będzie przechowywała wartość `outYet` w swojej wartości `Key`:

```
into groupedBooks
```

W celu zakończenia zapytania pogrupowane dane są zbierane za pomocą wartości klucza, względem której zostały pogrupowane:

```
select new
{
   Status = groupedBooks.Key,
   Values = groupedBooks
};
```

Nowa struktura wyników zapytania oznacza, że nie można ich teraz po prostu użyć jako źródła danych dla prostej kontrolki dołączającej dane, takiej jak `ListView`. Zamiast tego trzeba ręcznie przejść przez kolekcję i pobierać pogrupowane dane oraz wartości kluczowe, a następnie przejść ponownie przez pogrupowane dane w celu pobrania zwracanych przez nie informacji. Aby zademonstrować takie rozwiązanie, do witryny *C10_LINQ* dodajemy nową stronę internetową o nazwie *SimpleGroupBy.aspx*. Na stronie umieszczamy pojedynczą kontrolkę `Label` o nazwie `lblBooks`. Zawartość pliku ukrytego kodu zastępujemy kodem przedstawionym na listingu 10.6.

Listing 10.6. Tworzenie i używanie grupowanych zbiorów wynikowych LINQ w pliku SimpleGroupBy.aspx.cs

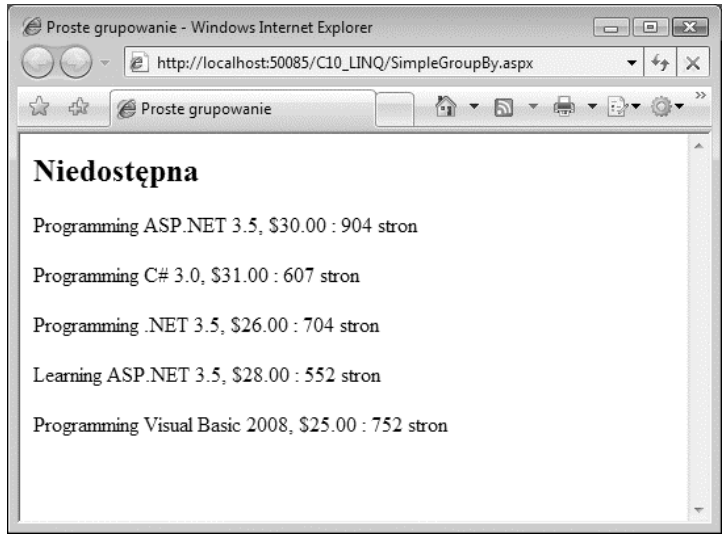
```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Web.UI;
public partial class SimpleGroupBy : Page
{
   protected void Page_Load(object sender, EventArgs e)
   {
      IEnumerable<Book> books = Book.GetBookList();
      IEnumerable<BookStats> stats = BookStats.GetBookStats();
      var bookTitles =
         from b in books
         join s in stats on b.ISBN equals s.ISBN
         let outYet =
            (b.ReleaseDate < DateTime.Now ? "Niedostępna" : "Już wkrótce")
         orderby s.Rank
         group new { Title = b.Title, Price = b.Price, Pages = s.Pages }
         by outYet
            into groupedBooks
            select new
            {
               Status = groupedBooks.Key,
               Values = groupedBooks
            };
      foreach (var group in bookTitles)
      {
         lblBooks.Text += String.Format("<h2>{0}</h2>", group.Status);
         foreach (var book in group.Values)
         {
            lblBooks.Text += String.Format(
               "<p>{0}, {1:c} : {2} stron</p>",
               book.Title, book.Price, book.Pages);
         }
      }
   }
}
```

Na rysunku 10.6 pokazano stronę po jej uruchomieniu.

Warto jednak zwrócić uwagę, że chociaż nazwy wydają się powiązane z szablonem grupowania kontrolki `ListView`, to przeznaczenie szablonu nie jest takie samo jak grupowania w zapytaniu LINQ. Nie należy więc próbować mapować ich względem siebie.

Inne operatory zapytania LINQ

Poza omówionymi powyżej siedmioma standardowymi klauzulami zapytania LINQ implementuje znacznie więcej standardowych operatorów zapytania, które przynajmniej częściowo mogą być znane Czytelnikowi.



Rysunek 10.6. Strona SimpleGroupBy.aspx w działaniu

W celu znacznie dokładniejszego poznania wszystkich operatorów warto poświęcić chwilę i pobrać przykłady C# dla VS2008 ze strony MSDN Code Gallery pod adresem *http://code.msdn.microsoft.com/csharpsamples*. Znajduje się tam około 500 przykładów zapytań LINQ przedstawiających każdy operator znacznie dokładniej niż w niniejszej książce.

W tabeli 10.2 wymieniono operatory zapytania pobierające dwa zbiory danych. Wartością zwrotną jest (w pewien sposób) połączenie obu zbiorów.

Tabela 10.2. Implementacja ustawiania operatorów arytmetycznych w LINQ

Operator	Opis
`Union(set1, set2)`	Używany w celu utworzenia z dwóch zestawów pojedynczego zestawu danych zawierającego jedynie unikalne elementy obu zestawów.
`Except(set1, set2)`	Używany w celu utworzenia z dwóch zestawów pojedynczego zestawu danych zawierającego jedynie wartości w `set1`, które nie znajdują się w `set2`.
`Intersect(set1, set2)`	Używany w celu utworzenia z dwóch zestawów pojedynczego zestawu danych zawierającego jedynie wartości w `set1`, które znajdują się także w `set2`.
`Concat(set1, set2)`	Używany w celu utworzenia z dwóch zestawów pojedynczego zestawu danych, w którym zawartość `set2` zostanie umieszczona po `set1`.

W tabeli 10.3 wymieniono operatory zapytania generujące nowy zbiór danych, którego następnie można użyć w kodzie.

W tabeli 10.4 wymieniono operatory zapytania grupujące zbiory danych, przeprowadzające pewne funkcje na grupie, a następnie zwracające pojedynczy zbiór wynikowy.

W tabeli 10.5 wymieniono operatory wpływające na liczbę elementów znajdujących się w zbiorze wynikowym zapytania, który zostaje faktycznie zwrócony do kodu.

Tabela 10.3. Implementacja operatorów generowania w LINQ

Operator	Opis
`Range(seed, długość)`	Zwraca zestaw wszystkich liczb całkowitych z zakresu od `seed` do `(seed+długość-1)`.
`Repeat(wynik, liczba)`	Zwraca zestaw zawierający daną `liczbę` egzemplarzy `wyniku`.
`Empty()`	Zwraca zbiór pusty.

Tabela 10.4. Implementacja operatorów matematycznych w LINQ

Operator	Opis
`Count()`	Zwraca liczbę elementów w wywoływanym zbiorze.
`Sum()`	Zwraca sumę elementów (przy założeniu, że wszystkie są wartościami liczbowymi) w zbiorze.
`Min()`	Zwraca najniższą wartość elementu (przy założeniu, że wszystkie są wartościami liczbowymi) w zbiorze.
`Max()`	Zwraca najwyższą wartość elementu w zbiorze.
`Average()`	Zwraca wartość średnią w zbiorze liczb.
`Aggregate(funkcja)`	Wykonuje wskazaną funkcję na dwóch pierwszych liczbach zbioru, następnie na obliczonej wartości całkowitej i trzecim elemencie, dalej na obliczonej wartości całkowitej i czwartym elemencie itd.

Tabela 10.5. Implementacja operatorów ustawiania elementów składowych w LINQ

Operator	Opis
`Take(in)`	Określa liczbę elementów w bazowym zbiorze danych, które będą zawarte w wyniku zapytania.
`Skip(int)`	Określa liczbę elementów w bazowym zbiorze danych, które nie będą zawarte w wyniku zapytania.
`Reverse()`	Odwraca kolejność elementów w zbiorze wynikowym.
`Distinct()`	Upewnia się, że zbiór wynikowy nie zawiera duplikatów.
`First()`	Zwraca jedynie pierwszy wynik zapytania.
`FirstOrDefault()`	Zwraca jedynie pierwszy wynik zapytania lub wartość domyślną tego rodzaju, jeżeli zbiór wynikowy zapytania jest pusty.
`ElementAt(indeks)`	Zwraca jedynie wynik zapytania znajdujący się w określonym indeksie zbioru.
`Last()`	Zwraca jedynie ostatni wynik zapytania.
`LastOrDefault()`	Zwraca jedynie ostatni wynik zapytania lub wartość domyślną tego rodzaju, jeżeli zbiór wynikowy zapytania jest pusty.
`ElementAtOrDefault(indeks)`	Zwraca jedynie wynik zapytania znajdujący się w określonym indeksie zbioru lub wartość domyślną tego rodzaju, jeżeli zbiór wynikowy zapytania jest pusty.
`Single(wyrażenie)`	Zwraca pojedynczą wartość ze zbioru wynikowego, która powoduje spełnienie podanego wyrażenia. Jeżeli nie ma takiej wartości lub jest ich więcej niż jedna, operator zwraca błąd.
`SingleOrDefault(wyrażenie)`	Zwraca pojedynczą wartość ze zbioru wynikowego, która powoduje spełnienie podanego wyrażenia. Jeżeli jest ich więcej niż jedna, funkcja zwraca błąd. W przypadku braku wartości spełniającej wyrażenie zwracana jest wartość domyślna dla danego rodzaju.

Wreszcie w tabeli 10.6 wymieniono operatory, które można zastosować w klauzuli `where` zapytania.

Tabela 10.6. Implementacja operatorów Boolean w LINQ

Operator	Opis
`Any(warunek)`	Zwraca wartość Boolean określającą, czy podany warunek jest spełniany przez jakikolwiek element zbioru.
`All(warunek)`	Zwraca wartość Boolean określającą, czy podany warunek jest spełniany przez wszystkie elementy zbioru.
`SequenceEqual(sekwencjaB)`	Zwraca wartość `true`, jeżeli `sekwencjaB` zawiera dokładnie te same elementy i w dokładnie takiej samej kolejności jak sekwencja, wobec której wywoływany jest operator `SequenceEqual`.
`Contains(wartość)`	Zwraca wartość `true`, jeśli zbiór zawiera podaną wartość.

Za kulisami zapytania LINQ: C# 3.0 w działaniu

Za kulisami zapytanie LINQ może używać jednocześnie do pięciu nowych funkcji języka C# 3.0. Kompilator C# wykorzystuje te funkcje w celu ponownego zapisania zapytania oraz obsłużenia wyników zapytania w wymienionych poniżej wywołaniach metod i typach deklaracji, których faktycznie może użyć:

- typy anonimowe i inicjalizatory obiektu;
- niejawnie określone zmienne lokalne;
- metody rozszerzające;
- wyrażenia Lambda.

Przyjrzymy się kolejno każdej pozycji z powyższej listy.

Typy anonimowe i inicjalizatory obiektu

Bardzo często programista nie chce tworzyć nowej klasy wyłącznie w celu przechowywania wyników zapytania. Języki .NET 3.x oferują tak zwane *typy anonimowe*, które pozwalają na zadeklarowanie zarówno klasy anonimowej, jak i egzemplarza tej klasy przy użyciu inicjalizatorów obiektu. Przykładowo, anonimowy obiekt książki można zainicjalizować w następujący sposób:

```
new { Title = "Programming ASP.NET 3.5",
      ReleaseDate = Convert.ToDateTime("2008-07-15"),
      Stats = bookStats };
```

Powyżej przedstawiono deklarację klasy anonimowej z trzema własnościami — `Title`, `Release` ↪`Data` i `Stats` — oraz inicjalizację tych zmiennych za pomocą ciągu tekstowego, klasy `Date` ↪`Time` i egzemplarza klasy `BookStats`. Kompilator C# może określać typy właściwości na podstawie przypisanych im wartości. Dlatego też właściwość `ReleaseData` jest typu `DateTime`, natomiast właściwość `Stats` jest typu `BookStats`. Podobnie jak w przypadku zwykłych, nazwanych klas, klasy anonimowe mogą mieć właściwości dowolnego typu.

W tle dla każdego nowego typu kompilator C# generuje unikalną nazwę. Ponieważ do tej nazwy nie można odnieść się w kodzie aplikacji, typ jest uznawany za nieposiadający nazwy.

Jeżeli Czytelnik jest ciekaw, to w celu dokładnego ustalenia wywoływanych klas może użyć aplikacji takiej jak Reflector (*http://www.red-gate.com/products/reflector/index.htm*).

Niejawnie określone zmienne lokalne

W każdym przedstawionym dotąd przykładzie wyniki zapytania były przypisywane zmiennej typu `var`:

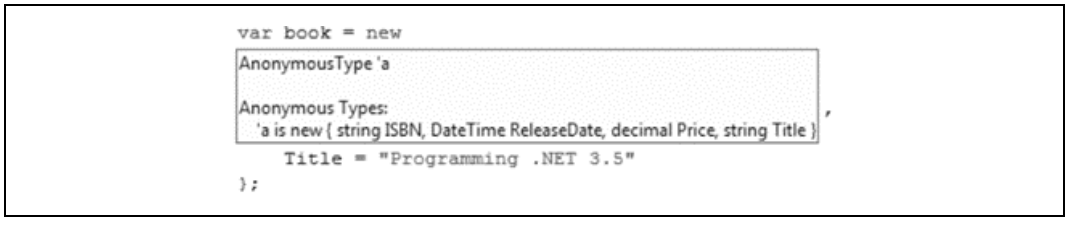
```
var bookTitles =
   from b in books
   select b.Title;
```

Ponieważ klauzula `select` zwraca egzemplarz typu anonimowego, nie można jawnie zdefiniować typu `IEnumerable<T>`. Na szczęście C# 3.0 oferuje inną funkcję, nazywaną niejawnie określonymi zmiennymi lokalnymi, które rozwiązują ten problem.

Niejawnie określoną zmienną lokalną można zadeklarować poprzez ustawienie jej typu jako `var`:

```
var pages = 902;
var isbn = "0596529562";
var stats = new List<BookStats>();
var book = new {ISBN = "059652756X",
                ReleaseDate = Convert.ToDateTime("2008-06-15"),
                Price = 26.0m, Title = "Programming .NET 3.5"};
```

Kompilator C# ustala typ niejawnie określonej zmiennej lokalnej na podstawie jej wartości początkowej. Dlatego też taką zmienną trzeba zainicjalizować podczas jej zadeklarowania. W powyższym fragmencie kodu typ `pages` został ustawiony jako liczba całkowita, typ `isbn` jako ciąg tekstowy, a typ `stats` jako ściśle określony `List<T>` obiektów `BookStats`. Typ ostatniej zmiennej `book` jest typem anonimowym zawierającym cztery właściwości: `ISBN`, `ReleaseDate`, `Price` i `Title`. Chociaż w kodzie ten typ nie ma nazwy, kompilator C# po cichu przydziela mu nazwę i śledzi egzemplarze tego typu. W rzeczywistości lista *IntelliSense* środowiska IDE Visual Studio również jest powiadamiana o typach anonimowych, co pokazano na rysunku 10.7.



