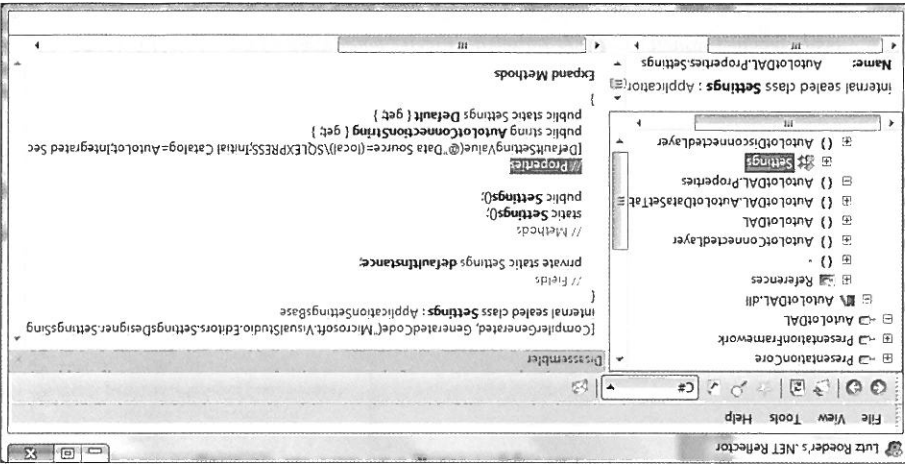


Egészen pontosan a megtehetően hosszú VisualDataGrndvIEWApp.Properties.Settings.AutoLotConnectionString értéket látjuk a kapcsolatsztring nevéként. Még furcsább az a tény, hogy ha átfutjuk a generált kódot, nem találunk referenciát a ConfigurationManager típusra, hogy kiolvassuk az értéket a <connectionstring> elemből. Azt azonban láthatjuk, hogy az automatikusan generált adatillesztő objektumot (lásd részleteseen később) részben a következő privát segédfüggvény meghívásával hoztuk létre:

```
private void InitConnection()
{
    this._connection =
        new global::System.Data.SqlClient.SqlConnection();
    this._connection.ConnectionString = global::
        VisualDataGrndvIEWApp.Properties.Settings.Default.
        AutoLotConnectionString;
}
```

A connectionString tulajdonságot a settings.default meghívásával állítjuk be. Minden Visual Studio 2008 projekt típus fenntart az egész alkalmazásra kiterjedő beállításokat, amelyek metaadatként égneek bele a szerelvénybe, amikor lefordítjuk az alkalmazást. Ha megnyitunk egy lefordított alkalmazást a reflector.exe használataival (lásd az előző kötet 2. fejezetét), akkor megnevezhetjük ezt a belső típust (lásd a 23.22. ábrát).



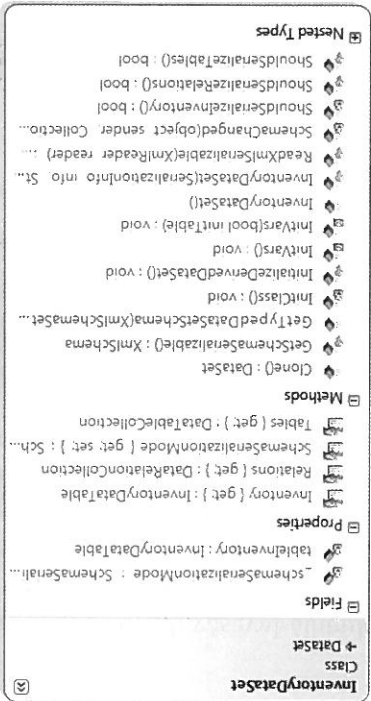
23.22. ábra: A Settings objektum tartalmaz egy beágyazott kapcsolatsztringértéket

A fentiek ismeretében lehetséges lenne úgy telepíteni az alkalmazást, hogy nem adjuk oda a *.config fájlt, mivel az alapértelmezés szerint a beágyazott értéket használjuk, ha nincsen jelen kliensoldali *.config fájl.

Megjegyzés A Visual Studio 2008 beállításában a programozási modellek teljes ismertetése túlmutat a fejezet hatáskörén. Ha szeretnének többet meg tudni ezekről, olvassuk el a „NET Framework 3.5 SDK dokumentációjának „Managing Application Settings” című témakörét.

