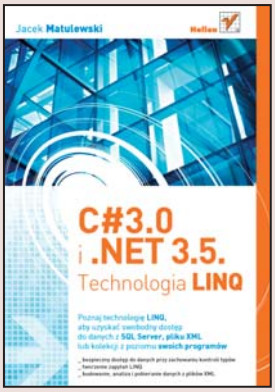
C# 3.0 i .NET 3.5. Technologia LINQ

Autor: Jacek Matulewski
ISBN: 83-246-1447-8
Format: 158x235, stron: 88



Poznaj technologię LINQ

aby uzyskać swobodny dostęp do danych z SQL Server, pliku XML lub kolekcji z poziomu swoich programów

- Bezpieczny dostęp do danych przy zachowaniu kontroli typów
- Tworzenie zapytań LINQ
- Budowanie, analiza i pobieranie danych z plików XML

Zastosowanie technologii LINQ (ang. Language-Integrated Query, zintegrowany język zapytań) w zasadniczy sposób upraszcza projektowanie aplikacji bazodanowych. Zapytanie LINQ zwraca kolekcję z przestrzeni nazw typów ogólnych. Kolekcja ta może być modyfikowana, a następnie zwrócona do źródła. Dzięki temu zachowywana jest pełna kontrola typów danych i ich konwersji w poszczególnych mechanizmach pośredniczących w pobieraniu danych.

Książka „C# 3.0 i .NET 3.5. Technologia LINQ" prezentuje prosty sposób korzystania ze zintegrowanego języka zapytań. Dzięki temu podręcznikowi nauczysz się pobierać dane z różnego rodzaju źródeł; tworzyć pliki XML w nowy, bardziej intuicyjny sposób; stosować składowane rozszerzenia ? nowego typu metody (oraz odpowiadające im operatory), zdefiniowane w najnowszej wersji języka C#. Ponadto dowiesz się, jak tworzyć własne źródła danych LINQ. Krótko mówiąc, z książką „C# 3.0 i .NET 3.5. Technologia LINQ" nawet jako średnio zaawansowany programista poradzisz sobie ze stworzeniem bardzo skomplikowanych aplikacji bazodanowych.

- Wyrażenia Lambda
- Nowe operatory LINQ
- Zapytania LINQ
- LINQ to DataSet
- Aktualizacja danych w bazie
- Korzystanie z rozszerzeń
- Kontrolka LinqDataSource
- Tworzenie i modyfikacja pliku XML
- Tworzenie źródeł danych LINQ

Ta książka wskaże Ci **prosty sposób tworzenia** doskonałych i nowoczesnych aplikacji bazodanowych

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

Spis treści

Rozdział 3.

LINQ i SQL Server

Wspominając o technologii LINQ, mówi się zazwyczaj o zanurzeniu języka SQL w języku C#. W przypadku *LINQ to SQL* zanurzenie to można rozumieć niemal dosłownie — zapytanie LINQ jest w tym przypadku tłumaczone bezpośrednio na zapytanie SQL wysyłane do bazy danych.

Zacznijmy od prostej sytuacji: połączmy się z bazą danych *Telefony.mdf* (zob. rysunek 2.3 z poprzedniego rozdziału) i pobierzmy z niej listę osób pełnoletnich. Wykorzystamy do tego obiekt klasy `DataContext`, która jak na razie współpracuje tylko z bazami SQL Server (nie dotyczy to jednak lokalnych baz SQL Server Compact Edition). Klasa `DataContext` jest główną bramą do technologii *LINQ to SQL*. Brama ta wymaga jednak strażnika. Jego rolę przejmuje klasa encji, tj. klasa C#, która zdefiniuje typ każdego rekordu (encji) tabeli bazy danych. Dzięki tej klasie możliwe jest zbudowanie pomostu między tabelą pobieraną za pomocą zapytania SQL a kolekcją zwracaną przez zapytanie LINQ, tj. między typem encji a typem elementu kolekcji. Dzięki tej klasie w zapytaniu LINQ możemy używać nazw pól, które obecne są w tabeli, wręcz identyfikować kolekcję z tabelą.

Zanim przejdziemy do konkretów, chciałbym jeszcze zwrócić uwagę Czytelnika na jeden istotny fakt. W *LINQ to Object*, które poznaliśmy w pierwszym rozdziale, źródłem danych była kolekcja istniejąca w pamięci i w pełni dostępna z poziomu programu. Nie było zatem konieczności odwoływania do zasobów zewnętrznych (np. plików bazy danych). Dzięki temu cały kod aplikacji mógł być skompilowany do kodu pośredniego. W przypadku *LINQ to SQL*, którym zajmiemy się w tym rozdziale, lub *LINQ to XML*, omówionego w kolejnym rozdziale, taka pełna kompilacja nie jest możliwa. Analiza danych pobranych ze źródła danych może odbyć się dopiero w trakcie działania programu. Realizowane jest to za pomocą drzewa zapytań, które poznamy dokładniej w ostatnim rozdziale, przy okazji projektowania własnego źródła danych LINQ.

Klasa encji

Klasa encji (ang. *entity class*) to zwykła klasa C#, w której za pomocą atrybutów powiązane są pola klasy z polami tabeli (kolumnami). Mamy więc do czynienia z modelowaniem zawartości relacyjnej bazy danych w typowanych klasach języka, w którym

przygotowujemy program. W tym kontekście używany jest angielski termin *strongly typed*, który podkreśla izomorficzność relacji klasy encji i struktury tabeli. Ów związek jest niezwykle istotny — przede wszystkim umożliwia kontrolę typów, która w pewnym sensie rozciąga się na połączenie z bazą danych. Dzięki tej relacji programista może w pewnym stopniu utożsamiać tę klasę i reprezentowaną przez nią tabelę. To pozwala również przygotowywać zapytania LINQ, korzystając z nazw pól tabeli — jego tłumaczenie na zapytanie SQL jest dzięki temu szczególnie proste, a jednocześnie w pełni zachowana jest kontrola typów. Domyślne wiązanie realizowane jest na podstawie nazw obu pól. Możliwa jest jednak zmiana tego domyślnego sposobu wiązania oraz zmiana własności poszczególnych pól.

Przed całą klasą znaleźć się powinien atrybut `Table` z przestrzeni nazw `System.Data.Linq.Mapping`, w którym wskazujemy nazwę tabeli z bazy danych[1]. Natomiast przed polami odpowiadającymi kolumnom w tabeli należy umieścić atrybut `Column` (z tej samej przestrzeni nazw), w którym możemy wskazać m.in. nazwę kolumny w tabeli (parametr `Name`), poinformować go, czy jest kluczem głównym (`IsPrimaryKey`) lub czy może przyjmować puste wartości (`CanBeNull`). Listing 3.1 zawiera przykład klasy encji, w której definiujemy typ rekordu z tabeli *ListaOsob* z bazy danych *Telefony.mdf* znanej z poprzedniego rozdziału.[2]

Listing 3.1. *Klasa encji związana z tabelą ListaOsob z bazy Telefony.mdf*

```
[Table(Name = "ListaOsob")]
public class Osoba
{
   [Column(Name = "Id", IsPrimaryKey = true)]
   public int Id;
   [Column(Name = "Imię", CanBeNull = false)]
   public string Imię;
   [Column(Name = "Nazwisko", CanBeNull = false)]
   public string Nazwisko;
   [Column]
   public int NumerTelefonu;
   [Column]
   public int Wiek;
}
```

W powyższym listingu wiązanie klasy ze strukturą bazy danych przeprowadzane zostaje na podstawie atrybutów dołączanych do definicji klasy. W terminologii Microsoft nazywane jest to mapowaniem opartym na atrybutach (ang. *attribute-based mapping*).