Rysunek 10.7. Lista IntelliSense śledzi typy anonimowe

Jak wyjaśniono wcześniej, wynik dowolnego zapytania LINQ jest zmienną typu `IEnumerable<T>`, gdzie argument `T` to typ (anonimowy bądź nazwany), który zawiera nazwane właściwości w klauzuli `select` lub `group`. Po zdefiniowaniu zapytania można przechodzić przez wyniki za pomocą pętli `foreach`, jak przedstawiono na wcześniejszym listingu 10.2:

```
var bookTitles =
   from b in books
   select b.Title;
foreach (var title in bookTitles)
{
   lblBooks.Text += String.Format("{0}<br />", title);
}
```

Ponieważ wynik jest niejawnie określonym `IEnumerable<T>`, gdzie `T` to ciąg tekstowy, zmienna iteracji również będzie niejawnie rzutowana do tej samej klasy — `String`. Dla każdego obiektu w zbiorze wynikowym przykład ten po prostu wyświetli właściwości obiektu.

Taka sama zasada ma zastosowanie względem wyników pogrupowanego zapytania przedstawionego na wcześniejszym listingu 10.6:

```
foreach (var group in bookTitles)
{
   lblBooks.Text += String.Format("<h2>{0}</h2>", group.Status);
   foreach (var book in group.Values)
   {
      lblBooks.Text += String.Format(
         "<p>{0}, {1:c} : {2} stron</p>",
         book.Title, book.Price, book.Pages
      );
   }
}
```

Zmienna iteracji grupująca wyniki w pętli zewnętrznej jest typu `IEnumerable<T>`, gdzie `T` oznacza niejawny typ `{string, IGrouping<string, U>}`. W tym typie `U` wskazuje na niejawny typ zmiennej iteracji `book` — `{string, decimal, int}`.

Metody rozszerzające

Metody rozszerzające to sztuczka stosowana przez kompilator — metody statyczne rozszerzające klasy, do których w innym przypadku nie można dodawać metod, na przykład:

```
"someString".PrefixWith("asd"); //Zwraca asdsomeString.
```

zamiast:

```
StringExt.PrefixWith("someString", "asd");
```

Jeżeli Czytelnik zna choć trochę SQL, wyrażenia zapytania przedstawione w poprzednim podrozdziale okażą się całkiem intuicyjne i łatwe do zrozumienia, ponieważ LINQ jest formułowany w sposób podobny do SQL. Ponieważ kod C# jest ostatecznie wykonywany przez .NET CLR, kompilator C# musi przekształcić wyrażenia zapytania na format zrozumiały przez środowisko uruchomieniowe platformy .NET. Ponieważ CLR rozumie wywołania metod, które mogą być wykonywane, wyrażenia zapytania LINQ napisane w języku C# są przekształcane na serię wywołań metod.

Na przykład, poniższe zapytanie:

```
var query =
   from book in books
   where book.Price > 25m
   select book;
```

jest przez kompilator przekształcane na:

```
var query =
    books.Where(book => book.Price > 25m)
         .Select(book => book);
```

Ponieważ metoda `select` nie wykonuje niczego na książce (nie rzutuje obiektu `book` na inną postać), to może zostać pominięta:

```
var query =
   books.Where(book => book.Price > 25m);
```

Jeżeli zapytanie zwraca jedynie na przykład cenę książki, polecenie `select` będzie miało następującą postać:

```
var query =
   books.Where(book => book.Price > 25m)
        .Select(book => book.Price);
```

W rzeczywistości wszystkie standardowe operatory LINQ są metodami rozszerzonymi i podczas kompilacji zostaną przez kompilator napisane na nowo, podobnie jak przedstawiona powyżej.

Tworzenie własnych metod rozszerzających

Podobnie jak przypadku wszystkich opisanych dotąd funkcji, dla zapewnienia wygody istnieje także możliwość tworzenia własnych metod rozszerzonych w dowolnej aplikacji. Jeżeli programista kiedykolwiek napisał klasę nazwaną `Utils`, `StringExt`, `DateExt` itd., to występuje duże prawdopodobieństwo, że metody znajdujące się w wymienionych klasach są dobrymi kandydatami do przepisania ich na postać metod rozszerzających.

Zapoznajmy się z przykładem. Jedną z możliwych do przepisania metod narzędziowych klasy `StringExt` jest `PrefixWith()`. Jak można się spodziewać, dodaje ona określony ciąg tekstowy prefiksu do już istniejącego. Przed zmianą jej na metodę rozszerzającą była wywoływana w następujący sposób:

```
StringExt.PrefixWith(someString, prefixString);
```

Po zaimplementowaniu jako metoda rozszerzająca może być wywoływana w sposób przedstawiony poniżej, prawie jak rzeczywista klasa `System.String` zawierająca tę metodę:

```
someString.PrefixWith(prefixString);
```

Zmiana metody „standardowej" na „rozszerzającą" jest bardzo łatwa. Na listingu 10.7 przedstawiono pełny kod źródłowy klasy, w której `PrefixWith` zdefiniowano jako metodę rozszerzającą.

Listing 10.7. Plik Extensions.cs

```
using System;
public static class StringExt
{
   public static string PrefixWith(
      this string someString, string prefixString)
   {
      return prefixString + someString;
   }
}
```

W języku C# metoda rozszerzająca musi być zdefiniowana jako metoda statyczna klasy statycznej. Pierwszy parametr metody rozszerzającej oznaczony słowem kluczowym `this` zawsze wskazuje typ docelowy, którym w omawianym przykładzie jest `string`. Dlatego też powyższy kod definiuje `PrefixWith` jako rozszerzenia klasy `string`.

Wszystkie kolejne parametry są zwykłymi parametrami metody rozszerzającej. Część główna metody nie różni się niczym od zwykłych metod. Przedstawiona powyżej funkcja po prostu zwraca przekształcony ciąg tekstowy.

Aby użyć metody rozszerzającej, musi się ona znajdować w tym samym zasięgu, w którym jest kod klienta. Jeżeli metoda rozszerzająca będzie zdefiniowana w innej przestrzeni nazw, trzeba zastosować dyrektywę `using` importującą przestrzeń nazw, w której zdefiniowano metodę rozszerzającą. W przeciwieństwie do zwykłych metod nie można używać pełnych nazw metod rozszerzającej. Poza tym ograniczeniem używanie metody rozszerzającej jest identyczne z używaniem dowolnych metod wbudowanych typu docelowego. W omawianym przykładzie następuje po prostu wywołanie metody `System.String`, nawet jeśli metoda jest elementem składowym klasy `StringExt`.

Warto w tym miejscu wspomnieć, że metody rozszerzające są w pewnych kwestiach bardziej rygorystyczne od zwykłych metod składowych — metody rozszerzające mogą uzyskać dostęp jedynie do publicznych elementów składowych typu docelowego. Uniemożliwia to naruszenie hermetyzacji typów docelowych.

Wyrażenia Lambda

Wyrażeń Lambda można użyć w celu zdefiniowania definicji delegatów wewnątrz kodu. W przedstawionym poniżej wyrażeniu:

```
book => book.Price > 25m
```

lewy operand — `book` — jest parametrem danych wejściowych. Prawy operand to wyrażenie Lambda, które sprawdza, czy właściwość `Price` obiektu `book` ma wartość większą niż `25`. Następnie rzutuje wartość na postać typu anonimowego lub nazwanego, który będzie wynikiem wyrażenia. Dlatego też dla danego obiektu książki przeprowadzane jest sprawdzenie, czy cena książki jest wyższa niż 25. Wyrażenie Lambda następnie zostaje przekazane metodzie `Where()` w celu przeprowadzenia operacji porównania względem każdej książki znajdującej się na liście.

Zapytania zdefiniowane z użyciem metod rozszerzających są nazywane *zapytaniami bazującymi na metodach*. Chociaż składnia metody i zapytania jest odmienna, to semantycznie są identyczne i kompilator przekształca je na taki sam kod IL. Programista może więc używać dowolnej z nich w zależności od własnych upodobań.

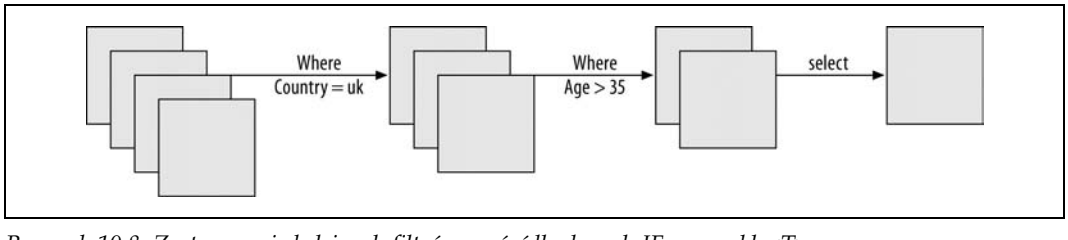
IEnumerable dobrze, IQueryable lepiej

Jak już wcześniej wspomniano, zapytania LINQ mogą być zastosowane jedynie względem typów implementujących `IEnumerable<T>`. Warto jednak dodać, że jeśli klasa zawiera opracowany przez programistę zestaw danych implementujących również `IQueryable<T>` (który dziedziczy po `IEnumerable<T>`), takie rozwiązanie będzie znacznie bardziej pożądane.

Przyjmujemy założenie, że mamy obiekt `Collection` mapujący 1000 rekordów w innym komputerze, i wykonujemy następujące zapytanie:

```
for Customer c in Customers
where c.Country == "uk"
where c.Age > 35
select ......
```

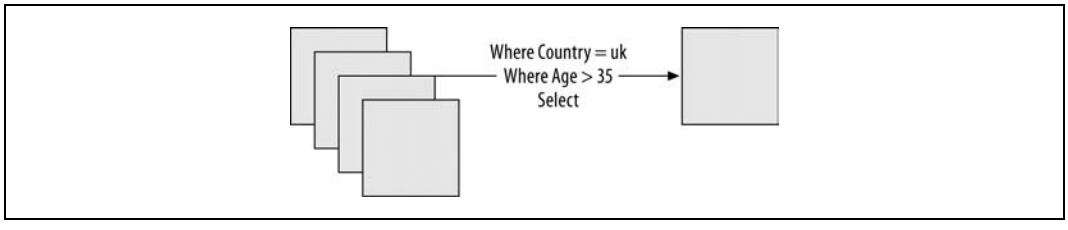
Jeżeli obiekt `Customers` implementuje jedynie `IEnumerable<Customer>`, zapytanie spowoduje pobranie 1000 rekordów do komputera lokalnego jeszcze przed zastosowaniem jakiegokolwiek filtrowania. Następnie będzie stosować po kolei każdy filtr (klauzula `where`), jak pokazano na rysunku 10.8.



Rysunek 10.8. Zastosowanie kolejnych filtrów na źródle danych IEnumerable<T>

Zapytanie przechodzi przez każdą klauzulę znajdującą się w zapytaniu LINQ, ale jeśli źródło danych zostało zmienione między kolejnymi operacjami, to wyniki będą się różniły.

Dla porównania — jeżeli obiekt `Customers` implementuje `IQueryable<Customer>`, to wszystkie operacje filtrowania są połączone w jedną (pod względem technicznym oznacza to połączenie w pojedyncze drzewo wyrażenia). Dlatego też zapytanie będzie wykonane w zdalnym komputerze tylko jednorazowo *podczas żądania danych*. Technicznie nosi to nazwę *wykonania odroczonego*, jest szybszym i stabilniejszym sposobem dostarczenia wyników z (ogromnych) źródeł danych, jak pokazano na rysunku 10.9.



Rysunek 10.9. Wykonanie odroczone, w którym wszystkie filtry zostają nałożone jednocześnie

Poprzez wywołanie `ToList<T>` na samym zapytaniu lub wynikach zapytania istnieje również możliwość wymuszenia wykonania zapytania w dowolnej chwili, na przykład:

```
List<Book> books =
   bookList.Select(book => book.Price > 25m).ToList<Book>();
```

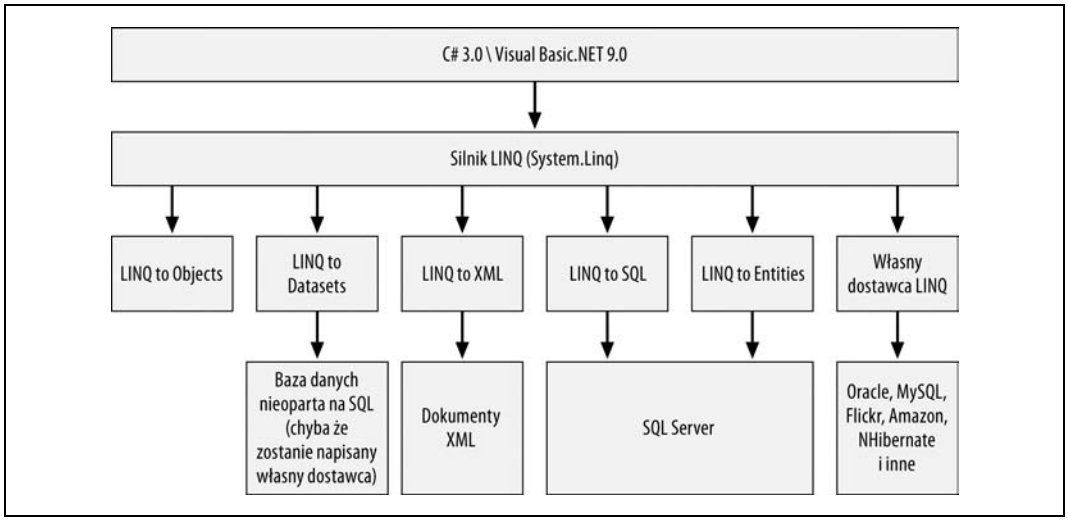
Dostawcy LINQ

A zatem za pomocą języka C# 3.0 LINQ działa w charakterze pośrednika między C# i dowolnym magazynem danych. Biblioteki w przestrzeni nazw `System.Linq` implementują różne klauzule i operatory zapytania, wymienione w przedstawionych wcześniej tabelach od 10.1 do 10.6. Owe operatory z kolei komunikują się z dostawcami LINQ, natomiast LINQ wie, jak zastosować te zapytania względem określonych źródeł danych, co pokazano na rysunku 10.10.