A generált DataSet vizsgálata

Nézünk meg a generált kód néhány alapvető aspektusát. Először szúrjunk be egy új osztálydiagramot a projektbe, amelyhez válasszuk ki a projekt ikonját a Solution Explorerben, majd kattintsunk a View Class Diagram gombra. A be-menet alapján a varázsló létrehoz egy új dataset típust, amelynek ebben az esetben InventoryDataSet a neve. Az osztály számos tagot definiál, amelyek közül az Inventory tulajdonság a legfontosabb (lásd a 23.23. ábrát).



23.23. ábra: A Data Source Configuration Wizard létrehozott egy erősen típusos DataSet típust

Ha a Solution Explorerben kattintunk az InventoryDataSet.xsd fájlra, akkor betöltjük a Visual Studio 2008 DataSet Designer eszközt (erről részlete-ebben később lesz szó). Ha ezen a tervezőn belül bárhova kattintunk a jobb egérgombbal, és kiválasztjuk a View Code lehetőséget, meglehetősen üres részleges osztálydefiniációt láthatunk:

```
public partial class InventoryDataSet {
    {
    }
    {
    }
}
```

Az események a tervező által karbantartott fájlon (InventoryDataSet.Designer.cs) belül zajlanak. Ha a Solution Explorerrel megnyitjuk ezt a fájlt, látható, hogy az InventoryDataSet valójában kibővít a DataSet osztálytípust. Amikor mi (vagy a varázsló) létrehozunk egy olyan osztályt, amely kibővít a DataSet típust, akkor *erősen típusos* dataSetet valósítunk meg. Az erősen típusos dataSet objektumok használatának egyik előnye az, hogy több olyan tulajdonságot tartalmaznak, amelyek közvetlenül az adatbázis tábláik neveire képezhetők le. Így ahelyett, hogy bele kéne ásni magunkat a táblák gyűjteményébe a `Tables` tulajdonság segítségével, egyszerűen használhatjuk az `Inventory` tulajdonságot. Vizsgáljuk meg az alábbi kódreszletet, amely az érthetőség kedvéért megjegyzéseket is tartalmaz:

```
// Ez mind a tervező által létrehozott kód.
public partial class InventoryDataSet : global::System.Data.DataSet
{
    // InventoryDataTable típusú tagváltozó.
    private InventoryDataTable tableInventory;

    // Minden konstruktor meghív egy InitClass() segédmetódust.
    public InventoryDataSet()
    {
        this.InitClass();
    }

    // Az InitClass() előkészíti a dataSetet, és hozzáadja az
    // InventoryDataTable osztályt a Tables gyűjteményhez.
    private void InitClass()
    {
        this.DataSetName = "InventoryDataSet";
        this.Prefix = "";
        this.Namespace = "http://tempuri.org/InventoryDataSet.xsd";
        this.EnforceConstraints = true;
        this.SchemaSerializationMode =
            global::System.Data.SchemaSerializationMode.IncludesSchema;
        this.tableInventory = new InventoryDataTable();
        base.Tables.Add(this.tableInventory);
    }
}
```

Az egyedi datarow típus sokkal kevésbé egzotikus, mint a generált dataset vagy datatabl_e. Ez az osztály kibővíti a datarow osztályt (lásd 23.25. ábra), és olyan tulajdonságokat nyújt, amelyek közvetlenül az Inventory tábla sémájára képezhetők le (természetesen a tulajdonságok típusa az oszlopok típusának megfelelő).

Megjegyzés Az erősen típusos DataTabl_e számos olyan eseményt is definiál, amelyekkel megfigyelhetjük a táblázat adatainak módosításait.

Sokkal érdekesebb, hogy ez az erősen típusos datatabl_e osztály egy olyan módszerkészletet definiál (lásd a 23.24. ábrát), amely sorok beszűrését, hely-meghatározását és törölését teszi lehetővé a táblában erősen típusos tagok használatával (vonzó alternatíva a row és column indexelők manuális navigálásához).