[1] Klasy z przestrzeni nazw `System.Data.Linq` i jej podprzestrzeni, m.in. z `System.Data.Linq.Mapping`, do której należy klasa `Table`, zdefiniowane są w osobnej bibliotece *System.Data.Linq.dll*, należącej do zbioru bibliotek platformy .NET 3.5. Domyślnie nie jest ona włączana do zbioru bibliotek projektu. Należy ją zatem dodać samodzielnie, korzystając z polecenia *Project*, *Add Reference*.... Omówione niżej narzędzie wizualnego projektowania klas encji zwolni nas z tego obowiązku.

[2] Tworząc tabelę w bazie SQL Server mogliśmy zezwolić, aby niektóre jej pola dopuszczały pustą wartość. Definiując klasę encji warto zadeklarować odpowiadające im pola klasy w taki sposób, aby dopuszczały przypisanie wartości `null`. W przypadku łańcuchów nie ma problemu – typ `String` jest typem referencyjnym i zawsze można mu przypisać wartość `null`. Inaczej wygląda to np. w przypadku typu `int`, który jest typem wartościowym. Należy wówczas w deklaracji pola wykorzystać typ parametryczny `Nullable<int>`.

Możliwe jest również podejście alternatywne, w którym mapowanie odbywa się na podstawie struktury zapisanej w pliku XML. Takie podejście nosi nazwę mapowania zewnętrznego i nie będę się nim tu zajmował. Po jego opis warto zajrzeć na stronę MSDN: *http://msdn2.microsoft.com/en-us/library/bb386907.aspx*.

LINQ to SQL daje dobry pretekst do wspomnienia o zagadnieniu modelowania danych. Jest to ciekawa gałąź informatyki, ale nie chciałbym snuć dygresji na jej temat. Osoby zainteresowane teorią przechowywania danych powinny w Google wpisać hasło `data modeling` lub `modelowanie danych`.

Jak wspomniałem wcześniej, klasa encji zwykle nie jest tworzona ręcznie. Zwykle do jej projektowania wykorzystywany jest edytor *O/R Designer*, który omówię w dalszej części tego rozdziału. Wydaje mi się jednak, że przy pierwszych próbach korzystania z technologii *LINQ to SQL* warto zrobić wszystko samodzielnie. Dotyczy to również powoływania instancji klasy `DataContext`, czym zajmiemy się już za chwilę.

Pobieranie danych

Zacznijmy od zdefiniowania pól przechowujących referencje do instancji klasy `Data ↪Context` i jej tabeli *ListaOsob*:

```
static string nazwaPliku = "Telefony.mdf";
static DataContext bazaDanychTelefony =
   ↪new DataContext(System.IO.Path.GetFullPath(nazwaPliku));
static Table<Osoba> listaOsob = bazaDanychTelefony.GetTable<Osoba>();
```

Tworzymy obiekt `DataContext` i pobieramy z niego referencję do tabeli (klasa `Table` parametryzowana klasą encji `Osoba`). Klasa `DataContext` znajduje się w przestrzeni nazw `System.Data.Linq`, należy więc uwzględnić ją w grupie poleceń `using`[3]. Ostatnia instrukcja tworzy pole o nazwie `listaOsob` typu `Table` parametryzowanego naszą klasą encji `Osoba`. Referencja ta będzie umożliwiać dostęp do danych pobranych z tabeli *ListaOsob*. Nazwa pobieranej tabeli wskazana została w atrybucie `Table`, którym poprzedziliśmy klasę encji. Dlatego jej nazwa nie pojawia się w powyższych definicjach.

Obiekt klasy `DataContext` reprezentuje bazę danych. Z kolei klasa encji zdefiniowana w listingu 3.1 dostarcza informacji o strukturze konkretnej tabeli (encji). Dzięki obu klasom możemy połączyć się z bazą danych i pobrać dane z interesującej nas tabeli. Prezentuje to listing 3.2.

Listing 3.2. *Pobieranie danych z tabeli w bazie SQL Server za pomocą LINQ to SQL*

```
private void button1_Click(object sender, EventArgs e)
{
   //pobieranie kolekcji
   var listaOsobPelnoletnich = from osoba in listaOsob
```

[3] Tu stosuje się również uwaga z poprzedniego przypisu. Ponadto konstruktor klasy `DataContext` wymaga, aby ścieżka do pliku podana w argumencie była bezwzględna (ang. *full path*).

```
                                where osoba.Wiek>=18
                                select osoba;

    //wyświetlanie pobranej kolekcji
    string s = "Lista osób pełnoletnich:\n";
    foreach (Osoba osoba in listaOsobPelnoletnich) s+=osoba.Imię+" "+osoba.Nazwisko
       ↪+" ("+osoba.Wiek+")\n";
    MessageBox.Show(s);
}
```

W zapytaniu LINQ (pierwsza instrukcja metody widocznej w listingu 3.2) pobiera z tabeli listę osób, które mają co najmniej 18 lat. Zwróćmy uwagę na podobieństwo tego zapytania do tego, które stosowaliśmy w przypadku *LINQ to Objects* w pierwszym rozdziale.

W listingu 3.2 klasa encji `Osoba` pojawia się tylko raz w pętli `foreach`. Co więcej, skoro jest jasne, że każdy rekord jest obiektem typu `Osoba`, a to wiadomo dzięki parametryzacji pola `listaOsob`, jawne wskazanie typu w tej pętli nie jest konieczne i z równym skutkiem moglibyśmy użyć słowa kluczowego `var` — kompilator sam wywnioskowałby, z jakim typem ma do czynienia.

Aktualizacja danych w bazie

Pobieranie danych, wizytówka LINQ, to zwykle pierwsza czynność wykonywana przez aplikacje. Technologia *LINQ to SQL* nie poprzestaje tylko na tym. Zmiany wprowadzone w pobranej zapytaniem LINQ kolekcji można w łatwy sposób przesłać z powrotem do bazy danych. Służy do tego metoda `SubmitChanges` obiektu `DataContext`. Tym samym wszelkie modyfikacje danych stają się bardzo naturalne: nie ma konieczności przygotowywania poleceń SQL, odpowiedzialnych za aktualizację tabel w bazie danych — wszystkie operacje wykonujemy na obiektach C#. Zachowana jest w ten sposób spójność programu, co pozwala na kontrolę typów i pełną weryfikację kodu już w trakcie kompilacji.

Modyfikacje istniejących rekordów

Pokażmy to na prostym przykładzie. Załóżmy, że mija Nowy Rok i z tej okazji zwiększamy wartość w polach *Wiek* wszystkich osób. Rzecz jasna wszystkich, poza kobietami, które są już pełnoletnie. Pobieramy zatem kolekcję osób, które są mężczyznami lub są niepełnoletnie. Spójnik „lub” użyty w poprzednim zdaniu rozumiany powinien być tak, jak zdefiniowany jest operator logiczny `OR` — powinniśmy uzyskać sumę mnogościową zbiorów osób niepełnoletnich i zbioru mężczyzn. Następnie zwiększamy wartość pola `Wiek` każdej z osób tak uzyskanego zbioru. I najmilsza rzecz: aby zapisać nowe wartości do pliku bazy danych, wystarczy tylko wywołać metodę `SubmitChanges` na rzecz obiektu `bazaDanychTelefony`. Powyższe czynności ujęte w języku C# prezentuje listing 3.3.