Platforma .NET 3.5 jest dostarczana z czterema wbudowanymi dostawcami LINQ:

- Możliwość wykonywania zapytań względem tablic, list, słowników i innych znajdujących się w pamięci źródeł informacji zademonstrowana jak dotąd w rozdziale jest znana jako *LINQ to Objects* i także stanowi część `System.Linq`.
- Możliwość wykonywania zapytań względem dokumentu XML jest znana jako *LINQ to XML* i została zaimplementowana w `System.Xml.Linq`.
- Możliwość wykonywania zapytań względem dowolnej bazy danych SQL Server jest znana jako *LINQ to SQL* i została zaimplementowana w `System.Data.Linq`.
- Możliwość wykonywania zapytań do innego, dowolnego rodzaju bazy danych jest obecnie zaimplementowana poprzez umieszczanie danych w obiekcie `DataSet` znajdującym się w pamięci, a następnie wykonywanie zapytań do wymienionego obiektu. Taką operację umożliwia zestaw rozszerzeń zaimplementowanych w `System.Data.DataSetExtensions`.

Piąty dostawca LINQ znany jako *LINQ to Entities* jest dostępny jako część .NET 3.5 Service Pack 1. To po prostu „przemysłowa" wersja dostawcy *LINQ to SQL*.



Rysunek 10.10. Graficzna prezentacja LINQ i predefiniowanych dostawców

Jak pokazano na rysunku 10.10, pewna liczba zewnętrznych dostawców LINQ pozwala na wykonywanie zapytań do wielu różnych źródeł danych, takich jak Oracle, MySQL, Flickr, usługi sieciowe Amazon, NHibernate i inne.

Lista aktualnych zewnętrznych dostawców LINQ znajduje się na stronie *http://oakleafblog.blogspot.com/2007/03/third-party-linq-providers.html*, choć wiele z nich jest nazywanych *LINQ to xzy*. Wyszukiwarka Google prawdopodobnie będzie największym przyjacielem Czytelnika w odkryciu dodatkowych informacji o dostawcy dla źródła, które ma zostać wykorzystane.

Zagadnienie tworzenia własnego dostawcy LINQ wykracza poza zakres tematyczny niniejszej książki. Istnieje jednak wiele pomocnych zasobów, jeśli Czytelnik będzie chciał spróbować. W pozostałej części rozdziału omówimy dwóch głównych dostawców dostarczanych z VS2008: LINQ to XML oraz LINQ to SQL.

LINQ to XML

Dostawca LINQ to XML wczytuje dokument XML do pamięci i przekształca go na zestaw obiektów (takich jak `XElement` i `XAttribute`), względem których można wykonywać zapytania. Wymienione obiekty w pełni opisują dokument i pozwalają na poruszanie się po nich w stylu XPath i XQuery.

Na listingu 10.8 przedstawiono bardzo prosty dokument XML zawierający informacje o tym, które książki zostały napisane przez danego autora. Plik należy utworzyć i dodać do witryny *C10_LINQ*. Szczegółowe informacje dotyczące autorów znajdują się w bazie danych AdventureWorksLT, do której dostęp uzyskamy w podrozdziale poświęconemu dostawcy LINQ to SQL. Natomiast informacje szczegółowe o książkach znajdują się w utworzonych wcześniej obiektach przechowywanych w pamięci.

Listing 10.8. Plik Authors.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<authorlist>
   <author id="1">
      <book isbn="0596529562" />
      <book isbn="059652756X" />
   </author>
   <author id="10">
      <book isbn="059652756X" />
      <book isbn="0596527438" />
   </author>
   <author id="38">
      <book isbn="0596518439" />
   </author>
   <author id="201">
      <book isbn="0596518439" />
      <book isbn="0596527438" />
   </author>
</authorlist>
```

Przede wszystkim trzeba utworzyć stronę wyświetlającą identyfikatory autorów. Do witryny *C10_LINQ* dodajemy nową stronę internetową o nazwie *SimpleXmlQuery.aspx*. Na stronie umieszczamy kontrolkę `Label` o nazwie `lblAuthors`. Plik ukrytego kodu strony zastępujemy kodem przedstawionym na listingu 10.9.

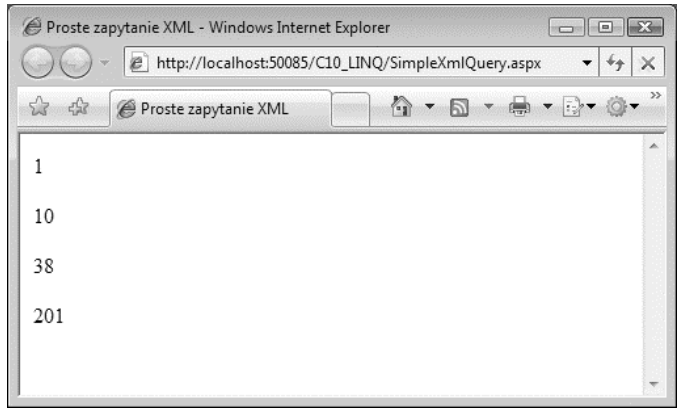
Listing 10.9. Plik ukrytego kodu SimpleXmlQuery.aspx.cs

```
using System;
using System.Linq;
using System.Web.UI;
using System.Xml.Linq;
public partial class SimpleXmlQuery : Page
{
   protected void Page_Load(object sender, EventArgs e)
   {
      XElement doc = XElement.Load(Request.ApplicationPath + "\\authors.xml");
      var authorIds = from authors in doc.Elements("author")
                      select authors.Attribute("id").Value;
      foreach (var id in authorIds)
      {
         lblAuthors.Text += String.Format("<p>{0}</p>", id);
      }
   }
}
```

Po zapisaniu pliku i uruchomieniu strony powinniśmy otrzymać wyświetlone cztery identyfikatory — 1, 10, 38 i 201, jak pokazano na rysunku 10.11.

Kluczowy fragment kodu na powyższym listingu 10.9 został przedstawiony pogrubioną czcionką. Klasa `XElement` jest używana w celu wczytania dokumentu XML do pamięci. Następnie do zbioru wszystkich elementów w dokumencie mających w nazwie „author" wykonywane są zapytania w celu pobrania wartości każdego z ich atrybutów `id`.

Warto zwrócić uwagę, że `XElement.Load` wymaga podania ścieżki dostępu w systemie plików wskazującej plik *authors.xml* (na przykład *C:\Projekty\authors.xml*) zamiast wirtualnego adresu URL (*http://localhost/authors.xml*). Ponadto zastosowano `Request.ApplicationPath` w celu konwersji katalogu głównego witryny internetowej na jego odpowiednik w systemie plików.



Rysunek 10.11. Strona SimpleXmlQuery w działaniu

Jeżeli Czytelnik chce spróbować, to może użyć zmiennej wyniku zapytania `authorsIds` jako źródła danych (`DataSource`) i dołączyć je do kontrolki źródeł danych. Jednak znacznie łatwiej jest powtórzyć zapytanie i w wynikach nadać nazwę ciągowi tekstowemu zawierającemu identyfikator autora, jeśli dane mają być dołączane do kontrolki wykorzystującej szablony:

```
var authorIds = from authors in doc.DescendantsAndSelf("author")
   select new { AuthorId = authors.Attribute("id").Value };
```

Przykład stosujący omówione zapytanie i dołączający dane do kontrolki `ListView` można znaleźć na stronie *SimpleXmlQuery2.aspx*, która znajduje się w materiałach dołączonych do książki.

Klasa `XElement` dostarcza dużą liczbę metod (na przykład `Elements` użytą w poprzednim przykładzie) zwracających zbiór `IEnumerable<T>` dla zapytania, przez który trzeba przejść. Metody te zostały wymienione w tabeli 10.7.

Warto zwrócić uwagę, że wszystkie metody zwracają zbiory obiektów przedstawiających obiekty XML *względne wobec obiektu bieżącego*. Jeżeli to wydaje się nieprzekonujące, należy pamiętać, że wymienione metody rozszerzające mogą być ze sobą łączone. Dlatego też, gdy zachodzi potrzeba pobrania wszystkich wartości `ID` autorów, którzy napisali książkę o ISBN równym 059652756X, istnieją co najmniej dwa sposoby wykonania takiego zapytania.

W pierwszym zapytanie może wyszukiwać każdy element `<book>` w pliku *authors.xml*, który ma odpowiednią wartość atrybutu `isbn`. Następnie sprawdzi element nadrzędny `<author>` znalezionego elementu i pobierze wartość jego atrybutu `id`:

```
var authorIds =
   from book in doc.DescendantsAndSelf("book")
      let authorId = book.Ancestors("author").Attributes("id").Single()
      where book.Attribute("isbn").Value == "059652756X"
      select new { AuthorId = authorId.Value };
```

W drugim zapytanie może przejść przez wszystkie elementy `author`, a następnie sprawdzić zbiór wszystkich atrybutów `isbn` dla wszystkich elementów potomnych `<book>` danego elementu `<author>`. Jeżeli którykolwiek z nich będzie miał odpowiednią wartość, zapytanie zatrzyma wartość atrybutu `id` bieżącego elementu `<author>`.

Tabela 10.7. Zbiory dostępne w obiekcie XElement

Zbiór	Opis
`Ancestors(`*nazwa*`)`	Zbiór elementów `XElements` przedstawiający elementy nadrzędne bieżącego elementu XML na wszystkich poziomach aż do najwyższego. Jeżeli zostanie podana *nazwa*, zbiór będzie ograniczony jedynie do elementów znajdujących się w podanej nazwie.
`AncestorsAndSelf(`*nazwa*`)`	Zbiór elementów `XElements` przedstawiający elementy nadrzędne bieżącego elementu XML na wszystkich poziomach aż do najwyższego plus sam bieżący obiekt. Jeżeli zostanie podana *nazwa*, zbiór będzie ograniczony jedynie do elementów znajdujących się w podanej nazwie.
`Annotations(T), Annotations<T>()`	Zbiór obiektów przedstawiających przypisy do bieżącego obiektu rodzaju `T`.
`Attributes(`*nazwa*`)`	Zbiór elementów `XAttribute` przedstawiający atrybuty bieżącego elementu XML. Jeżeli zostanie podana *nazwa*, atrybuty zostaną ograniczone do wskazanych.
`DescendantNodes()`	Zbiór elementów `XNodes` przedstawiający wszystkie węzły potomne (elementy plus wartości tekstowe) bieżącego elementu. Zbiór zostaje wygenerowany z zachowaniem kolejności elementów w dokumencie.
`DescendantNodesAndSelf()`	Zbiór elementów `XNodes` przedstawiający wszystkie węzły potomne (elementy plus wartości tekstowe) bieżącego elementu plus same element bieżący. Zbiór zostaje wygenerowany z zachowaniem kolejności elementów w dokumencie.
`Descendants(`*nazwa*`)`	Zbiór elementów `XElements` przedstawiający wszystkie elementy potomne XML bieżącego obiektu. Zbiór zostaje wygenerowany z zachowaniem kolejności elementów w dokumencie. Jeżeli zostanie podana *nazwa*, elementy zostaną ograniczone do wskazanych.
`DescendantsAndSelf(`*nazwa*`)`	Zbiór elementów `XElements` przedstawiający wszystkie elementy potomne XML bieżącego obiektu oraz sam obiekt bieżący. Zbiór zostaje wygenerowany z zachowaniem kolejności elementów w dokumencie. Jeżeli zostanie podana *nazwa*, elementy zostaną ograniczone do wskazanych.
`Elements(`*nazwa*`)`	Zbiór elementów `XElements` przedstawiający wszystkie elementy potomne XML bieżącego obiektu. Jeżeli zostanie podana *nazwa*, elementy zostaną ograniczone do wskazanych.
`ElementsBeforeSelf(`*nazwa*`)`, `ElementsAfterSelf(`*nazwa*`)`	Zbiór pokrewnych elementów `XElements` znajdujących się przed lub za bieżącym w kolejności elementów w dokumencie. Jeżeli zostanie podana *nazwa*, elementy zostaną ograniczone do wskazanych.
`Nodes(`*nazwa*`)`	Zbiór elementów `XNodes` przedstawiający wszystkie węzły potomne XML (elementy oraz tekst) obiektu bieżącego. Jeżeli zostanie podana *nazwa*, węzły zostaną ograniczone do wskazanych.
`NodesBeforeSelf(`*nazwa*`)`, `NodesAfterSelf(`*nazwa*`)`	Zbiór pokrewnych elementów `XNodes` znajdujących się przed lub za bieżącym w kolejności elementem w dokumencie. Jeżeli zostanie podana *nazwa*, elementy zostaną ograniczone do wskazanych.