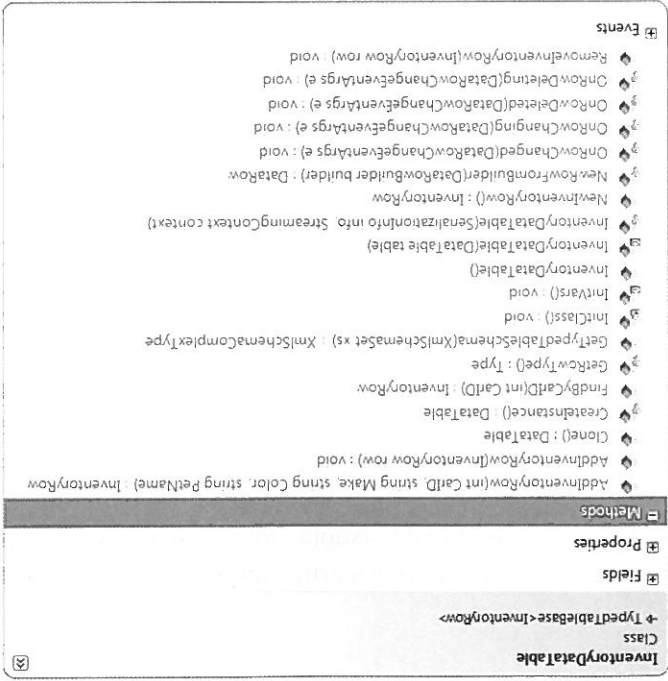
Sokkal érdekesebb, hogy ez az erősen típusos datatabl_e osztály egy olyan rekordok aktuális számát.

valamint egy egyedi indexelőt és egy count tulajdonságot, hogy beolvassa a in alapul (CarIDColumn, ColorColumn, MakeColumn és PetNameColumn), olyan tulajdonságkészletet definiál, amely a fizikai Inventory tábla oszlopneve-pusú, mint a most megvizsgált erősen típusos dataset tagváltozója) egy torydataset osztályba. Az Inventorydatatabl_e osztály (amely ugyanolyan ti-egy *erősen típusos* datarow osztályt, amelyek mindegyikét beágyazta az Inven-A fentiekhez hasonlóan, a varázsló létrehozott egy *erősen típusos* datatabl_e és

A generált DataTabl_e és DataRow típusok vizsgálata

Amellett, hogy egyesíti a datatabl_e objektum fenntartásának részleteit, a ter-vező által generált erősen típusos dataset tartalmazhat hasonló logikát, hogy felárjon olyan datarel_{ation} objektumokat (ezekkel jelenleg nem rendelke-zünk), amelyek a táblák közötti kapcsolatot képviselik.

```
// Az frásvédeett Inventory tulajdonság visszaadja
// az InventoryDataTable tagváltozót.
public InventoryDataTable Inventory
{
    get { return this.tableInventory; }
}
```

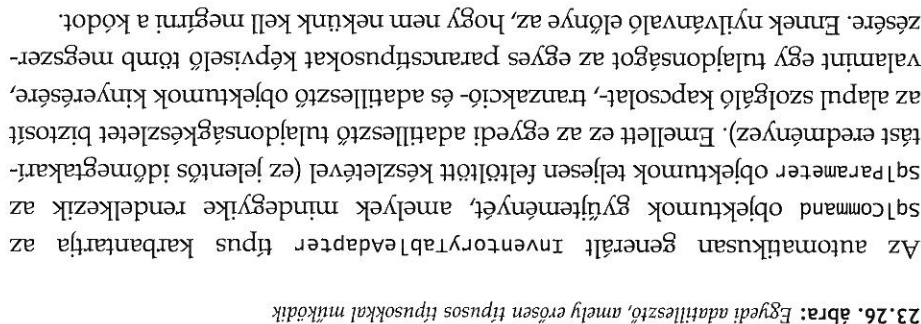



23.24. ábra: Az egyedi DataTable típus



23.25. ábra: Az egyedi DataRow típus

Az erős típusok megítélése a kapcsolatok nélküli típusok számára a Data Source Configuration Wizard használatának komoly előnye, ugyanis az erősen típusos osztályok manuális hozzáadása megtehetően hosszadalmas (bár kivitelezhető) lenne. A varázsló szerencsére egyedi adatillesztő objektumot generál, amely képes felhívni és módosítani az inventortorpydataset és az inventortorpydatasetle osztálytípusokat (lásd a 23.26. ábrát).



valamint egy tulajdonságot az egyes paraméterpárokat képviselő tömb megszerzésére. Ennek nyilvánvaló előnye az, hogy nem nekünk kell megírni a kódot.