Listing 3.3. *Modyfikowanie istniejącego rekordu*

```
private void button2_Click(object sender, EventArgs e)
{
   //pobieranie kolekcji
   var listaOsobDoZmianyWieku = from osoba in listaOsob
                                where (osoba.Wiek<18 || !osoba.Imię.EndsWith("a"))
                                select osoba;

   //wyświetlanie pobranej kolekcji
   string s = " Lista osób niebędących pełnoletnimi kobietami:\n";
   foreach (Osoba osoba in listaOsobDoZmianyWieku) s += osoba.Imię + " " +
         ↪osoba.Nazwisko + " (" + osoba.Wiek + ")\n";
   MessageBox.Show(s);

   //modyfikowanie kolekcji
   foreach (Osoba osoba in listaOsobDoZmianyWieku) osoba.Wiek++;

   //wyświetlanie pełnej listy osób kolekcji po zmianie
   s = "Lista wszystkich osób:\n";
   foreach (Osoba osoba in listaOsob) s += osoba.Imię + " " + osoba.Nazwisko + " ("
        ↪+ osoba.Wiek + ")\n";
   MessageBox.Show(s);

   //zapisywanie zmian
   bazaDanychTelefony.SubmitChanges();
}
```

Musi pojawić się pytanie, skąd obiekt `bazaDanychTelefony` (obiekt typu `DataContext` reprezentujący w naszej aplikacji bazę danych) wie o modyfikacjach wprowadzonych do kolekcji `listaOsobDoZmianyWieku` otrzymanej zapytaniem LINQ z kolekcji uzyskanej metodą `bazaDanychTelefony.GetTable<Osoba>()`. Otóż stąd, że kolekcja `listaOsobDo` ↪`ZmianyWieku` przechowuje tylko referencje do obiektów tej tabeli, a nie kopie tych obiektów (por. zapytania LINQ w podrozdziale „Możliwość modyfikacji danych źródła", w rozdziale pierwszym). Zresztą w przypadku *LINQ to SQL* próba uruchomienia zapytania, w którym tworzymy kopie obiektów, skończy się zgłoszeniem wyjątku `NotSupportedException`. Tak więc wszystkie zmiany wprowadzane do kolekcji wprowadzane są automatycznie w buforze obiektu `bazaDanychTelefony`, przechowującym dane pobrane z bazy danych. A obiekt ten już potrafi zapisać je z powrotem do tabeli SQL Server. Warto przy tym pamiętać, że wszystkie zmiany wprowadzane w kolekcji są przechowywane tylko w niej aż do momentu wywołania metody `SubmitChanges`. Możemy więc dowolnie (wielokrotnie) modyfikować kolekcję bez obawy, że nadużywamy zasobów bazy danych i komputera, zmniejszając wydajność programu.


Wskazówka

W trakcie projektowania aplikacji należy zwrócić uwagę, czy plik bazy danych widoczny w podoknie *Solution Explorer* jest kopiowany do katalogu, w którym umieszczana jest skompilowana aplikacja (podkatalog *bin/Debug* lub *bin/Release*). Może to wprowadzać pewne zamieszanie i utrudniać śledzenie zmian w bazie danych np. za pomocą podokna *Database Explorer*.

Dodawanie i usuwanie rekordów

Co jeszcze możemy zmienić w tabeli? W zasadzie pozostaje tylko dodawanie nowych i usuwanie istniejących rekordów. Również to zadanie jest dzięki *LINQ to SQL* bardzo proste. W tym przypadku zmiany muszą być jednak wprowadzane wprost w obiekcie reprezentującym tabelę — instancji klasy `DataContext`, a nie w pobranej z niego kolekcji. Listing 3.4 pokazuje, jak dodać do tabeli nowy rekord. Po utworzeniu obiektu `DataContext` (polecenia identyczne jak w poprzednich listingach) obliczamy wartość pola *Id* dla nowego rekordu — do największej wartości *Id* w tabeli dodajemy jeden[4] — korzystamy przy tym z rozszerzenia `Max`. Następnie tworzymy obiekt typu `Osoba` i poleceniem `InsertOnSubmit` dodajemy go do tabeli. Rzeczywista zmiana nastąpi w momencie najbliższego wywołania metody `SubmitChanges` obiektu `DataContext`.

Listing 3.4. *Dodawanie rekordu do tabeli*

```
private void button3_Click(object sender, EventArgs e)
{
   //dodawanie osoby do tabeli
   int noweId = listaOsob.Max(osoba => osoba.Id) + 1;
   MessageBox.Show("Nowe Id: " + noweId);
   Osoba noworodek = new Osoba { Id = noweId, Imię = "Nela", Nazwisko =
                                 ↪"Matulewska", NumerTelefonu = 0, Wiek = 0 };
   listaOsob.InsertOnSubmit(noworodek);

   //zapisywanie zmian
   bazaDanychTelefony.SubmitChanges(); //dodawany jest także nowy rekord

   //wyświetlanie tabeli
   string s = "Lista osób:\n";
   foreach (Osoba osoba in listaOsob) s += osoba.Imię + " " + osoba.Nazwisko + " ("
        ↪+ osoba.Wiek + ")\n";
   MessageBox.Show(s);
}
```

Równie łatwo usunąć z tabeli rekord lub ich grupę. Należy tylko zdobyć referencję do odpowiadającego im obiektu (względnie grupy obiektów). W przypadku pojedynczego rekordu należy użyć metody `DeleteOnSubmit`, w przypadku ich kolekcji — `Delete` ↪`AllOnSubmit`. W obu przypadkach rekordy zostaną oznaczone jako przeznaczone do usunięcia i rzeczywiście usunięte z bazy danych przy najbliższym wywołaniu metody `SubmitChanges`. Listing 3.5 prezentuje metodę usuwającą z tabeli wszystkie osoby o imieniu Nela.

Listing 3.5. *Wszystkie modyfikacje w bazie danych wykonywane są w momencie wywołania metody SubmitChanges*

```
private void button4_Click(object sender, EventArgs e)
{
   //wybieranie elementów do usunięcia i ich oznaczanie
   IEnumerable<Osoba> doSkasowania = from osoba in listaOsob
```

[4] Właściwsze byłoby pewnie wyszukanie najmniejszej wolnej wartości, ale nie chciałem niepotrzebnie komplikować kodu. Zresztą zwykle tak postępuję, w myśl zasady „prosty kod oznacza mniejsze ryzyko błędu".

```
                                        where osoba.Imię == "Nela"
                                        select osoba;
    listaOsob.DeleteAllOnSubmit(doSkasowania);

    //zapisywanie zmian
    bazaDanychTelefony.SubmitChanges();

    //wyświetlanie tabeli
    string s = "Lista osób:\n";
    foreach (Osoba osoba in listaOsob) s += osoba.Imię + " " + osoba.Nazwisko + " ("
        ↪+ osoba.Wiek + ")\n";
    MessageBox.Show(s);
}
```

Klasa `DataContext` służy nie tylko do pobierania i modyfikacji zawartości bazy danych. Za pomocą jej metod `CreateDatabase` i `DeleteDatabase` można także tworzyć i usuwać bazy danych SQL Server z poziomu kodu. Za pomocą metody `ExecuteCommand` można wykonać podane w argumencie polecenie SQL, a za pomocą `ExecuteQuery` — uruchomić zapytanie SQL i odczytać pobrane przez nie dane.

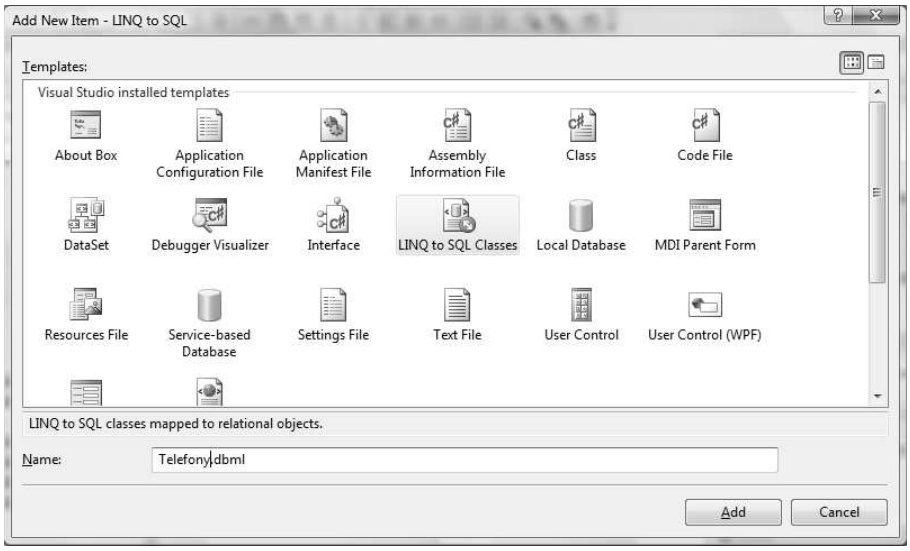
Wizualne projektowanie klasy encji

W przypadku bardziej rozbudowanych baz danych najprościej klasę encji utworzyć, korzystając z wbudowanego w Visual Studio narzędzia, którego nazwa w dosłownym tłumaczeniu to „projektant obiektowo-relacyjny” (ang. *Object Relational Designer*, w skrócie: O/R Designer) i który w istocie umożliwia automatyczne mapowanie struktury tabeli bazy danych w klasach C#. Nie chciałem od niego rozpoczynać omawiania *LINQ to SQL*, aby dać Czytelnikowi możliwość poznania technologii „od podszewki”, tj. od własnoręcznie przygotowanej klasy encji. Ale skoro już taką klasę sami przygotowaliśmy, nie musimy tego więcej powtarzać. Co prawda samodzielnie przygotowane klasy encji są zwykle bardziej zwarte, ale pisząc je, możemy popełniać błędy, których unikniemy, tworząc klasy myszką.