```
var authorIds2 =
   from author in doc.DescendantsAndSelf("author")
      where author.Elements("book").Attributes("isbn")
                  .Any(attr => attr.Value == "059652756X")
      select new { AuthorId = author.Attribute("id").Value };
```

Przykład stosujący oba omówione zapytania można znaleźć na stronie *SimpleXmlQuery3.aspx*, która znajduje się w materiałach dołączonych do książki.

Dołączanie XML do różnego rodzaju danych

Jedną z największych zalet używania LINQ jest możliwość łatwego łączenia różnych rodzajów danych, tak jakby były danymi tego samego rodzaju. W bieżącym przykładzie utworzymy stronę łączącą listę `Book` zdefiniowaną wcześniej w klasie *Books.cs* z danymi w pliku *authors.xml* oraz grupami `authorIds` za pomocą tytułów książek, które napisali autorzy. Ogólnie rzecz biorąc, strona odwróci związek autor-do-książki opisany w pliku XML, natomiast lista książek będzie używana w celu opisania książki za pomocą jej tytułu, a nie numeru ISBN.

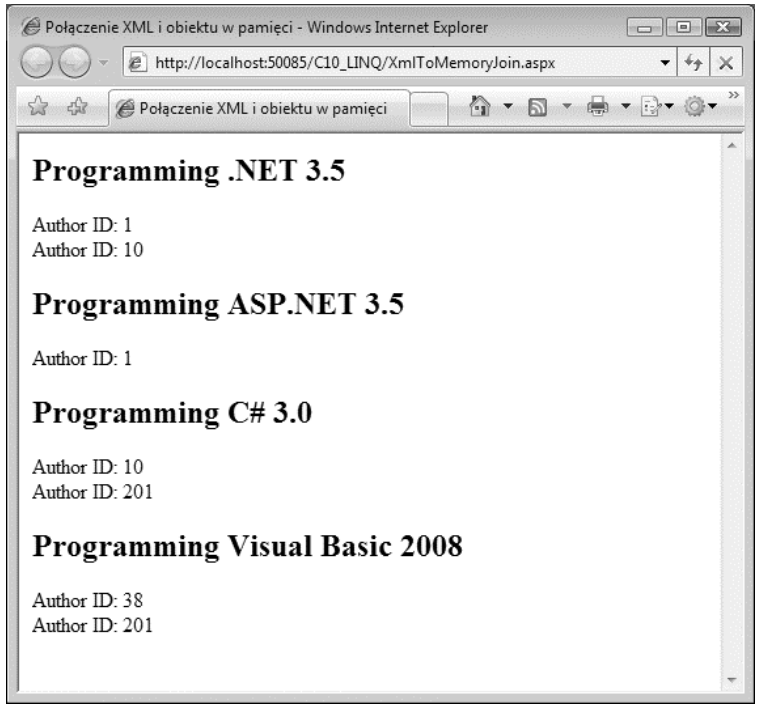
Rozpoczynamy od dodania nowej strony internetowej do witryny *C10_LINQ*. Stronie nadajemy nazwę *XmlToMemoryJoin.aspx* i umieszczamy na niej pojedynczą kontrolkę `Label` o identyfikatorze `lblBooks`. Po otworzeniu pliku ukrytego kodu zastępujemy istniejący kod przedstawionym na listingu 10.10.

Listing 10.10. Plik ukrytego kodu XmlToMemoryJoin.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.UI;
using System.Xml.Linq;
public partial class XmlToMemoryJoin : Page
{
   protected void Page_Load(object sender, EventArgs e)
   {
      List<Book> bookList = Book.GetBookList();
      XElement doc =
         XElement.Load(Request.ApplicationPath + "\\authors.xml");
      var authorsByBooks =
         from book in doc.DescendantsAndSelf("book")
         join bookInfo in bookList on book.Attribute("isbn").Value
            equals bookInfo.ISBN
         let authorId = book.Parent.Attribute("id").Value
         orderby bookInfo.Title
         group new { AuthorId = authorId }
         by bookInfo.Title
            into groupedAuthors
            select new
            {
               Title = groupedAuthors.Key,
               Authors = groupedAuthors
            };
      foreach (var book in authorsByBooks)
      {
         lblBooks.Text += String.Format("<h2>{0}</h2>", book.Title);
         foreach (var author in book.Authors)
         {
            lblBooks.Text += String.Format("Author ID: {0}<br />",
               author.AuthorId);
         }
      }
   }
}
```

Po zapisaniu i uruchomieniu strony zobaczymy wyniki pokazane na rysunku 10.12.

Zapoznamy się z kodem procedury obsługi zdarzeń `Page_Load`. Przede wszystkim następuje utworzenie dwóch źródeł danych:



Rysunek 10.12. Strona XmlToMemoryJoin.aspx w działaniu

```
protected void Page_Load(object sender, EventArgs e)
{
   List<Book> bookList = Book.GetBookList();
   XElement doc =
      XElement.Load(Request.ApplicationPath + "\\authors.xml");
```

Następnie rozpoczyna się zapytanie. Dwa źródła danych są ze sobą łączone za pomocą wartości ciągu tekstowego numeru ISBN. Zapytanie przechodzi więc przez elementy `<book>` w dokumencie XML, zamiast przez elementy `<author>`, i łączy dwa źródła danych:

```
var authorsByBooks =
   from book in doc.DescendantsAndSelf("book")
   join bookInfo in bookList on book.Attribute("isbn").Value
      equals bookInfo.ISBN
```

Teraz zapytanie ustala wartość `authorId` dla książki w dokumencie XML. Wiadomo, że element `<author>` jest elementem nadrzędnym dla `<book>`. Dlatego też można wykorzystać metodę `Parent` klasy `XElement` w celu pobrania szukanej wartości atrybutu `id` przy minimalnym wysiłku:

```
let authorId = book.Parent.Attribute("id").Value
```

Mając wyniki pogrupowane w kolejności tytułów książek:

```
orderby bookInfo.Title
```

zapytanie przeprowadza rzeczywiste grupowanie identyfikatorów autorów:

```
group new { AuthorId = authorId }
```

względem tytułu książki, do której powstania się przyczynili:

```
   by bookInfo.Title
      into groupedAuthors
      select new
      {
         Title = groupedAuthors.Key,
         Authors = groupedAuthors
      };
```

Na koniec, po zakończeniu przetwarzania danych, funkcja przechodzi przez wyniki zapytania i wyświetla najpierw tytuł książki jako wartość kluczową dla każdej grupy identyfikatorów autorów, a następnie same identyfikatory:

```
      foreach (var book in authorsByBooks)
      {
         lblBooks.Text += String.Format("<h2>{0}</h2>", book.Title);
         foreach (var author in book.Authors)
         {
            lblBooks.Text += String.Format("Author ID: {0}<br />",
               author.AuthorId);
         }
      }
   }
```

Trzeba w tym miejscu koniecznie wspomnieć, że przedstawiony ogólny kształt zapytania nie ulega zmianie nawet pomimo tego, iż zapytanie działa z dwoma zupełnie odmiennymi rodzajami danych.

Tworzenie XML za pomocą LINQ

Jak dotąd można przypuszczać, że LINQ to API działające jedynie w trybie do odczytu i pozbawione możliwości zapisu nowych lub przekształconych danych z powrotem w źródle danych, z których pierwotnie pochodzą. Choć to prawda odnośnie do samego LINQ, jest już nieprawdą w przypadku różnych dostawców LINQ. W rzeczywistości dostawcę LINQ to XML można wykorzystać do zapisywania nowych dokumentów XML w przeglądarce internetowej bądź z powrotem do pliku na dysku. Funkcja ta jest udostępniana dzięki elastyczności nowego API XML używanego przez LINQ.

Klasa `XElement` ma przeciążonego konstruktora w poniższej postaci:

```
XElement xml = new XElement(string nazwa, object elementyPotomne)
```

Kluczową kwestią tutaj jest możliwość użycia zapytania LINQ w celu zapełnienia obiektu `elementyPotomne`, a tym samym zbudowania obiektu `XElement`, którego metoda `ToString()` wygeneruje dokument XML.

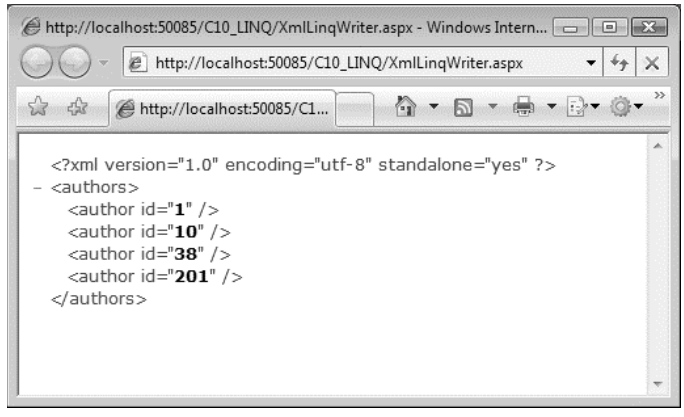
Spójrzmy na przykład. Do witryny *C10_LINQ* dodajemy kolejną stronę internetową o nazwie *XmlLinqWriter.aspx*. Przechodzimy do widoku *Source view* i usuwamy wszystko poza dyrektywą `Page`. Strona wygeneruje czysty dokument XML, więc nie potrzebujemy żadnego kodu HTML. Następnie otwieramy plik ukrytego kodu strony i zastępujemy jego treść kodem przedstawionym na listingu 10.11.

Listing 10.11. Plik ukrytego kodu XmlLinqWriter.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.UI;
using System.Xml.Linq;
```

```
public partial class XmlLinqWriter : Page
{
   protected void Page_Load(object sender, EventArgs e)
   {
      List<Book> bookList = Book.GetBookList();
      XElement doc =
         XElement.Load(Request.ApplicationPath + "\\authors.xml");
      XElement xml = new XElement("authors",
         from author in doc.DescendantsAndSelf("author")
         select new XElement("author",
            new XAttribute("id", author.Attribute("id").Value)
            )
         );
      Response.Write(new XDeclaration("1.0", "utf-8", "yes").ToString());
      Response.Write(xml.ToString());
   }
}
```

Po zapisaniu i uruchomieniu strony dokument XML wygenerowany przez stronę zostanie wyświetlony w przeglądarce internetowej (zobacz rysunek 10.13).



Rysunek 10.13. Czysty dokument XML wyświetlony na ekranie przez stronę XmlLinqWriter.aspx

Jeśli przyjrzeć się dokładniej powyższemu kodowi, można zauważyć, że zapytanie LINQ zostało osadzone w konstruktorze obiektu `XElement`:

```
XElement xml = new XElement("authors",
   from author in doc.DescendantsAndSelf("author")
   select new XElement("author",
      new XAttribute("id", author.Attribute("id").Value)
      )
   );
```

Zewnętrzny konstruktor tworzy element główny nowego dokumentu — `<authors>`:

```
XElement xml = new XElement("authors", obiektyPotomne);
```

W omawianym przykładzie obiekt `obiektyPotomne` będzie wynikiem zapytania LINQ. Jeśli pominąć na chwilę wstawianie dokumentu XML, zastosowane tutaj zapytanie jest bardzo proste. Przechodzi przez elementy `<author>` pliku *authors.xml* i zwraca zbiór identyfikatorów wszystkich autorów:

```
from author in doc.DescendantsAndSelf("author")
select new { author.Attribute("id").Value };
```

Jedyną różnicą między tym i poprzednim zapytaniem jest to, że zamiast tworzenia egzemplarza typu anonimowego klauzula `select` tworzy nowy obiekt `XElement` i osadza w nim wartości `id`. W rzeczywistości istnieje możliwość osadzenia wartości zapytania bezpośrednio w dowolnym typie, o ile jest to konieczne.

Zbudujmy nieco bardziej skomplikowane zapytanie i wygenerujmy inny dokument XML. Na listingu 10.10 dokument *authors.xml* był łączony z znajdującą się w pamięci listą książek, a wartością zwrotną była lista autorów dla każdej książki wymienionej na liście. Przyjmując założenie, że chcemy wygenerować dokument XML o następującej strukturze:

```
<books>
   <book title="...">
      <author id="..." />
      ...
   </book>
   ...
</books>
```

możemy osadzić zapytanie LINQ użyte na listingu 10.10 w konstruktorze klasy `XElement` dla elementu głównego `<books>` i zbudować żądany dokument. Konstruktor będzie się przedstawiał następująco (fragmenty faktycznie odpowiedzialne za generowanie XML zostały przedstawione pogrubioną czcionką):

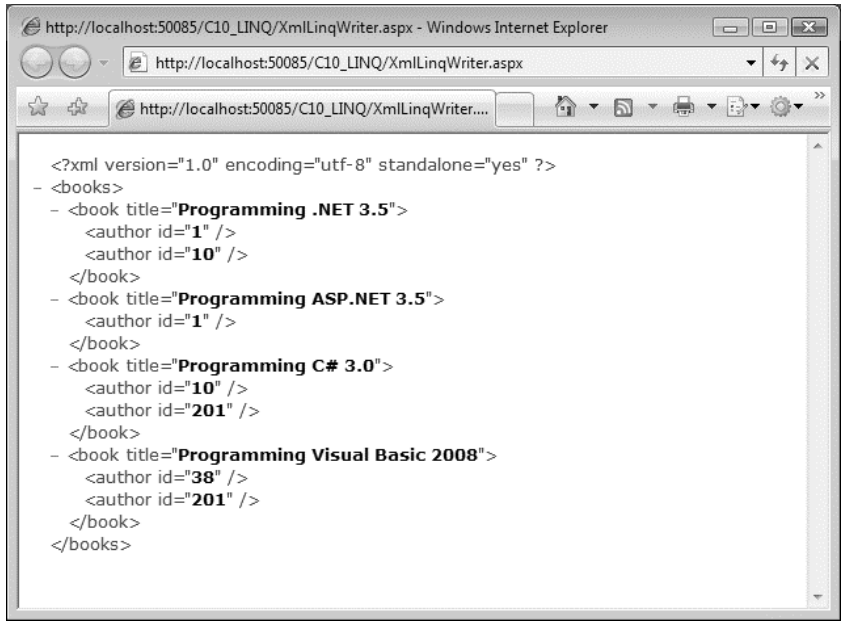
```
XElement xml = new XElement("books",
   from book in doc.DescendantsAndSelf("book")
   join bookInfo in bookList on book.Attribute("isbn").Value
      equals bookInfo.ISBN
   let authorId = book.Parent.Attribute("id").Value
   orderby bookInfo.Title
   group new XElement("author", new XAttribute("id", authorId))
   by bookInfo.Title
      into groupedAuthors
      select new XElement("book",
         new XAttribute("title", groupedAuthors.Key),
         groupedAuthors
      )
   );
```

Jeżeli powyższy kod dodamy do kodu w pliku *XmlLinqWriter.aspx.cs* i umieścimy w komentarzu drugi konstruktor `XElement`, to po uruchomieniu strony zobaczymy wyświetlony dokument końcowy, który został pokazany na rysunku 10.14.

Wiele przykładów zapytań LINQ to XML można znaleźć w przykładach C# dla VS2008 dostępnych w MSDN Code Gallery na stronie *http://code.msdn.microsoft.com/csharpsamples*.

LINQ to SQL

Przejdźmy do baz danych, które najczęściej są wykorzystywane jako główne źródło danych dla witryny internetowej. Microsoft oferuje dostawcę LINQ umożliwiającego komunikację SQL Server z platformą .NET 3.5, a także kilka innych narzędzi do użycia z pakietem VS2008, dzięki którym życie programisty staje się nieco łatwiejsze. Chodzi tutaj przede wszystkim o obiekt `LinqDataSource` i technologię ORM (mapowanie obiektowo-relacyjne). Przeznaczenie pierwszego z wymienionych jest całkiem oczywiste, ale w odniesieniu do drugiego rodzi się pytanie:



Rysunek 10.14. Używanie pogrupowanego zapytania LINQ w celu wygenerowania dokumentu XML

Czego nie można tworzyć, uaktualniać lub usuwać w bazie danych SQL Server?

Odpowiedź: *obiektów*.

Od związków do obiektów

Problem polega na tym, że SQL Server nie może przedstawić obiektów dziedziczących po `IQueryable` i `IEnumerable`. W rzeczywistości w ogóle nie może przedstawiać obiektów, jedynie rekordy i kolumny. Żadnych właściwości, zdarzeń, a nawet metod. Dlatego też w przypadku obiektu `Book` z właściwościami `ISBN`, `Title` i `ReleaseDate`, zapisanie takiego obiektu w bazie danych — jak się przekonaliśmy — wymaga przeprowadzenia skomplikowanych operacji konwersji obiektu na postać zwykłych danych, które mogą być obsłużone przez bazę danych. Zazwyczaj oznacza to przedstawienie obiektu poprzez rekord tabeli, a właściwości za pomocą kolumn wewnątrz tego rekordu. Podobnie podczas pobierania danych z bazy danych pobrane dane muszą być skonwertowane na postać odpowiednich obiektów i właściwości, aby program mógł normalnie funkcjonować.

Użycie obiektów `DataSource` i technologii ADO.NET umożliwia przezwyciężenie tej *niezgodności impedancji*, ale jedynie poprzez pozwolenie na zdefiniowanie sposobu, w jaki elementy mogą być wybrane bądź zapisane w bazie danych w ramach podanego kontekstu. Znacznie bardziej pożądanym rozwiązaniem jest posiadanie warstwy dostępu do danych, która mapuje obiekty do treści jednej lub kilku tabel. W takim przypadku, aby zapisać obiekt w bazie danych, trzeba jedynie wywołać metodę `Save()`, natomiast pobranie zbioru jest możliwe dzięki metodzie `Select()`.

Ogólnie rzecz biorąc, taka możliwość istnieje już od dawna — jest to coś, co nazywamy warstwą *mapowania obiektowo-relacyjnego* (ORM). W pakiecie VS2008 przy wykorzystaniu odrobiny C# i ADO.NET można to zrobić na wiele różnych sposobów. Generalnie dzięki LINQ takie zadanie okazuje się łatwiejsze niż dotychczas.

Przykładowo, przedstawionego na listingu 10.12 szkieletu kodu można użyć do mapowania prostych operacji względem tabeli `SalesLT.Customer` w bazie danych AdventureWorksLT (i wreszcie nie martwić się ciągle o przyrostek `SalesLT`).

Listing 10.12. LINQ oferuje elegancką składnię mapowania tabeli do klasy