Generált típusok használata a kódban

Ha megvizsgáljuk az űrlapból származtatott típus Load eseménykezelőjét, látható, hogy az egyedi adattílesztő f111() metódusa indításakor meghívódik, és átadja az egyedi dataset által fenntartott egyedi datatable típust:

```
private void MainForm_Load(object sender, EventArgs e)
{
    this.inventoryTablAdapter.F111(this.inventoryDataset.Inventory);
}
```

Ugyanezt az egyedi adattílesztő objektumot használhatjuk arra, hogy frissítsük a táblázat módosításait. Adjunk az űrlap felhasználói felületéhez egy Button vezérőelemet (btnUpdateInventory), készítsünk eseménykezelőt a Click eseményhez, és írjuk bele a következő kódot:

```
private void btnUpdateInventory_Click(object sender, EventArgs e)
{
```

```
    // Ez visszaadja az Inventory táblában végrehajtott összes
    // módosítást feldolgozásra az adatbázisnak.
    this.inventoryTablAdapter.Update();
    this.inventoryDataset.Inventory);
}
```

```
    // Megkapjuk a táblázat friss másolatát.
    this.inventoryTablAdapter.F111(this.inventoryDataset.Inventory);
}
```

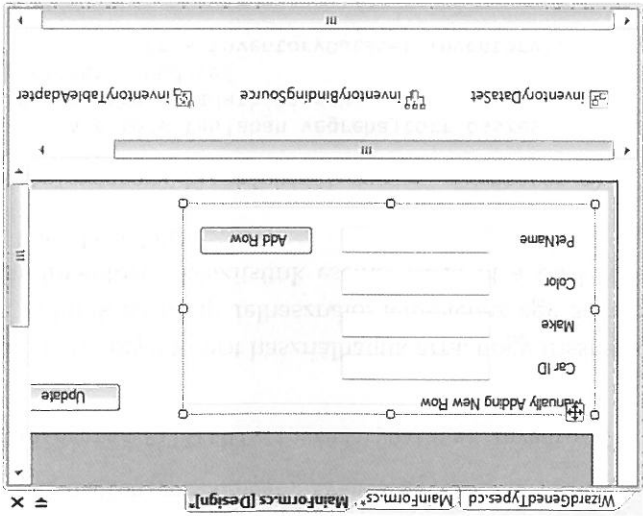
Futtassuk megint az alkalmazást; adjunk hozzá töröljünk vagy módosítsunk rekordot a táblázatban; végül kattintsunk az Update gombra. Ha megint futtatjuk a programot, láthatjuk, hogy a változások megjelentek.

Az összes erősen típusos osztályt közvetlenül felhasználhatjuk a kódukból (nagyjából) ugyanúgy, ahogy azt már meglettük a korábbiakban. Tételünk fel például, hogy olyan felhasználói felület-tartománnyal frissítettük az űrlapunkat (lásd a 23.27. ábrát), amely lehetővé teszi a felhasználó számára új rekord beírását szövegmezők segítségével (ez voltaképpen felesleges ebben a példában, ugyanis ezt a DataGridview megteszi helyettünk).

Az új Button vezérőelem Click eseménykezelőjébe beírhatjuk az alábbi kódot:

```
private void btnAddRow_Click(object sender, EventArgs e)
{
    // Lekérdezzük az adatokat a szövegmezőkből
    int id = int.Parse(txtCarID.Text);
    string make = txtMake.Text;
    string color = txtColor.Text;
    string petName = txtPetName.Text;
}
```

```
// Egyedi ílllesztő használataval beszzúnuk egy sort.  
inventoryTableAdapter.Insert(id, make, color, petname);  
  
// Újra feltöltjük a táblázatot.  
this.inventoryTableAdapter.Fill(this.inventoryDataSet.Inventory);  
}
```



23.27. ábra: Egyszerű frissítés az újlap típusához

Manuálisan is hozzáadhatunk egy új sort:

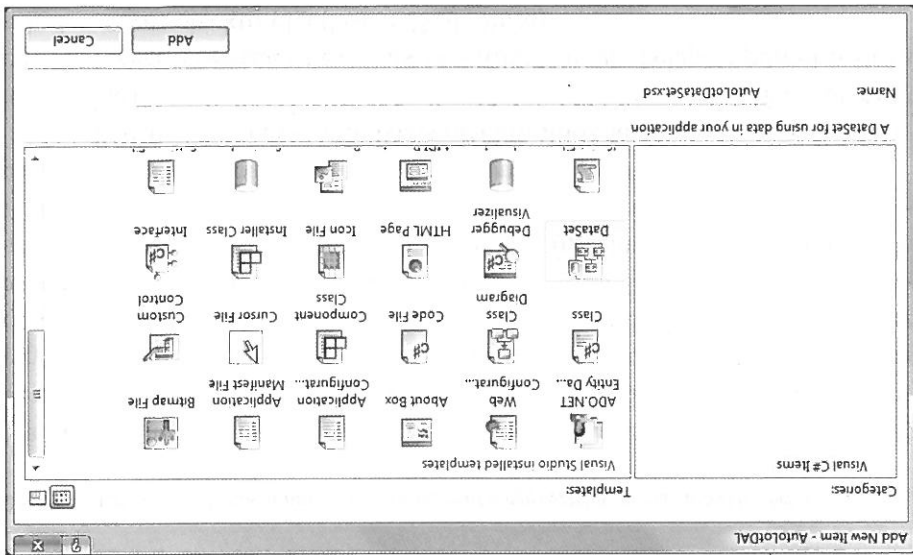
```
private void btnAddRow_Click(object sender, EventArgs e)  
{  
    // új sor beolvasása.  
    InventoryDataSet.InventoryRow newRow =  
        inventoryDataSet.Inventory.NewInventoryRow();  
    newRow.CarID = int.Parse(txtCarID.Text);  
    newRow.Make = txtMake.Text;  
    newRow.Color = txtColor.Text;  
    newRow.PetName = txtPetName.Text;  
    inventoryDataSet.Inventory.AddInventoryRow(newRow);  
  
    // Egyedi ílllesztő használataval beszzúnuk egy sort.  
    inventoryTableAdapter.Update(inventoryDataSet.Inventory);  
  
    // Újra feltöltjük a táblázatot.  
    this.inventoryTableAdapter.Fill(this.inventoryDataSet.Inventory);  
}
```

Forráskód A VisualDataGridViewApp kódfájlokat a forráskódkönyvtár 23. fejezetének al-könyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Az automatikusan generált kód elválasztása a felhasználófelület-rétegről

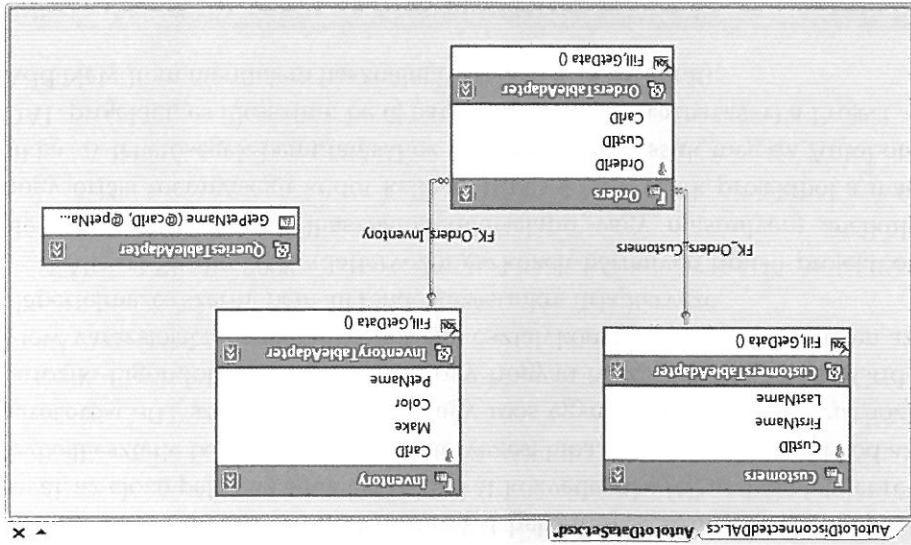
Végül mutassunk rá, hogy míg a DataGridView által elindított Data Source Configuration Wizard elvégezte a munkát helyettünk rengeteg kód létrehozásával, az előző példa az adatelérési logikát közvetlenül a felhasználófelület-rétegbe illesztette be – ez pedig súlyos tervezési hiba. Ideális esetben ez a kód az AutoLoDaL.d11 szerelvényünkhöz (vagy más adatelérési könyvtárhoz) tartozik. Elgondolkozhatunk azon, hogy hogyan használhatjuk a DataGridView varázslójában generált kódot egy osztálykönyvtár-projektben, hiszen az alapértelmezés szerint nem áll rendelkezésünkre urlaptervező.