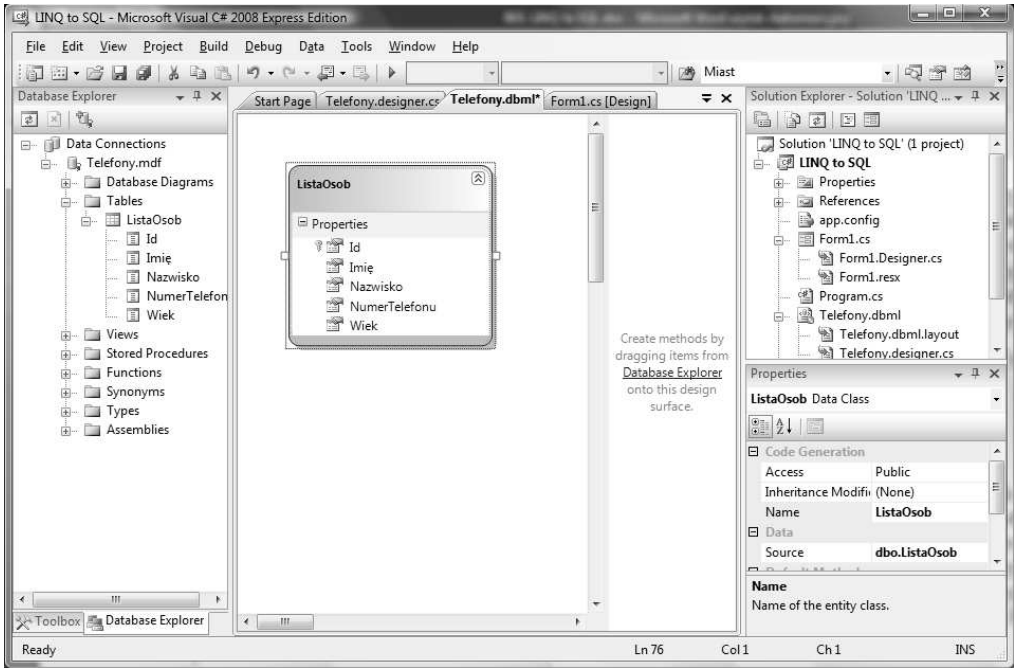
O/R Designer

Zacznijmy od utworzenia klasy encji dla tabeli *ListaOsob* w bazie *Telefony.mdf*. W tym celu z menu *Project* wybieramy polecenie *Add New Item*. Pojawi się okno widoczne na rysunku 3.1. Zaznaczamy w nim ikonę *LINQ to SQL Classes*. W polu *Name* podajemy nazwę nowego pliku: *Telefomy.dbml*. Po zamknięciu okna dialogowego zobaczymy nową zakładkę. Na razie pustą, nie licząc napisów informujących, że na dwa widoczne na niej panele należy przenosić elementy z podokna *Database Explorer*.

Przenieśmy na lewy panel tabelę *ListaOsob* (rysunek 3.2). Konwencja nazw w pliku *Telefony.dbml* nie przystaje niestety do nazw, jakich użyłem w bazie *Telefony.mdf*. Automatycznie tworzona klasa encji przejmuje bowiem nazwę tabeli, tj. `ListaOsob` (nasza ręcznie przygotowana klasa encji nazywała się `Osoba`). Z kolei własność tylko do odczytu zdefiniowana w klasie `TelefonyDataContext`, która zwraca tabelę (a konkretnie



Rysunek 3.1. *Tworzenie klasy reprezentującej dane*



Rysunek 3.2. *Zakładka z O/R Designer*

kolekcję typu `Table<ListaOsob>`), nazywa się `ListaOsobs` z „s” dodanym na końcu. Gdyby tabela nazywała się np. *Person*, klasa encji nazywałaby się `Person`, a własność reprezentująca tabelę — `Persons`. Wówczas nazwy brzmiałyby dużo lepiej. A tak mamy `ListaOsobs`.

Co prawda nazwy utworzonych przez O/R Designera klas można łatwo zmienić — wystarczy na zakładce *Telefony.dbml* kliknąć nagłówek tabeli (rysunek 3.2) i wpisać nowe nazwy. To samo dotyczy nazw pól domyślnie ustalonych na identyczne jak znalezione w tabeli SQL Server. Jeżeli wpiszemy nazwę różną od tej w tabeli, O/R Designer zmodyfikuje atrybut `Column`, który wskaże pole tabeli, z jakim definiowane pole obiektu ma być związane.

Utworzona przez O/R Designer klasa `ListaOsob` (można ją znaleźć w pliku *Telefony.designer.cs*) jest klasą encji o identycznej funkcji jak zdefiniowana przez nas wcześniej klasa `Osoba`. Nie jest jednak z nią identyczna. Tę pierwszą O/R Designer wyposażył poza polami, które i my zdefiniowaliśmy, również w zbiór metod i zdarzeń. Zawiaduje nimi O/R Designer i poza wyjątkowymi sytuacjami raczej nie należy edytować ich kodu. Same pola, inaczej niż w naszej prostej implementacji, są w nowej klasie prywatne i poprzedzone znakiem podkreślenia. Zwróćmy uwagę, że te z nich, które odpowiadają tym polom tabeli, które dopuszczają pustą wartość, zdefiniowane zostały przy użyciu typu parametrycznego `System.Nullable<typ>`, gdzie *typ* to oryginalny typ danych .NET (w naszym przypadku `int` lub `string`). Dostęp do tych pól możliwy jest poprzez publiczne własności. Przykład takiej własności odpowiadającej polu *Id* widoczny jest na listingu 3.6. Własność poprzedzona jest atrybutem `Column`, którego używaliśmy też w naszej wcześniejszej klasie encji. Jej argumenty są jednak inne (poza `IsPrimaryKey` ustawionym na `true`). Ze względu na identyczną nazwę pola tabeli i własności klasy argument `Name` został tu pominięty. Dwa nowe argumenty to `Storage`, który wskazuje prywatne pole przechowujące wartość komórki, oraz `DbType`. W tym ostatnim przechowywana jest informacja o oryginalnym typie pola tabeli (kolumny). Tworzy go nazwa typu dozwolona przez SQL Server (np. `Int` lub `NVarChar(MAX)`), uzupełniona ewentualnie frazą `NOT NULL`, jeżeli baza nie dopuszcza pustych wartości dla tego pola.

Listing 3.6. *Własność Id klasy ListaOsob. Towarzyszy jej pole zdefiniowane jako private int _Id*

```
[Column(Storage="_Id", DbType="Int NOT NULL", IsPrimaryKey=true)]
public int Id
{
   get
   {
      return this._Id;
   }
   set
   {
      if ((this._Id != value))
      {
         this.OnIdChanging(value);
         this.SendPropertyChanging();
         this._Id = value;
         this.SendPropertyChanged("Id");
         this.OnIdChanged();

   }
}
```

Wspomniana klasa reprezentująca całą bazę danych zawiera własności odpowiadające tabelom bazy. Jest zatem, to modne ostatnio określenie, *strongly typed*. Pozwala dzięki temu na bezpieczny, tj. zachowujący typy danych, dostęp do zewnętrznego zasobu — do danych w bazie SQL Server.