```
[Database(Name="AdventureWorksLT")]
public partial class AdventureWorksLT : DataContext
{
   [Table(Name="SalesLT.Customer")]
   public partial class Customer
   {
      [Column(Storage="_CustomerID", DbType="Int NOT NULL IDENTITY",
             IsPrimaryKey=true, IsDbGenerated=true)]
      public int CustomerID { get; set; }
      [Column(Storage="_NameStyle", DbType="Bit NOT NULL")]
      public bool NameStyle { get; set; }
      ...
   }
}
```

Oczywiście, utworzenie pełnego kodu dla tej tabeli i pozostałej części bazy danych zabrałoby wieki, ale na szczęście nie trzeba tego robić. Zamiast tego wystarczy użyć narzędzia Object Relational Desinger wbudowanego w VS2008, które wygeneruje warstwę ORM dla tylu tabel bazy danych, ile jest koniecznych. Wymieniona warstwa składa się z:

Klas lub Entity Framework przedstawiających bazę danych
: Każda tabela będzie mapowana do swojej klasy, a każda kolumna tej tabeli będzie mapowana do odpowiedniej właściwości w danej klasie. Dodatkowo każda właściwość jest ściśle określonego typu, który odpowiada typowi mapowanej kolumny w bazie danych. Jeżeli nastąpi próba umieszczenia w kolumnie wartości błędnego typu, kompilator wygeneruje komunikat ostrzeżenia.

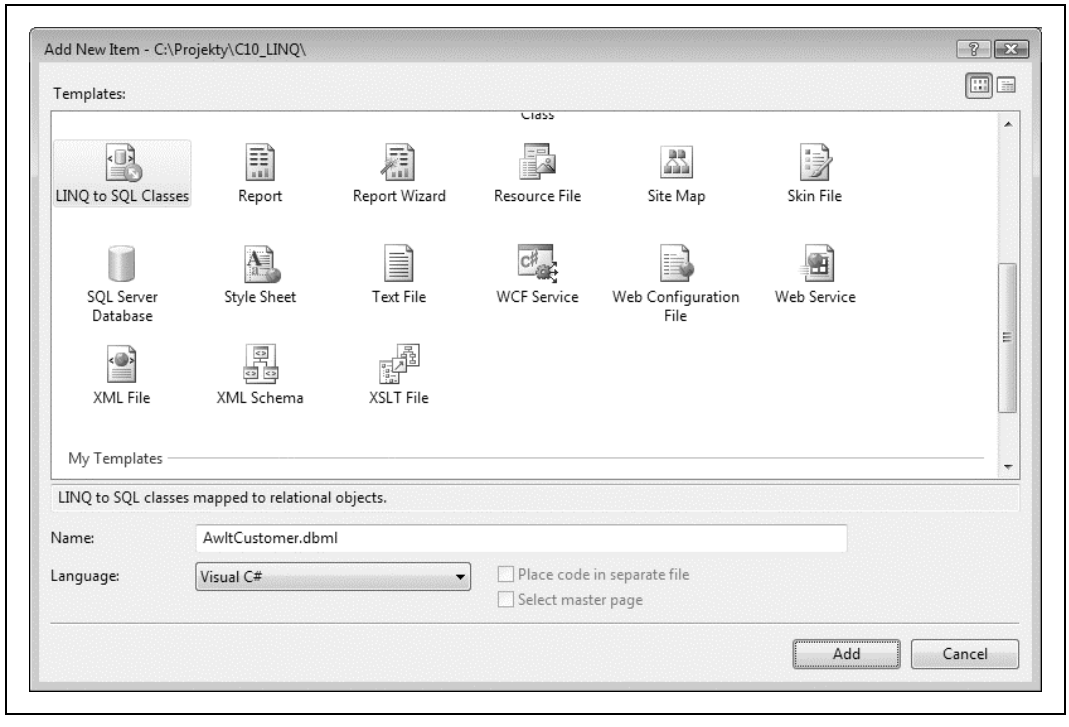
Klasa `DataContext` *działająca w charakterze pomostu między modelem obiektowym LINQ i samą bazą danych*
: Obiekt ten będzie używany w celu wysyłania i odbierania danych między Entity Framework i bazą danych.

Narzędzie generujące klasę `DataContext` nie jest unikalne pod względem możliwości utworzenia warstwy ORM na podstawie treści bazy danych. Obecnie na rynku dostępnych jest kilka dojrzałych produktów ORM, a także kilkanaście wariantów typu open source, które można pobrać bezpłatnie. Chociaż nie jest do końca solidny, to coraz większa popularność modelu ActiveRecord używanego przez Ruby on Rails do generowania modeli danych dla architektury Model-View-Controller (MVC) niewątpliwie przyczyniła się do uwydatnienia i spopularyzowania wykorzystania tego rodzaju produktów w sieci.

Mając to wszystko na uwadze, spójrzmy na przykład i zobaczmy, co potrafi narzędzie Object Relational Designer.

Używanie narzędzia Object Relational Designer

Aby użyć narzędzia Object Relational Designer do utworzenia warstwy ORM dla LINQ, której będzie można używać na stronach internetowych, należy w VS2008 kliknąć menu *Website/Add New Item*. W wyświetlonym oknie dialogowym trzeba wskazać *LINQ to SQL Classes*. W omawianym przykładzie utworzymy warstwę na podstawie informacji o klientach przechowywanych w bazie danych AdventureWorksLT, więc plikowi należy nadać nazwę *AwltCustomer.dbml*, jak pokazano na rysunku 10.15. Jeżeli pojawi się pytanie, trzeba się zgodzić na zapisanie pliku w katalogu *App_Code* witryny.

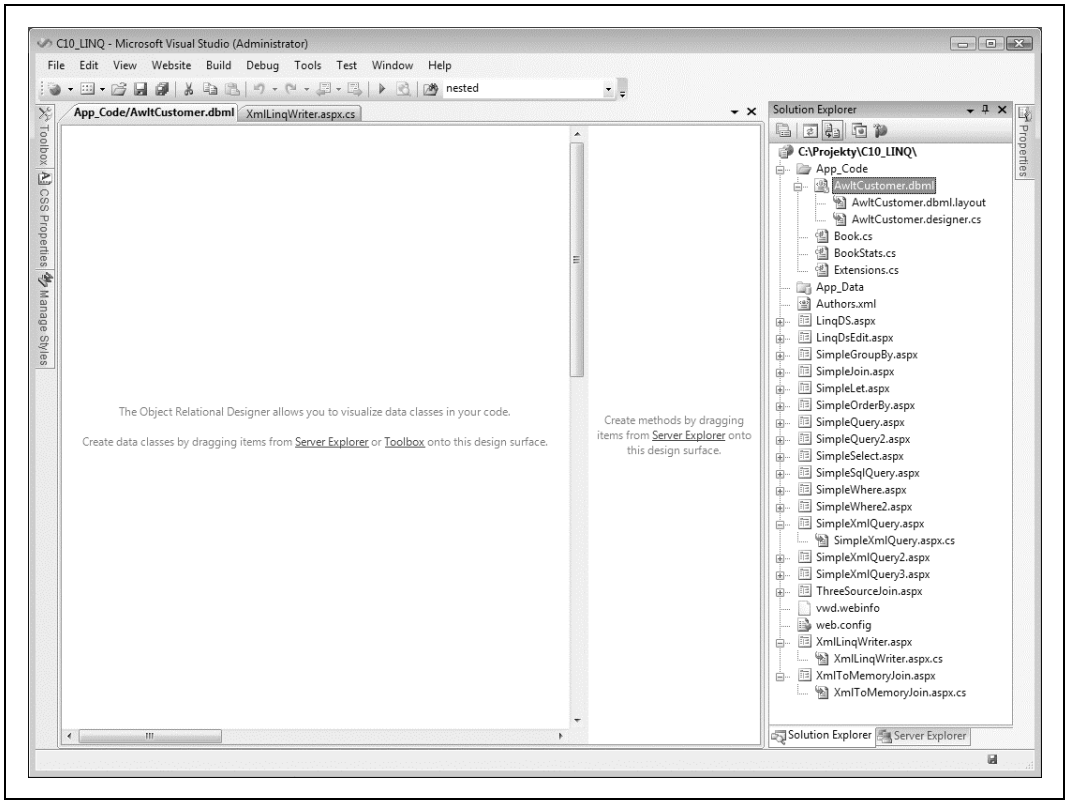


Rysunek 10.15. Dodawanie do witryny internetowej nowej warstwy ORM w VS2008

Po chwili do katalogu *App_Code* witryny internetowej zostanie dodany plik *AwltCustomer.dbml*, natomiast w oknie głównym VS2008 wyświetli się narzędzie Object Relational Designer (zobacz rysunek 10.16).

Na tym etapie pasek narzędziowy (Toolbox) zawiera kontrolki używane w narzędziu Object Relational Designer. Programista może tworzyć własne klasy poprzez przeciągnięcie kontrolki `Class` na przestrzeń roboczą. Do klasy można dodać właściwości — wystarczy kliknąć klasę prawym klawiszem myszy i wybrać opcję *Add/Property*. Istnieje również możliwość utworzenia związków między klasami poprzez wykorzystanie kontrolek `Association` i `Inheritance`. W niniejszej książce nie będziemy używać wymienionych kontrolek, choć w bardziej zaawansowanych sytuacjach są one niezwykle użyteczne.

Gdy narzędzie Object Relational Designer jest uruchomione, można rozpocząć budowanie własnego modelu obiektowego na podstawie bazy danych. Należy wyświetlić okno *Server*

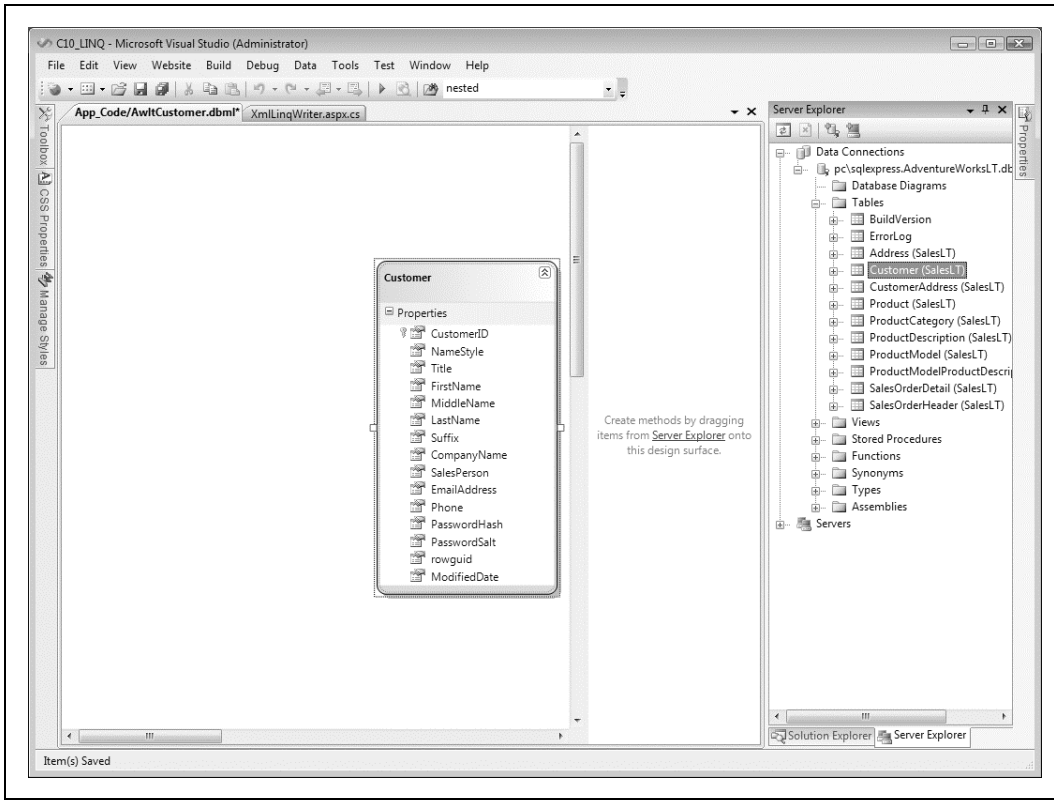


Rysunek 10.16. Puste okno narzędzia Object Relational Designer

Explorer i rozwinąć połączenie z bazą danych AdventureWorksLT — wystarczy po prostu kliknąć znak plus obok nazwy bazy danych. Następnie, aby rozwinąć listę tabel, trzeba kliknąć znak plus obok *Tables*. Kolejny krok to kliknięcie tabeli `Customer` i przeciągnięcie jej z okna *Server Explorer* do lewego panelu narzędzia Object Relational Designer. Ekran powinien wyglądać tak, jak pokazano na rysunku 10.17.

Do narzędzia Object Relational Designer trzeba przeciągnąć także tabele `Address` i `Customer` ↪`Address`. Po ustawieniu tabel w żądany sposób będzie można zauważyć związki między tabelami oznaczone w dużej mierze w taki sam sposób jak w narzędziu SQL Server Management Studio (SSMS). Jak pokazano na rysunku 10.18, strzałki łączą dwie tabele, a grot strzałki wskazuje tabelę zawierającą pole klucza zewnętrznego. Ponieważ baza danych definiuje związki zachodzące między tymi tabelami, wymienione związki zostają odzwierciedlone w wizualnym modelu danych. Co ważniejsze, związki te są teraz także odzwierciedlone w klasach utworzonych przez narzędzie.

I to na tyle. Pakiet VS2008 generuje rzeczywisty kod warstwy obiektowo-relacyjnej po przeciągnięciu lub usunięciu tabel z narzędzia Object Relational Designer. Jeżeli w bazie danych znajdują się procedury składowane, których programista chce również używać w LINQ (na przykład w celu zapisywania danych klientów z powrotem w bazie danych), należy je przeciągnąć na prawy panel narzędzia.



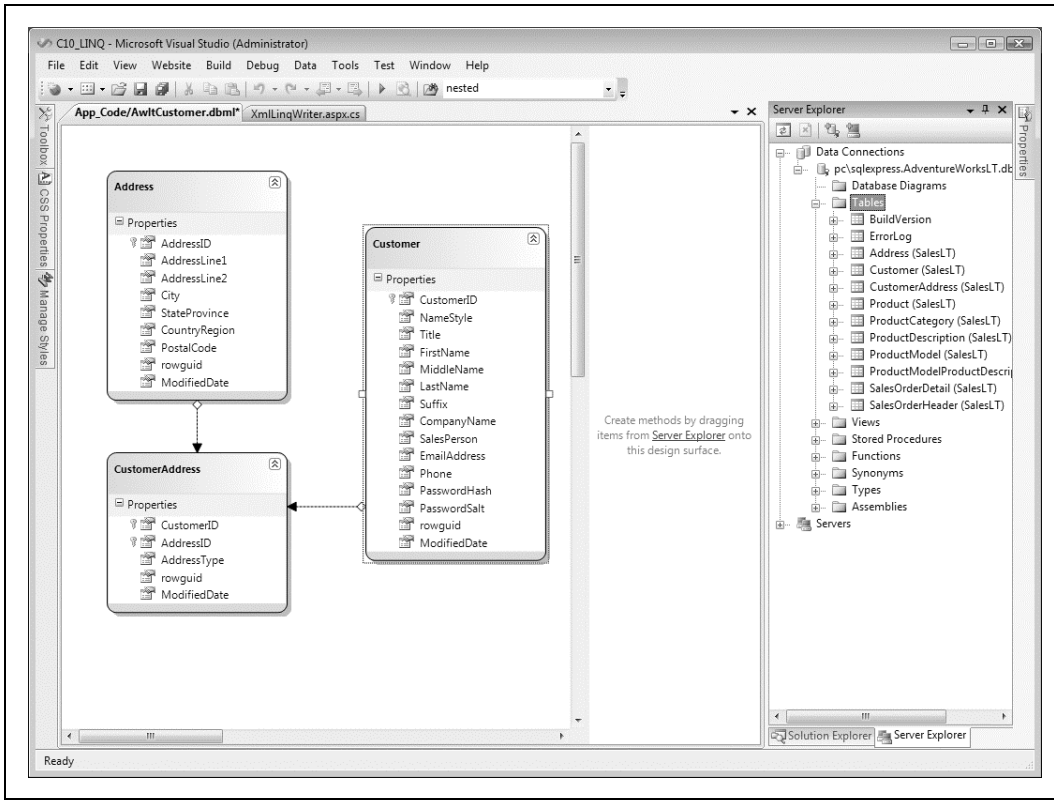
Rysunek 10.17. Tabela Customer wyświetlana przez narzędzie Object Relational Designer

W tle okno *Solution Explorer* daje programiście dostęp do rzeczywistego kodu wygenerowanego przez narzędzie Object Relational Designer (zobacz rysunek 10.19).

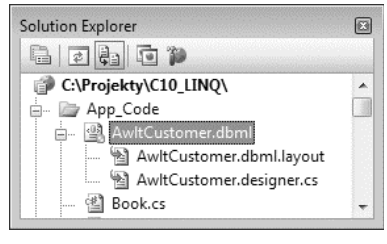
Narzędzie Object Relational Designer generuje trzy pliki:

- *AwltCustomer.dbml* to plik XML opisujący tabele przeciągnięte na obszar roboczy narzędzia, treść tabel oraz występujące między nimi związki.
- *AwltCustomer.dbml.layout* to plik śledzący wizualne położenie elementów oraz inne aspekty każdej tabeli znajdującej się w obszarze roboczym narzędzia.
- *AwltCustomer.designer.cs* to plik zawierający rzeczywiste klasy `Table` i `DataContext`. Oprócz kodu szkieletowego plik definiuje trzy podklasy dla klasy `AwltCustomerDatacontext`, po jednej dla każdej tabeli. W każdej podklasie dla każdej kolumny tabeli znajduje się właściwość publiczna. Można sprawdzić, że każda właściwość ma taki sam rodzaj danych jak odpowiadająca jej kolumna w tabeli. W rzeczywistości struktura jest przedstawiona niemal w taki sam sposób, jaki pokazano na listingu 10.12.

Klasa `AwltCustomersDataContext` zapewnia pomost między LINQ i bazą danych, więc pozostało jedynie utworzenie stron internetowych wykorzystujących tę klasę.



Rysunek 10.18. Związki między tabelami przedstawione graficznie w narzędziu Object Relational Designer



Rysunek 10.19. Kod wygenerowany przez narzędzie Object Relational Designer

Ręczne wykonywanie zapytań do obiektów DataContext

Po zbudowaniu obiektu `DataContext` można wydawać względem niego zapytania za pomocą poleceń LINQ w ten sam sposób, jaki stosowano do obiektów znajdujących się w pamięci lub dokumentów XML. Jedyna różnica polega na tym, że podczas zwrotu wyników zapytania na stronę połączenie z bazą danych zostaje ponownie otworzone i zamknięte. Rozsądnym rozwiązaniem będzie upewnienie się, że połączeniem można rozporządzać w obiekcie `Data` ↪`Context` za pomocą polecenia `using`, jak przedstawiono na listingu 10.13. Wymieniony listing przedstawia plik ukrytego kodu strony wykonującej proste zapytanie LINQ to SQL.

Listing 10.13. Plik ukrytego kodu SimpleSqlQuery.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.UI;
public partial class SimpleSqlQuery : Page
{
   protected void Page_Load(object sender, EventArgs e)
   {
      using (AwltCustomersDataContext db = new AwltCustomersDataContext())
      {
         var firstFiveCustomers =
            from customer in db.Customers.Take(5)
            select customer;
         foreach (var customer in firstFiveCustomers)
         {
            lblCustomers.Text += String.Format("<p>{0} {1}</p>",
               customer.FirstName, customer.LastName);
         }
      }
   }
}
```