Nowe klasy `TelefonyDataContext` i `ListaOsob` można wykorzystać tak, jak robiliśmy do tej pory z klasą `Osoba` i „ręcznie" tworzoną instancją klasy `DataContext`. Listing 3.7 pokazuje kod metody, w której zmiany dotyczą w zasadzie wyłącznie nazw klas: z wcześniej używanych `DataContext` i `Osoba` na `TelefonyDataContext` i `ListaOsob`. Skorzystałem także z własności `ListaOsobs` (z „s" na końcu) pierwszej z klas, aby pobrać referencję do kolekcji zawierającej dane z tabeli `ListaOsob` (wyróżnienie w listingu 3.7).

Listing 3.7. *Korzystanie z automatycznie utworzonej klasy encji do pobierania i modyfikacji danych z tabeli SQL Server*

```
private void button5_Click(object sender, EventArgs e)
{
   //tworzenie obiektu DataContext i pobieranie danych z tabeli
   string nazwaPliku = "Telefony.mdf";
   nazwaPliku = Path.GetFullPath(nazwaPliku);
   if (!File.Exists(nazwaPliku))
   {
      MessageBox.Show("Brak pliku " + nazwaPliku);
      return;
   }
   TelefonyDataContext bazaDanychTelefony = new TelefonyDataContext(nazwaPliku);
   var listaOsob = bazaDanychTelefony.ListaOsobs;

   //pobieranie kolekcji
   var listaOsobPelnoletnich=from osoba in listaOsob where osoba.Wiek>=18 select osoba;

   //wyświetlanie pobranej kolekcji
   string s = "Lista osób pełnoletnich:\n";
   foreach (ListaOsob osoba in listaOsobPelnoletnich)
      s += osoba.Imię + " " + osoba.Nazwisko + " (" + osoba.Wiek + ")\n";
   MessageBox.Show(s);

   //informacje o pobranych danych
   MessageBox.Show("Typ: " + listaOsobPelnoletnich.GetType().FullName);
   MessageBox.Show("Ilość pobranych rekordów: " +
                  ↪listaOsobPelnoletnich.Count().ToString());
   MessageBox.Show("Suma wieku wybranych osób: "
                ↪+ listaOsobPelnoletnich.Sum(osoba => osoba.Wiek).ToString());
   MessageBox.Show("Imię pierwszej osoby: " + listaOsobPelnoletnich.First().Imię);

   s = "Pełna lista osób:\n";
   foreach (ListaOsob osoba in listaOsob)
      s += osoba.Imię + " " + osoba.Nazwisko + " (" + osoba.Wiek + ")\n";
   MessageBox.Show(s);
}
```

Współpraca z kontrolkami tworzącymi interfejs aplikacji

Ewidentnie nadużywam metody `MessageBox.Show`. Bardziej naturalne do prezentacji danych jest oczywiście korzystanie z kontrolek, np. `DataGridView`. Bez trudu można taki komponent wypełnić danymi udostępnianymi przez klasę `TelefonyDataContext`. Dodajmy do projektu nową formę i zdefiniujmy w niej pole — instancję tej ostatniej klasy:

```
TelefonyDataContext bazaDanychTelefony = new TelefonyDataContext();
```

Następnie wystarczy umieścić na formie kontrolkę `DataGridView` i przypisać jej własności `DataSource` odpowiednią tabelę (np. w konstruktorze za poleceniem `Initialize`
`↪Component();`):

```
dataGridView1.DataSource = bazaDanychTelefony.ListaOsobs;
```

Jeżeli nie chcemy prezentować pełnej zawartości tabeli, własności `DataSource` możemy przypisać kolekcję zwracaną przez zapytanie LINQ:

```
dataGridView1.DataSource = from osoba in bazaDanychTelefony.ListaOsobs
                           where osoba.Wiek >= 18
                           orderby osoba.Imię
                           select new { osoba.Imię, osoba.Nazwisko, osoba.Wiek };
```

Kontrolka `DataGridView` pozwala na edycję danych z kolekcji, jednak pod warunkiem, że prezentowana w niej kolekcja nie jest zbudowana z obiektów anonimowych typów, lecz z referencji do obiektów ze źródła danych (instancji klas encji):

```
dataGridView1.DataSource = from osoba in bazaDanychTelefony.ListaOsobs
                           where osoba.Wiek >= 18
                           orderby osoba.Imię
                           select osoba;
```

Aby wprowadzone w ten sposób modyfikacje danych wysłać z powrotem do bazy, należy tylko wywołać metodę `SubmitChanges` instancji klasy `TelefonyDataContext`, tj. obiektu `bazaDanychTelefony`.

Do formy możemy dodać także rozwijaną listę `ComboBox`. Możemy „podłączyć" ją do tego samego wiązania, którym z bazą połączona jest kontrolka `dataGridView1`. W tym celu do konstruktora należy dopisać dwa polecenia:

```
comboBox1.DataSource = dataGridView1.DataSource;
comboBox1.DisplayMember = "Imię";
```

Pierwsze wskazuje na źródło danych rozwijanej listy — jest nim ta sam kolekcja, która prezentowana jest w kontrolce `dataGridView1`. Drugie precyzuje, którą kolumnę danych rozwijana lista ma prezentować. Ze względu na korzystanie ze wspólnego wiązania zmiana aktywnego rekordu w siatce spowoduje zmianę rekordu w rozwijanej liście i odwrotnie.

Kreator źródła danych i automatyczne tworzenie interfejsu użytkownika

Do utworzenia odpowiednich obiektów można jednak ponownie zaprząc Visual C#. Oczywiście, w takim przypadku kod nie będzie tak prosty i przejrzysty jak ten napisany samodzielnie, ale za to nie musimy go przygotowywać i sprawdzać sami. Cel, do którego dążymy, to utworzenie źródła danych, takiego, do jakich przyzwyczaiło nas ADO.NET. Jednak w tym przypadku czerpać ono będzie dane nie bezpośrednio z bazy danych, ale za pomocą *LINQ to SQL*, a więc poprzez zaprojektowaną w O/R Designerze klasę `TelefonyDataContext` i towarzyszącą jej klasę encji. Dysponując takim źródłem, będziemy mogli wiązać z danymi kontrolki, z których budujemy interfejs aplikacji.