Podobnie jak obiekt `XElement` wczytujący najpierw dokument, egzemplarz obiektu `DataCon` ➥`text` oferuje drogę do znajdujących się w pamięci obiektów przedstawiających bazę danych. W tym egzemplarzu obiekt ma trzy metody — `Customers`, `CustomerAddress` i `Addresses`, z których każda reprezentuje treść odpowiedniej tabeli.

Łączenie obiektów DataContext z innymi rodzajami danych

W poprzednim przykładzie wykonywano zapytanie do tabeli `Customer` w celu pobrania imienia i nazwiska pierwszych pięciu klientów z bazy danych. Jednak podobnie jak na listingu 10.10, na którym przedstawiono zapytanie łączące dokument XML z obiektem znajdującym się w pamięci, dzięki warstwie `DataContext` możliwe staje się połączenie danych bazy danych SQL z innym źródłem danych. W bieżącym przykładzie rozbudujemy przykład zaprezentowany na listingu 10.10. Łączymy obiekt znajdujący się w pamięci i przechowujący informacje o książce z tabelą SQL zawierającą informacje o autorze (OK, tabela nosi nazwę *customers*, ale…). Połączenie odbywa się za pomocą dokumentu XML wskazującego, kto napisał daną książkę.

W witrynie *C10_LINQ* należy utworzyć kopię strony internetowej *XmlToMemoryJoin.aspx* i zmienić jej nazwę na *ThreeSourceJoin.aspx*. Strona z treścią pozostaje taka sama, ale w pliku ukrytego kodu trzeba zaimplementować obiekt `DataContext`. Na listingu 10.14 przedstawiono pełny kod pliku ukrytego kodu *ThreeSourceJoin.aspx.cs*, nowe lub zmodyfikowane wiersze zostały pogrubione.

Listing 10.14. Plik ukrytego kodu ThreeSourceJoin.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.UI;
using System.Xml.Linq;
public partial class ThreeSourceJoin : Page
{
   protected void Page_Load(object sender, EventArgs e)
   {
```

```
        List<Book> bookList = Book.GetBookList();
        XElement doc =
           XElement.Load(Request.ApplicationPath + "\\authors.xml");
        using (AwltCustomersDataContext db = new AwltCustomersDataContext())
        {
           var authorsByBooks =
              from book in doc.DescendantsAndSelf("book")
              join bookInfo in bookList on
                 book.Attribute("isbn").Value equals bookInfo.ISBN
              join authorInfo in db.Customers on
                 book.Parent.Attribute("id").Value equals
                    authorInfo.CustomerID.ToString()
              let authorName = authorInfo.FirstName + " "
                 + authorInfo.LastName
              orderby bookInfo.Title
              group new { Name = authorName}
              by bookInfo.Title
                 into groupedAuthors
                 select new
                 {
                    Title = groupedAuthors.Key,
                    Authors = groupedAuthors
                 };
           foreach (var book in authorsByBooks)
           {
              lblBooks.Text += String.Format("<h2>{0}</h2>", book.Title);
              foreach (var author in book.Authors)
              {
                 lblBooks.Text += String.Format("Autor: {0} <br />",
                    author.Name);
              }
           }
        }
     }
  }
```

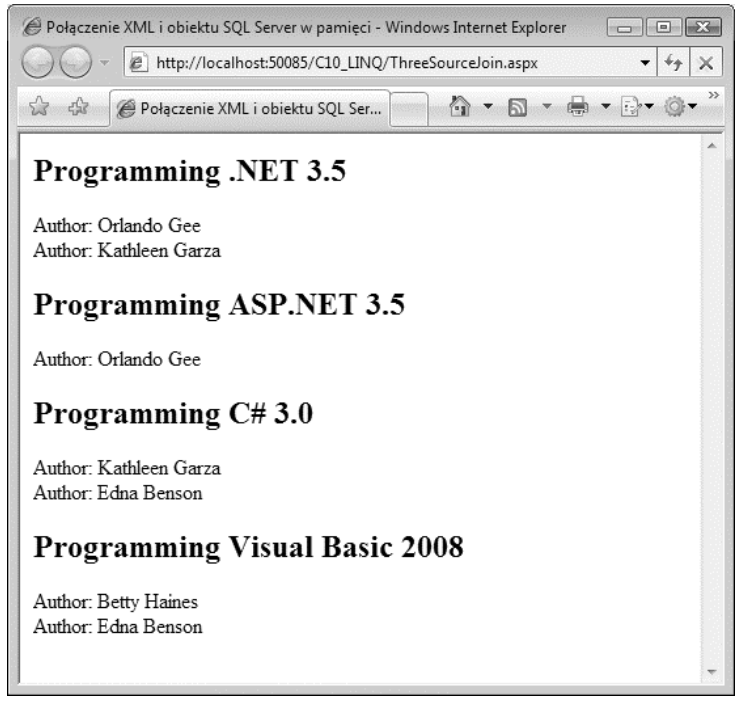
Po zapisaniu i uruchomieniu strony będzie można zauważyć, że nazwiska klientów pobrane z bazy danych i tytuły książek z obiektu `List` są prawidłowo połączone za pomocą dokumentu XML, jak pokazano na rysunku 10.20.

Warto zwrócić uwagę, że w połączeniu między obiektem `DataContext` i dokumentem XML następuje wywołanie metody `ToString()` względem `authorInfo.CustomerID`, ponieważ `Data` ↪`Context` prawidłowo określa typ jako liczbę całkowitą. Ten typ odpowiada typowi zastosowanemu w bazie danych. Jednak właściwość `Value` obiektu `XAttribute` zwraca ciąg tekstowy, a klauzula LINQ `join` nie przeprowadza rzutowania, konieczne jest zatem wywołanie metody `ToString()`, aby możliwe było połączenie dwóch ciągów tekstowych.