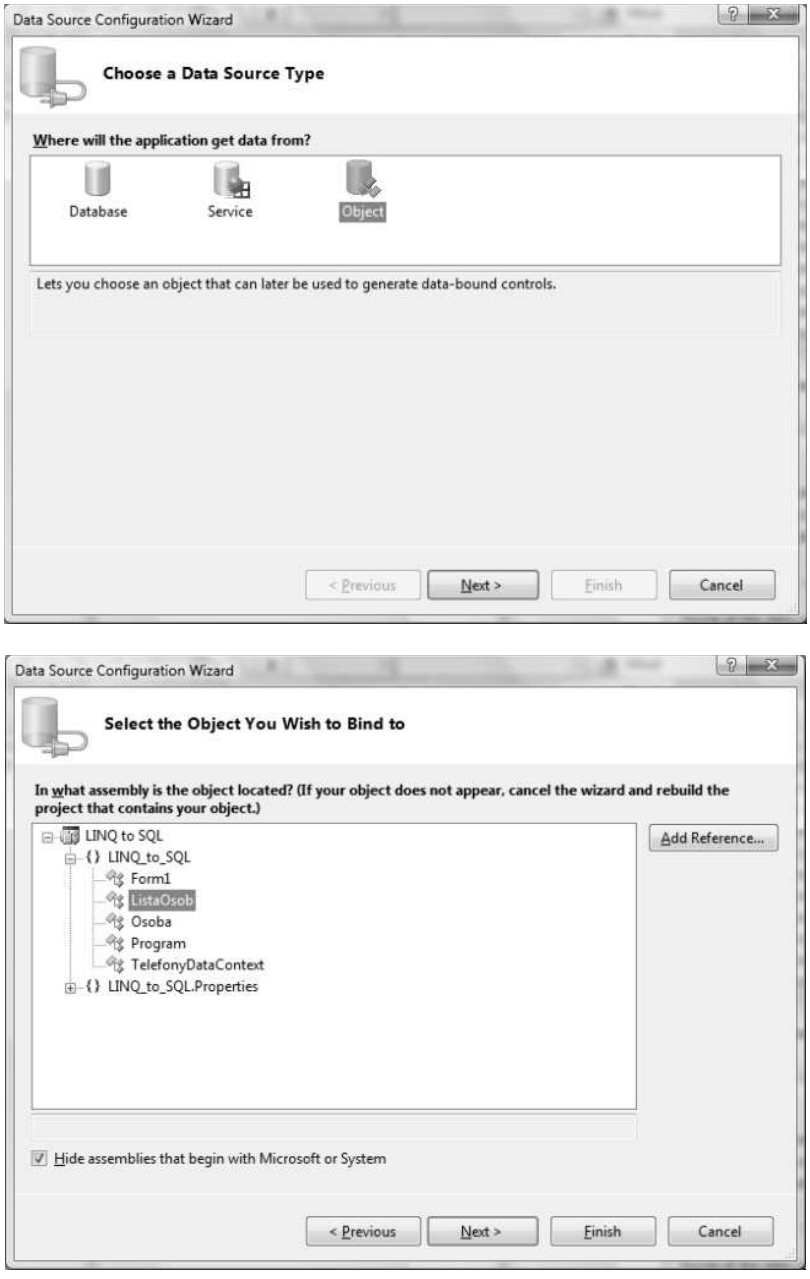
Wybierzmy z menu *Data* polecenie *Add New Data Source*.... Pojawi się kreator *Data Source Configuration Wizard*. W jego pierwszym kroku wybieramy źródło danych (rysunek 3.3, górny); zaznaczmy ikonę *Object*. Klikamy *Next* >. W następnym kroku wybieramy klasę, która zawiera informacje na temat struktury tabeli *ListaOsob* (rysunek 3.3, dolny). W naszym przypadku jest to klasa `ListaOsob`. Wreszcie klikamy *Finish*.

Po zamknięciu kreatora wybierzmy polecenie *Show Data Sources* z menu *Data*. Pojawi się nowe podokno o nazwie *Data Sources*, w którym zobaczymy nowe, utworzone przed chwilą źródło danych. Możemy sprawdzić jego pola — powinny odpowiadać polom tabeli (rysunek 3.4). Tak zdefiniowane źródło danych nie zawiera niestety samych danych, a jedynie informacje o ich strukturze. To jednak wystarczy, aby bardzo przyspieszyć projektowanie interfejsu użytkownika w aplikacjach bazodanowych.

Gdy dysponujemy źródłem danych, możemy zaprojektować interfejs aplikacji, korzystając z metody „przeciągnij i upuść". Możemy bowiem przeciągać na formę poszczególne pola tabeli *ListaOsob* widoczne w podoknie *Data Sources*. Powstanie wówczas kontrolka z automatycznie dodanym opisem. Typ kontrolki, która powstanie, można wcześniej wybrać w rozwijanych listach związanych z każdym polem, w podoknie *Data Sources* (rysunek 3.4). Przeniesienie całej tabeli spowoduje przy domyślnych ustawieniach umieszczenie na formie siatki `dataGridView`. Jednak jeżeli wcześniej w pozycji *ListaOsob* (w podoknie *Data Sources*) z rozwijanej listy wybierzemy *Details* zamiast domyślnego *DataGridView*, zamiast siatki na formie umieszczony zostanie zbiór kontrolek zgodnie z wyborem dla każdego z pól. Wraz z pierwszą umieszczoną w ten sposób kontrolką na formie pojawi się komponent wiązania (`listaOsobBindingSource`) oraz kontrolka nawigacyjna (`listaOsobBindingNavigator`). Tę ostatnią możemy zobaczyć na szczycie podglądu formy na rysunku 3.4.

Tak jak uprzedzałem, źródło danych nie zawiera danych, a jedynie informacje o ich strukturze. Zatem po skompilowaniu kontrolki nie będą prezentowały żadnych sensownych informacji. Wystarczy jednak odpowiednio „podpiąć" kontrolkę wiązania `listaOsobBindingSource`, aby wszystko zaczęło grać. Zaczynamy od zdefiniowania pola formy będącego instancją naszej klasy reprezentującej bazę danych:

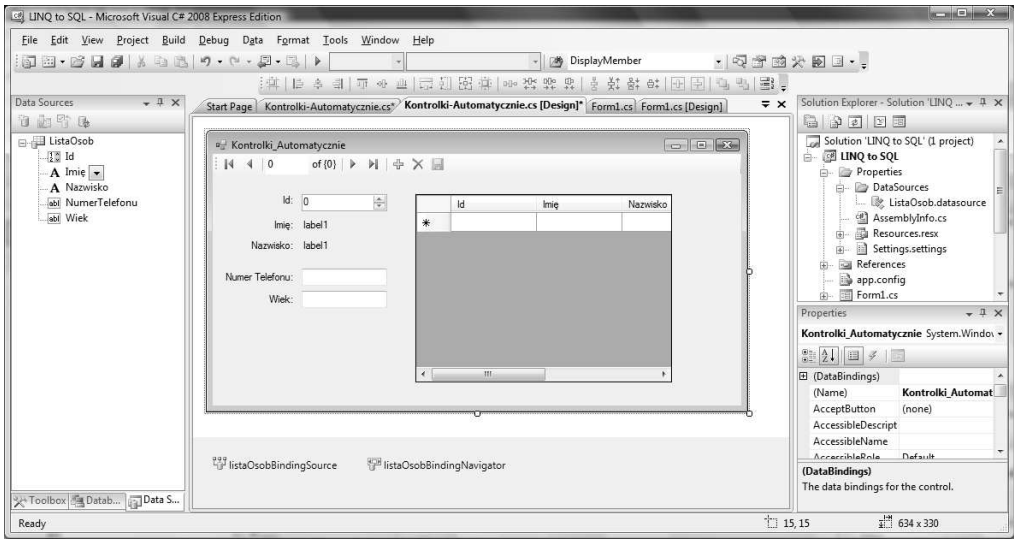
```
TelefonyDataContext bazaDanychTelefony = new TelefonyDataContext();
```



Rysunek 3.3. *Pierwszy krok kreatora obiektu — źródła danych LINQ to SQL*

Następnie w konstruktorze wiążemy udostępnianą przez ten obiekt tabelę *ListaOsob* z obiektem wiązania `listaOsobBindingSource`:

```
listaOsobBindingSource.DataSource = bazaDanychTelefony.ListaOsobs;
```



Rysunek 3.4. *Projektowanie interfejsu aplikacji na bazie źródła danych LINQ*

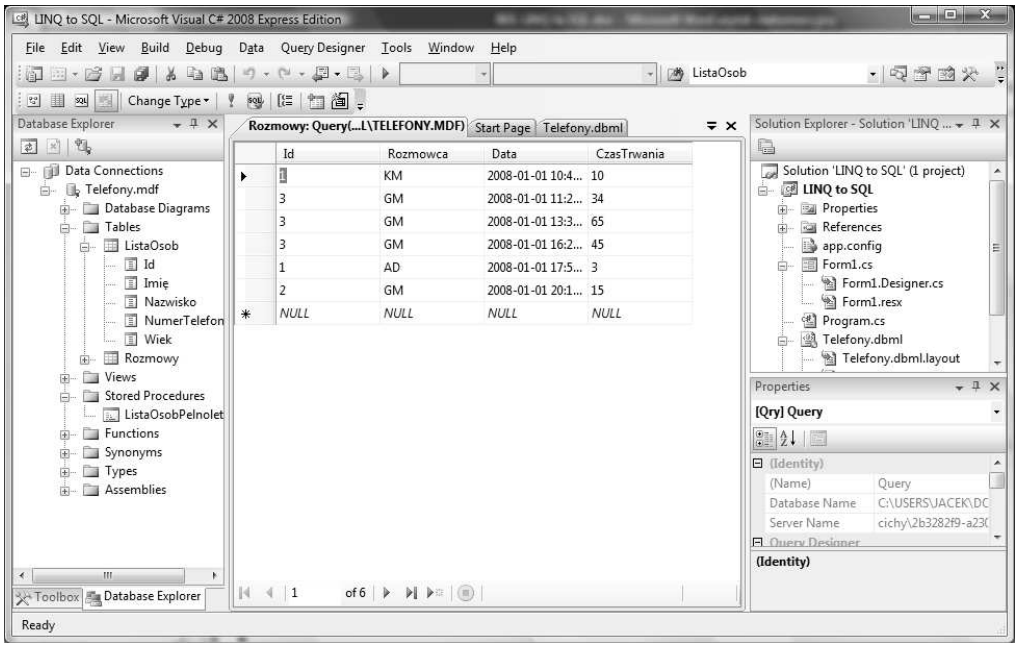
Wszystkie wizualne kontrolki tworzące interfejs użytkownika związane są z tym jednym obiektem wiązania, zatem zmiana aktualnego rekordu w którejkolwiek z nich spowoduje automatyczną zmianę widocznego rekordu w pozostałych. Możliwe jest oczywiście zapisywanie zmian wprowadzonych za pomocą kontrolek. Wystarczy jedynie wywołać metodę `bazaDanychTelefony.SubmitChanges`.

Łączenie danych z dwóch tabel — operator join

Informacje o osobach mogą być umieszczone w kilku tabelach. Dla przykładu w osobnej tabeli mógłby być umieszczony spis rozmów, jakie przeprowadzały osoby z tabeli *ListaOsob*. Tabela będzie zawierać identyfikator osoby, personalia jej rozmówcy, datę przeprowadzenia rozmowy i jej czas trwania w minutach. Pola o nazwie *Id* i *CzasTrwania* będą typu `int`, pole *Rozmowca* — typu `nvarchar(MAX)`, a pole *Data* — typu `datetime`. Żadne z pól nie będzie dopuszczać pustych wartości. Kluczem głównym możemy uczynić tylko pole *Data*. Zapiszmy tabelę, nazywając ją po prostu *Rozmowy*. Pole *Id* w nowej tabeli będzie tym samym *Id*, którym identyfikowana jest osoba w tabeli *ListaOsob*. Zatem każdej osobie będzie można przyporządkować listę rozmów, a każdej rozmowie — tylko jednego rozmówcę. Tabela *ListaOsob* będzie więc naszą tabelą-rodzicem, a tabela *Rozmowy* — tabelą-dzieckiem. Tabelę należy oczywiście wypełnić jakimiś przykładowymi danymi (rysunek 3.5).

Następnie przejdźmy na zakładkę *Telefony.dbml* i przenieśmy nową tabelę na lewy panel, obok tabeli *ListaOsob*. Do klasy `TelefonyDataContext` dodana zostanie nowa własność o nazwie `Rozmowies` (znowu nie dopasowaliśmy się do proangielskiego schematu nazw).

Pobierzmy teraz listę rozmów trwających dłużej niż dziesięć minut wraz z personaliami dzwoniącej osoby. W tym celu zapytanie LINQ rozszerzymy o operator `join`, który pobierać będzie rekordy o tych samych wartościach pól *Id* w obu tabelach. Pokazuje to listing 3.8.



Rysunek 3.5. *Nowa tabela bazy Telefony.mdf*

Listing 3.8. *Pominąłem polecenia tworzące obiekt bazaDanychTelefony*

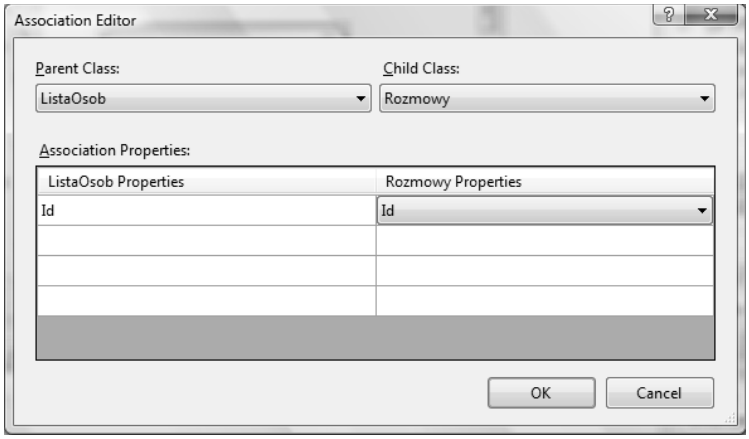
```
var listaOsob = bazaDanychTelefony.ListaOsobs;
var rozmowy = bazaDanychTelefony.Rozmowies;

IEnumerable<string> listaDlugichRozmow =
   from osoba in listaOsob
   join rozmowa in rozmowy on osoba.Id equals rozmowa.Id
   where rozmowa.CzasTrwania > 10
   select osoba.Imię + " " + osoba.Nazwisko + ", " + rozmowa.Data.ToString()
      ↪+" (" + rozmowa.CzasTrwania + ")";

string s = "Lista rozmów trwających dłużej niż 10 min:\n";
foreach (string opis in listaDlugichRozmow) s += opis + "\n";
MessageBox.Show(s);
```

Relacje (Associations)

Zamiast korzystać z operatora `join`, możemy połączyć obie tabele w O/R Designerze. A w zasadzie nie tabele, tylko reprezentujące je klasy. Przejdźmy na zakładkę *Telefony.dbml*, kliknijmy prawym przyciskiem myszy nagłówek tabeli *ListaOsob* i z kontekstowego menu wybierzmy polecenie *Add*, *Association*. Pojawi się edytor relacji (rysunek 3.6). Z rozwijanej listy *Child Class* wybieramy *Rozmowy*. Następnie w tabeli *Association Properties* pod spodem wybieramy w obu kolumnach pole *Id*.



Rysunek 3.6. *Edytor połączenia między obiektami reprezentującymi tabele*

Ustanowione połączenie zaznaczone zostanie w O/R Designerze strzałką łączącą dwie tabele. W podobny sposób zaznaczone byłyby relacje między tabelami, które zdefiniowane są w samej bazie danych. Skutek obu połączeń jest podobny.

Dzięki relacji łączącej obie tabele możemy uprościć zapytanie LINQ:

```
var listaDlugichRozmow = from rozmowa in rozmowy
                         where rozmowa.CzasTrwania > 10
                         select rozmowa.ListaOsob.Imię + " " +
                                ➥rozmowa.ListaOsob.Nazwisko + ", " +
                                ➥rozmowa.Data.ToString() +
                                ➥" (" + rozmowa.CzasTrwania + ")"; ;
```

Jest to możliwe, ponieważ do klasy encji `Rozmowy` dodana została własność `ListaOsob`, która wskazuje na jeden pasujący rekord tabeli *ListaOsob*. Z kolei klasa encji `ListaOsob` zawiera własność `Rozmowies` obejmującą pełen zbiór rekordów, w których wartość pola `Id` równa jest tej, jaka jest w bieżącym wierszu tabeli *ListaOsob*. To ostatnie pozwala na proste filtrowanie danych bez konieczności przygotowywania zapytania LINQ. A więc kolekcja uzyskana poleceniem:

```
var listaRozmowKarolinyMatulewskiej =
   ➥bazaDanychTelefony.ListaOsobs.Single(osoba => osoba.Id == 3).Rozmowies;
```

zawiera identyczne rekordy jak uzyskana zapytaniem:

```
var listaRozmowKarolinyMatulewskiej = from rozmowa in rozmowy
                                      where rozmowa.Id == 3
                                      select rozmowa;
```