```
join authorInfo in db.Customers on
   book.Parent.Attribute("id").Value equals
   authorInfo.CustomerID.ToString()
```

Zapis w bazie danych za pomocą LINQ

Dodawanie, uaktualnianie lub usuwanie danych z bazy danych również można przeprowadzić za pomocą obiektu `DataContext`. Jednak w przeciwieństwie do zapisywania nowego dokumentu XML wprowadzanie zmian w bazie danych w ogóle nie musi angażować zapytań LINQ. Zamiast tego procedurą dodawania nowych danych jest utworzenie w tabeli, w której mają zostać umieszczone dane nowego obiektu przedstawiającego te dane, i ustawienie ich



Rysunek 10.20. Strona ThreeSourceJoin.aspx w działaniu

właściwości. Trzeba też przygotować dane do wstawienia poprzez wywołanie `InsertOnSubmit` względem obiektu w tabeli, do której będą dodane dane, a następnie wywołać metodę `SubmitChanges()` względem obiektu `DataContext`. Przykładowo:

```
using (AwltCustomersDataContext db = new AwltCustomersDataContext())
{
   Address addr = new Address();
   addr.AddressLine1 = "1c Sharp Way";
   addr.City = "Seattle";
   addr.PostalCode = "98011";
   addr.StateProvince = "Washington";
   addr.CountryRegion = "United States";
   addr.ModifiedDate = DateTime.Today;
   addr.rowguid = Guid.NewGuid();
   db.Addresses.InsertOnSubmit(addr);
   db.SubmitChanges();
}
```

Podobnie uaktualnienie rekordu tabeli wymaga pobrania obiektu przedstawiającego ten rekord, ustawienia dla właściwości obiektu nowych wartości, a następnie ponownego wywołania metody `SubmitChanges()`:

```
using (AwltCustomersDataContext db = new AwltCustomersDataContext())
{
   Address addr = db.Addresses.Single(
      a => (a.AddressLine1 == "1c Sharp Way" && a.City == "Seattle"));
   addr.AddressLine1 = "12b Pointy Street";
   db.SubmitChanges();
}
```

Jeżeli obiekt był oddzielony od `Context` — na przykład z powodu przetwarzania przez usługę sieciową lub przechowywania jako oddzielnej, pośredniej warstwy biznesowej — można wywołać metodę `Attach` w jego odpowiednim zbiorze w `Context`, a następnie metodę `SubmitChanges()`:

```
using (AwltCustomersDataContext db = new AwltCustomersDataContext())
{
   db.Addresses.Attach(updatedAddress, true);
   db.SubmitChanges();
}
```

Warto zwrócić uwagę, że parametr Boolean w wywołaniu `Attach` wskazuje, czy obiekt został zmodyfikowany od chwili oddzielenia od `Context`.

Wreszcie — usunięcie rekordu tabeli wymaga pobrania obiektu przedstawiającego ten rekord, wywołania metody `DeleteOnSubmit` względem obiektu reprezentującego tabelę, z której będzie usunięty rekord, a następnie wywołania metody `SubmitChanges()`:

```
using (AwltCustomersDataContext db = new AwltCustomersDataContext())
{
   Address addr = db.Addresses.Single(
      a => (a.AddressLine1 == "12b Pointy Street "
            && a.City == "Seattle"));
   db.Addresses.DeleteOnSubmit(addr);
   db.SubmitChanges();
}
```

Jeżeli usunięcie rekordu nie spowoduje problemów — ponieważ nie będzie niszczyło związku między dwiema tabelami — rekord zostanie usunięty.

Spójność danych

Jednym z wyzwań dotyczących baz danych jest zapewnienie spójności danych. Aby zrozumieć, jak to działa, w pierwszej kolejności należy poznać koncepcję *normalizacji*, która wraz z innymi czynnikami wskazuje, że dane w relacyjnej bazie danych nie powinny być zbędnie powielane.

Przykładowo, jeżeli mamy bazę danych zawierającą informacje o klientach i ich zamówieniach, zamiast powielać w każdym zamówieniu informacje o kliencie (imię i nazwisko klienta, adres, dane kontaktowe itd.), należy utworzyć rekord klienta i przypisać mu unikalny identyfikator `KlientID`. W efekcie każde zamówienie będzie zawierało pole `KlientID` wskazujące klienta, który złożył dane zamówienie.

Tego rodzaju podejście ma wiele zalet. Jedną z nich jest to, że w przypadku modyfikacji danych klienta trzeba zmodyfikować tylko jeden rekord klienta, a nie każdy rekord złożonego przez niego zamówienia.

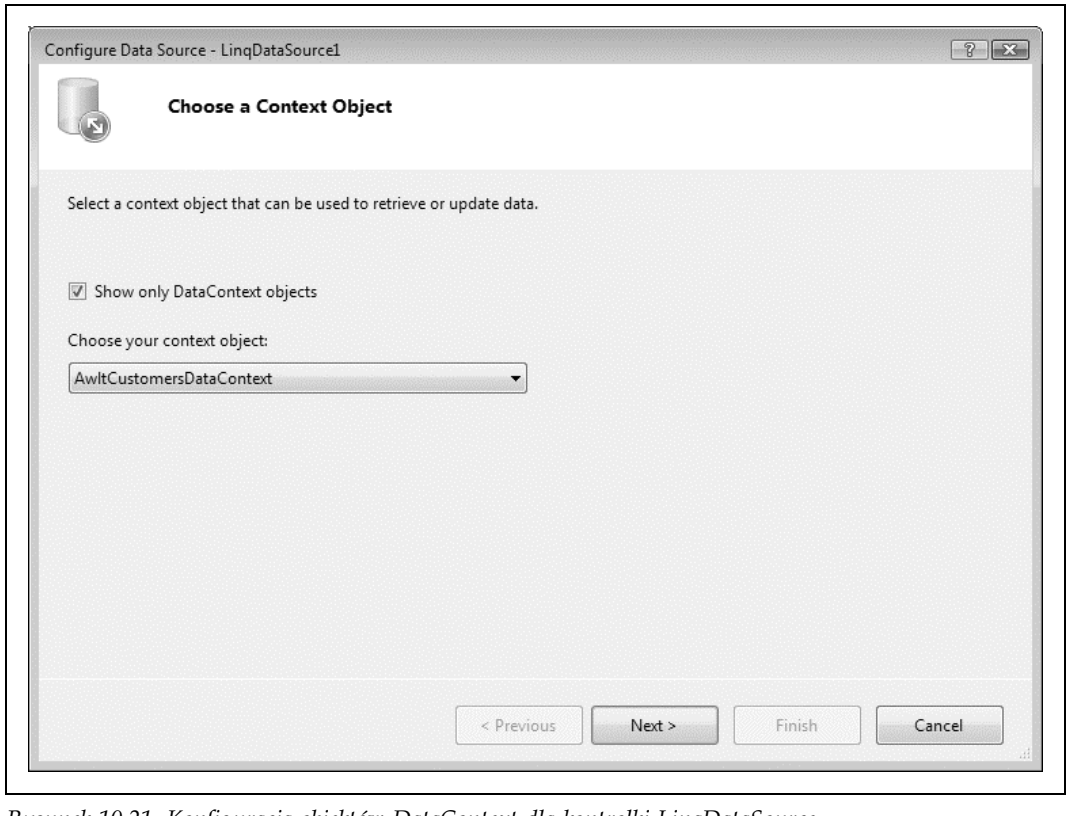
Jednak dane będą niespójne jeżeli identyfikator `KlientID` w zamówieniu nie będzie w ogóle odnosił się do klienta (lub gorzej, jeśli wskaże niewłaściwego klienta!). Aby uniknąć takich sytuacji, administratorzy baz danych lubią, gdy bazy danych wymuszają stosowanie reguł spójności. Przykładem może tutaj być brak możliwości usunięcia rekordu klienta aż do chwili, gdy wcześniej nie zostaną usunięte wszystkie złożone przez niego zamówienia. (To oznacza niepozostawianie „osieroconych" zamówień, które nie są przypisane żadnemu klientowi). Inna reguła to nieużywanie ponownie tych samych identyfikatorów `KlientID`.

Wprowadzenie obiektu LinqDataSource

W celu pobrania, uaktualnienia, dodania lub usunięcia danych z bazy danych za pomocą LINQ trzeba utworzyć całkiem dużą ilość kodu, więc warto nieco uprościć sobie pracę. W pakiecie VS2008 firma Microsoft wprowadziła obiekt `LinqDataSource` hermetyzujący cały kod w pojedynczym obiekcie, który przypomina obiekty przedstawione wcześniej w rozdziale 7. Jedyna różnica polega na tym, że zamiast poleceń SQL, definiujących dla bazy danych instrukcje pobrania, wstawienia, usunięcia i uaktualnienia danych, obiekt `LinqDataSource` oczekuje po prostu zapytań LINQ `select`. Mogą być wykonywane z użyciem dowolnego dostawcy LINQ (LINQ to XML, LINQ to Objects, LINQ to SQL itd.). Obiekt `LinqDataSource` automatycznie określi, kiedy wstawić, usunąć i uaktualnić dane w tym magazynie danych, o ile używany dostawca obsługuje te polecenia.

Aby zademonstrować wymienione możliwości, do witryny *C10_LINQ* należy dodać nową stronę internetową o nazwie *LinqDS.aspx*. W widoku *Design view* trzeba na stronie umieścić kontrolkę `LinqDataSource`. Po wybraniu opcji *Configure Data Source* z menu tagu inteligentnego kontrolki zostanie wyświetlony kreator Configure Data Source, który w działaniu jest podobny do kreatora stosowanego przez kontrolkę `SqlDataSource`.

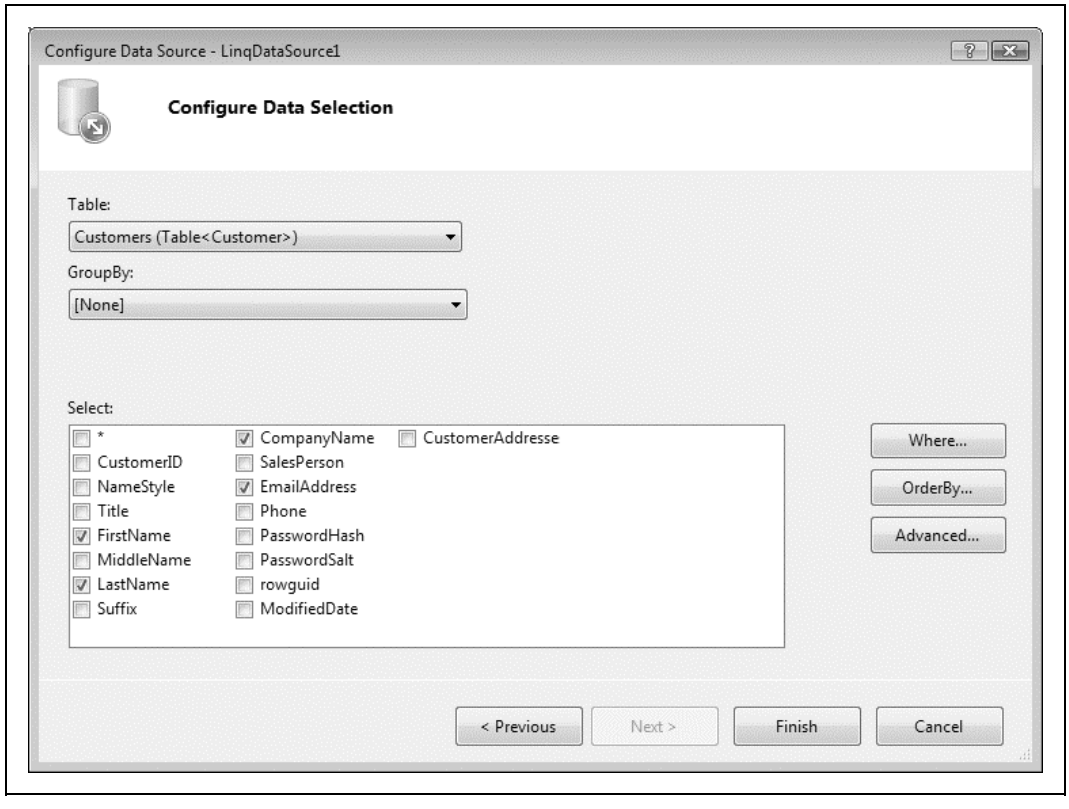
Pierwszym krokiem w kreatorze jest określenie obiektu `DataContext` przedstawiającego tabele bazy danych oraz procedury składowane, które będą używane (zobacz rysunek 10.21).



Rysunek 10.21. Konfiguracja obiektów DataContext dla kontrolki LinqDataSource

Wszystkie dostępne obiekty `DataContext` są wymienione na rozwijanej liście, ale w omawianym przykładzie jest tylko jeden — utworzony wcześniej `AwltCustomersDataContext`, więc należy kliknąć przycisk *Next*. Po odznaczeniu pola wyboru „Show only DataContext objects" możliwy będzie również wybór kontekstów SQL innych niż LINQ do wykonywania zapytań.

Kolejny krok to wskazanie pól tabeli w bazie danych, które będą wyświetlane (zobacz rysunek 10.22).



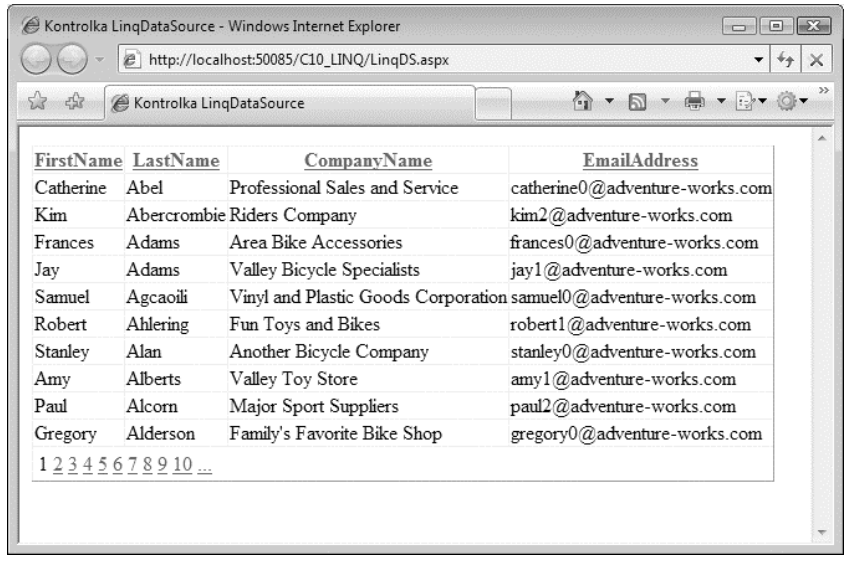
Rysunek 10.22. Wybór wyświetlanych pól

Podobnie jak miało to miejsce w rozdziale 7., należy wybrać tabelę `Customer` wraz z polami `FirstName`, `LastName`, `CompanyName` i `EmailAddress`, a następnie kliknąć przycisk *Finish*, by zakończyć tym samym pracę kreatora. Warto też zwrócić uwagę na możliwość pogrupowania danych zwracanych przez zapytanie LINQ, choć ten temat nie zostanie tutaj omówiony.

Jeśli przejrzeć kod wygenerowany przez kreatora, można dostrzec, że tylko klauzula `select` zapytania LINQ jest wyświetlana przez właściwość `Select` kontrolki `DataSource`. Pozostałe pozostają ukryte przed ciekawskimi.

Następnie trzeba przejść do widoku *Design view* i umieścić na stronie kontrolkę `GridView`, a następnie przy użyciu jej menu tagu inteligentnego ustawić kontrolkę `LinqDataSource` jako źródło danych kontrolki `GridView`. Kontrolka `GridView` zostanie natychmiast odświeżona w widoku *Design view* i wyświetli kolumny zdefiniowane w źródle danych.

Po wyświetleniu menu tagu inteligentnego należy zaznaczyć pola wyboru *Enable Paging* i *Enable Sorting*, a następnie uruchomić stronę. Na ekranie zostanie wyświetlona strona pokazana na rysunku 10.23 wraz z w pełni zaimplementowanym stronicowaniem i sortowaniem. Jedyny wyjątek polega na tym, że strona bazuje na kontrolce `LinqDataSource`, a nie `SqlDataSource`.



FirstName	LastName	CompanyName	EmailAddress
Catherine	Abel	Professional Sales and Service	catherine0@adventure-works.com
Kim	Abercrombie	Riders Company	kim2@adventure-works.com
Frances	Adams	Area Bike Accessories	frances0@adventure-works.com
Jay	Adams	Valley Bicycle Specialists	jay1@adventure-works.com
Samuel	Agcaoili	Vinyl and Plastic Goods Corporation	samuel0@adventure-works.com
Robert	Ahlering	Fun Toys and Bikes	robert1@adventure-works.com
Stanley	Alan	Another Bicycle Company	stanley0@adventure-works.com
Amy	Alberts	Valley Toy Store	amy1@adventure-works.com
Paul	Alcorn	Major Sport Suppliers	paul2@adventure-works.com
Gregory	Alderson	Family's Favorite Bike Shop	gregory0@adventure-works.com

1 2 3 4 5 6 7 8 9 10 ...

Rysunek 10.23. Kontrolka LinqDataSource w działaniu

Jaka jest więc różnica między dwoma wymienionymi źródłami danych, skoro wynik końcowy w obu przypadkach pozostaje identyczny? Jak już wcześniej wspomniano, LINQ to język pozwalający na konstruowanie zapytań do bazy danych bezpośrednio w wybranym języku programowania zamiast z użyciem SQL. W rozdziale 7. przedstawiono kod wygenerowany przez kontrolkę `SqlDataSource`. Kod zawierał atrybuty `ConnectionString` i `SelectCommand`, w drugim z wymienionych umieszczono polecenie SQL:

```
SELECT FirstName, LastName, CompanyName, EmailAddress
FROM SalesLT.Customer
```

Jeżeli strona *LinqDS.aspx* zostanie wyświetlona w widoku *Source view*, to zobaczymy poniższy kod kontrolki `LinqDataSource`:

```
<asp:LinqDataSource ID="LinqDataSource1" runat="server"
   ContextTypeName="AwltCustomersDataContext" OrderBy="LastName"
   Select="new (FirstName, LastName, CompanyName, EmailAddress)"
   TableName="Customers">
</asp:LinqDataSource>
```

Zamiast atrybutu `ConnectionString` wskazującego bazę danych kontrolka ma atrybut `Context`
`↪TypeName` określający klasę `DataContext` utworzoną za pomocą narzędzia Object Relational Designer. To będzie klasa `DataContext` przechowująca ciąg tekstowy połączenia.

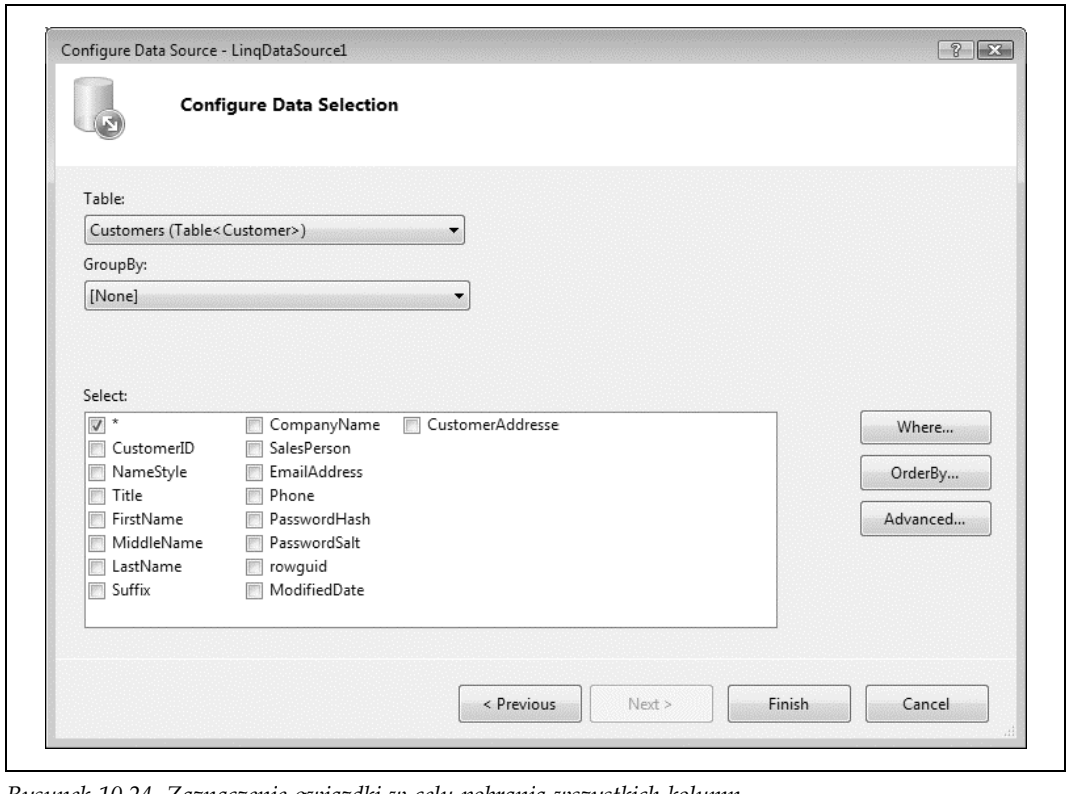
Ponadto, zamiast atrybutu `SelectCommand` z poleceniem SQL, w kodzie znajduje się atrybut `Select` wraz z poleceniem LINQ wskazującym właściwości tabeli określonej przez atrybut `TableName`. Poza tym są tutaj również atrybuty `Where`, `OrderBy`, `GroupBy` i `OrderGroupsBy` odpowiadające właściwym fragmentom zapytania LINQ.

Kontrolka `LinqDataSource` może również działać z kontrolką `GridView`, co pozwala na łatwą edycję danych, o ile źródło danych zostanie skonfigurowane tak, aby zwracało wszystkie kolumny tabeli i nie przeprowadzało operacji `GroupBy`. (Warto przypomnieć sobie z wcześniejszego przykładu, że używanie `GroupBy` powoduje zmianę struktury wyników zwracanych przez zapytanie na postać, której nie można przypisać i dołączyć kontrolce `GridView`). Kontrolka `GridView` nie musi wyświetlać wszystkich kolumn, ale kontrolka `LinqDataSource` musi pobierać wszystkie.

Aby przetestować takie rozwiązanie, należy ponownie w narzędziu Object Relatinal Designer otworzyć plik *AwltCustomers.dbml*, usunąć z panelu tabele `Address` i `CustomerAddress`, a następnie zapisać plik.

Teraz dodajemy kolejną stronę o nazwie *LinqDsEdit.aspx*. W widoku *Source* lub *Design view* przeciągamy kontrolkę `LinqDataSource` na stronę oraz kontrolkę `GridView` z sekcji `Data` paska narzędziowego. W widoku *Design view* wyświetlamy menu tagu inteligentnego kontrolki `Linq ↪DataSource` i wybieramy opcję *Configure Data Source*. Podobnie jak wcześniej trzeba się upewnić o wyborze opcji `CustomersDataContext` i kliknąć przycisk *Next*.

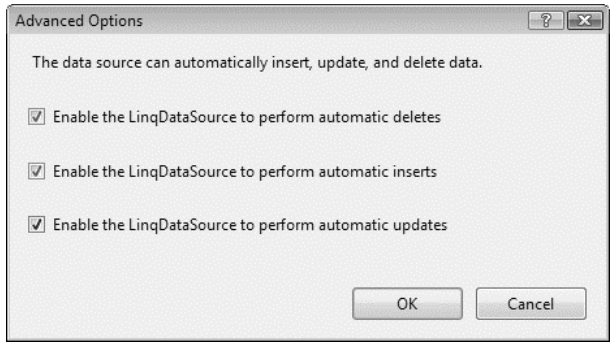
Domyślnie w pierwszej rozwijanej liście wybrana będzie jedyna dostępna tabela `Customer`, natomiast w części *Select* zaznaczona będzie pierwsza kolumna (zawierająca gwiazdkę, czyli wybór wszystkich pól), jak pokazano na rysunku 10.24.



Rysunek 10.24. Zaznaczenie gwiazdki w celu pobrania wszystkich kolumn

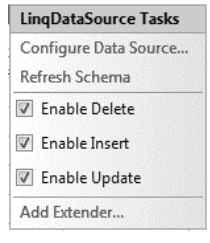
Jeżeli w oknie dialogowym nie będą wyświetlone żadne tabele lub kolumny, należy kliknąć przycisk *Cancel*, a następnie wybrać menu *Build/Build Website* w celu zbudowania witryny. Teraz można spróbować ponownie.

Kliknięcie przycisku *Advanced* powoduje wyświetlenie okna dialogowego z opcjami pokazanymi na rysunku 10.25.



Rysunek 10.25. Opcje zaawansowane zapytania

Należy zaznaczyć wszystkie trzy opcje, następnie kliknąć *OK* i *Finish*. Menu tagu inteligentnego kontrolki wyświetli wybrane zaznaczone pola wyboru (będą również zaznaczone) służące do usuwania, wstawiania i uaktualniania danych (zobacz rysunek 10.26).



Rysunek 10.26. Umożliwienie usuwania, wstawiania i uaktualniania danych w kontrolce LinqDataSource

Po przejściu do widoku *Source view* i spojrzeniu na deklarację kontrolki `LinqDataSource` zobaczymy następujący kod źródłowy:

```
<asp:LinqDataSource ID="LinqDataSource1" runat="server"
   ContextTypeName="AwltCustomersDataContext" EnableDelete="True"
   EnableInsert="True" EnableUpdate="True" TableName="Customers">
</asp:LinqDataSource>
```

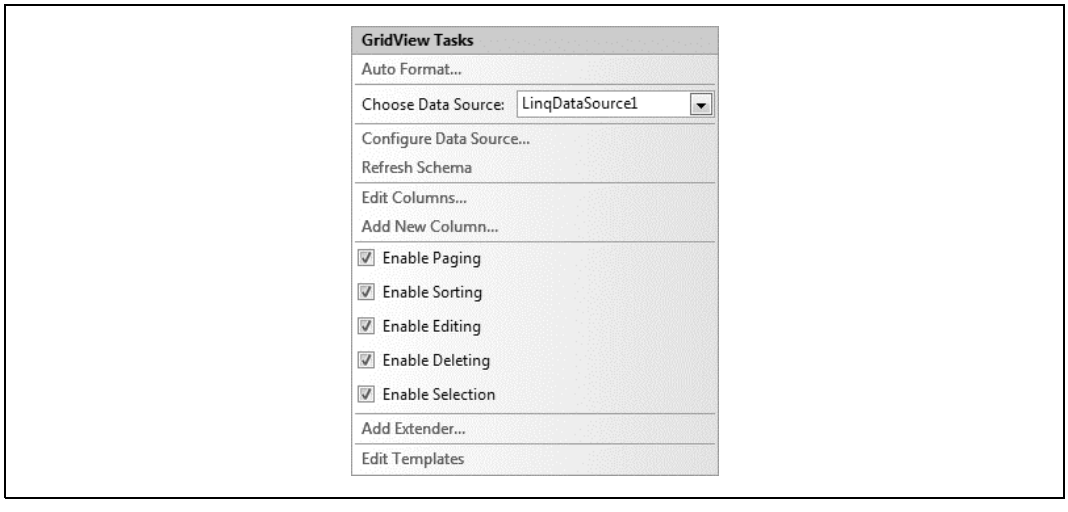
Po porównaniu z wcześniejszym kodem źródłowym kontrolki `LinqDataSource` można dostrzec, że nie tylko ma atrybuty umożliwiające usuwanie, wstawianie i uaktualnianie danych, ale równocześnie nie zawiera atrybutu `Select` zwracającego określone kolumny. Dlatego też z bazy danych pobrane będą wszystkie kolumny.

Teraz trzeba kliknąć tag inteligentny kontrolki `GridView`. Jako źródło danych należy wskazać `LinqDataSource1`. Kontrolka `GridView` będzie natychmiast odświeżona i wyświetli każdą kolumnę tabeli, czyli znacznie więcej, niż chcemy wyświetlać. Jak Czytelnik pamięta z rozdziału 9., istnieją dwa podstawowe sposoby usuwania z kontrolki `GridView` niechcianych pól:

- Sposób graficzny: w menu tagu inteligentnego trzeba kliknąć opcję *Edit Columns*, by wyświetlić okno edytora Fields. W edytorze można usunąć niechciane pola poprzez ich zaznaczanie, pojedynczo z listy w lewym dolnym rogu okna dialogowego, a następnie kliknięcie czerwonego przycisku *X*.
- Sposób tekstowy: należy po prostu usunąć niechciane deklaracje `BoundField` z elementu `Columns`.

Niechciane pola należy usunąć w sposób tekstowy, czyli poprzez usunięcie niechcianych deklaracji `BoundField`.

Teraz ostatni krok: trzeba powrócić do tagu inteligentnego kontrolki `GridView`. Menu będzie zawierało dwa nowe pola wyboru, które spotkaliśmy już wcześniej, czyli *Enable Editing* i *Enable Deleting*. Należy je zaznaczyć, tak jak pokazano na rysunku 10.27.



Rysunek 10.27. Możliwość usuwania, wstawiania i uaktualniania danych w kontrolce GridView

Po zapisaniu i uruchomieniu strony można zobaczyć, że z wyjątkiem problemów dotyczących spójności możliwe będzie edytowanie, wstawianie i usuwanie danych z tabeli `Customer`.

Więcej informacji na temat API LINQ i dostawców LINQ dostarczanych jako część platformy .NET 3.5 i .NET 3.5 Service Pack 1 można znaleźć na stronie domowej projektu LINQ pod adresem *http://msdn.microsoft.com/en-us/netframework/aa904594.aspx* oraz w bazie wiedzy na stronie *http://msdn.microsoft.com/en-us/library/bb397926.aspx*. Rozszerzona lista książek poświęconych LINQ znajduje się na stronie *http://blogs.msdn.com/charlie/archive/2008/02/17/ linq-books.aspx*.