Korzystanie z procedur składowanych

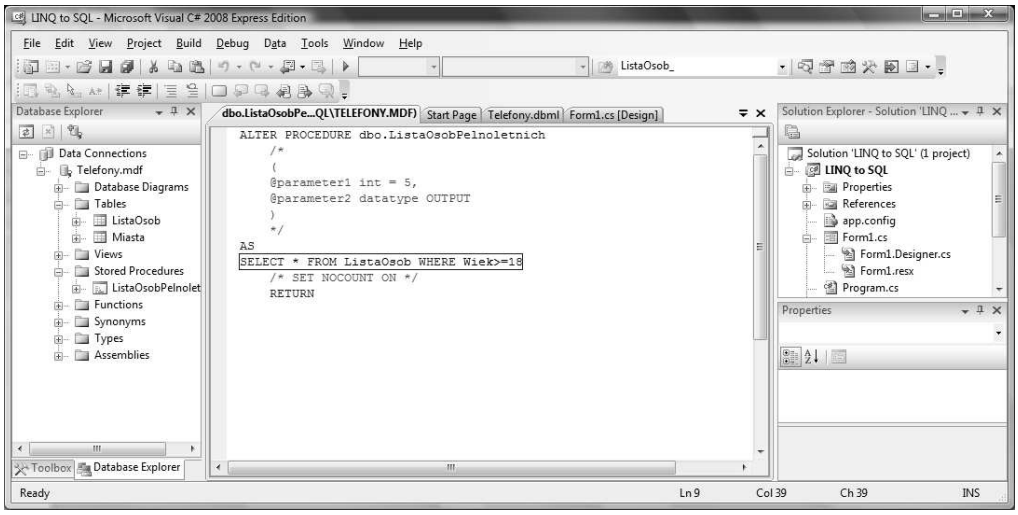
W momencie uruchamiania metody `SubmitChanges` obiekt klasy `DataContext` automatycznie tworzy zbiór poleceń SQL, które modyfikują zawartość bazy danych. Podobnie jest zresztą w momencie pobierania danych. Wówczas zapytanie LINQ jest tłumaczone na zapytanie SQL i wysyłane do bazy danych.

Zamiast korzystać z tych tworzonych automatycznie procedur i zapytań możemy wykorzystać własne procedury składowane w bazie danych (ang. *stored procedures*). Jest to w pewnym sensie krok wstecz, wymaga bowiem programowania w SQL, ale za to pozwala na zachowanie pełnej elastyczności podczas korzystania z SQL Server. I nawet jeżeli z procedur składowych nie będziemy korzystać do pobierania danych, warto je poznać, aby móc uruchamiać dowolne polecenia SQL, które pozwolą nam np. tworzyć nowe tabele w bazie.

Pobieranie danych za pomocą procedur składowanych

Utwórzmy i zapiszmy w bazie danych zapytanie pobierające personalia osób pełnoletnich zapisanych w tabeli *ListaOsob*. W tym celu otwórzmy plik *Telefony.mdf* w *Database Explorer*, przejdźmy do pozycji *Stored Procedures* i prawym przyciskiem myszy rozwińmy jej menu kontekstowe. Wybierzmy z niego polecenie *Add New Stored Procedure*.

Pojawi się edytor, w którym możemy wpisać kod SQL (rysunek 3.7). Za słowem kluczowym `AS` wpiszmy proste zapytanie SQL: `SELECT * FROM ListaOsob WHERE Wiek>=18`. Zmieńmy też nazwę procedury na `dbo.ListaOsobPelnoletnich`. Wreszcie zapiszmy zmiany w bazie danych (*Ctrl+S*). W podoknie *Database Explorer*, w gałęzi *Stored Procedures* bazy *Telefony.mdf*, powinna pojawić się pozycja *ListaOsobPelnoletnich*. Możemy wówczas przejść na zakładkę *Telefony.dbml* i przeciągnąć procedurę z podokna *Database Explorer* do pustego panelu po prawej stronie. To spowoduje dodanie do klasy `Telefony` ↪`DataContext` metody `ListaOsobPelnoletnich`. Zwraca ona dane pobrane przez zapytanie SQL w postaci kolekcji elementów typu `ListaOsobPelnoletnichResult`.



Rysunek 3.7. *Edycja zapytania SQL procedury umieszczonej w bazie danych*

Typ `ListaOsobPelnoletnichResult` został automatycznie utworzony za pomocą O/R Designera. W przypadku pobierania wszystkich pól z tabeli, tj. gdy w zapytaniu SQL po słowie `SELECT` widoczna jest gwiazdka (*), nowa klasa będzie równoważna typowi `ListaOsob` (i `Osoba`). Zapytanie SQL może jednak zwracać tylko wybrane pola, a wówczas tworzenie nowego typu jest bardziej przydatne.

Dzięki nowej metodzie uruchomienie składowanego zapytania SQL i odebranie zwracanych przez nie danych jest dziecinnie proste:

```
IEnumerable<ListaOsobPelnoletnichResult> listaOsobPelnoletnich =
   ↪bazaDanychTelefony.ListaOsobPelnoletnich();
```

Modyfikowanie danych za pomocą procedur składowanych

Przygotujmy teraz procedurę składowaną zwiększającą o jeden wiek wszystkich osób niebędących pełnoletnimi kobietami (listing 3.9). Zapiszmy ją w bazie pod nazwą `AktualizujWiek` i dodajmy do prawego panelu na zakładce *Telefony.dbml*.

Listing 3.9. *Procedura składowana modyfikująca dane w tabeli ListaOsob*

```
ALTER PROCEDURE dbo.AktualizujWiek
AS
UPDATE ListaOsob SET Wiek=Wiek+1 WHERE NOT SUBSTRING(Imię,LEN(Imię),1)='a' OR Wiek<18
 RETURN
```

Aby ją uruchomić z poziomu C#, wystarczy do kodu dodać instrukcję:

```
bazaDanychTelefony.AktualizujWiek();
```

Zmiany w tabeli bazy danych będą wprowadzone natychmiast i bezpośrednio, tj. np. bez konieczności uruchamiania metody `SubmitChanges` instancji `TelefonyDataContext`. Z tego wynika też, że jeżeli wcześniej pobraliśmy kolekcję danych zapytaniem LINQ, należy to teraz powtórzyć, aby uwzględnić aktualizacje.

Wykonywanie dowolnych poleceń SQL

Pobieranie danych i ich modyfikacja za pomocą składowanych poleceń SQL nie jest może tym, co programiści znający LINQ zbytnio doceniają. Warto jednak zwrócić uwagę, że istnieje grupa czynności, które w SQL jest wykonać najprościej. Przyjmijmy, że w naszej bazie danych tworzymy zbiór tabel dla każdego nowego zarejestrowanego użytkownika. Wówczas nie do przecenienia jest możliwość przygotowania składowanej procedury zawierającej kod SQL tworzący tabelę i wypełniający ją danymi. W tym celu wystarczy dodać nową procedurę składową (przykład widoczny jest na listingu 3.10), dodać ją za pomocą O/R Designera do klasy `TelefonyDataContext` i wywołać w kodzie C#. Nazwę i dane tabeli możemy przekazać przez parametry procedury. Pamiętajmy jednak, że nowa utworzona w ten sposób tabela nie będzie zmapowana i w konsekwencji nie będzie widoczna w obiekcie — instancji klasy `TelefonyDataContext`.

Listing 3.10. *Przykładowa procedura składowa tworząca tabelę*

```
ALTER PROCEDURE dbo.TwórzNowąTabelę
AS
CREATE TABLE "Faktury" (Id INT, NrFaktury INT, Opis NVARCHAR(MAX))
INSERT INTO "Faktury" VALUES(1, 1022, 'Faktura za styczeń 2008')
INSERT INTO "Faktury" VALUES(1, 1022, 'Faktura za luty 2008')
 RETURN
```