A Visual Studio 2008 adattervezői eszközeit bármilyen típusú projektből elérhetjük (legyen az felhasználófelület-alapú vagy másmilyen) anélkül, hogy óriási mennyiségű kódot kellene átmásolnunk egyik projektből a másikba. A lehetőségek bemutatásához még egyszer nyissuk meg az AutoLoDAL projektet, és illesszünk be új DataSet típust (AutoLoDataSet) a Project > Add New Item menüelem használatával (lásd a 23.28. ábrát).



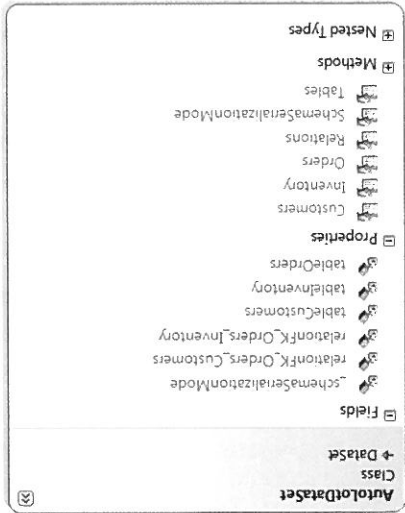
23.28. ábra: Új DataSet beszúrása

Ekkor megnyílik egy üres Dataset Designer felület. A Server Explorer segítségével kapcsolódjunk egy adott adatbázishoz (elvileg már kapcsolódunk az Autolothoz), és húzzunk át minden adatbázis-objektumot (most ne törődjünk a CreditRisk táblával), amelyet a felületen szeretnénk generálni. A 23.29. ábrán láthatjuk az Autolot minden olyan egyedi aspektusát, amelyről most gondoskodunk.

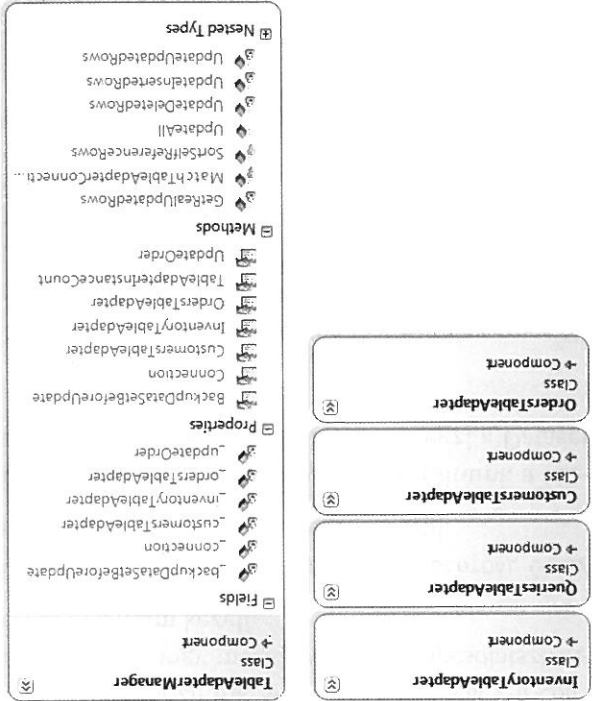


23.29. ábra: Az egyedi erősen típusos típusaink, ezúttal egy osztálykönyvtár-projektben belül

Há rápillantunk a létrehozott kódra, erősen típusos dataset, datacube és datarow típusok egy új köteget láthatjuk, valamint minden egyes táblához egy egyedi adattípusú objektumot. Mivel az autolotdataset típus tartalmaz olyan kódot, amely feltölti és frissíti az Autolot adatbázis minden tábláját, az automatikusan generált kód mennyisége több mint 3000 sor. Ennek nagy része azonban csak infrastruktúra, amelyet figyelmen kívül hagyhatunk. Ahogy a 23.30. ábrán látható, az autolotdataset típus az előző inventorydataset típus létrehozásához nagyon hasonlóan készült. Emellett minden egyes adatbázis-objektumhoz található egyedi adattípusú objektumot, amelyet áthúztunk a Dataset Designer felületére, valamint egy TableAdapterManager nevű hasznos típust, amely belépési pontot biztosít minden egyes objektumba (lásd a 23.31. ábrát).



23.30. ábra: Az *AutoloDataSet*



23.31. ábra: Az automatikusan generált adatillesztő objektumok

Forráskód Az AutoloDAL (Part 3) kódrájlokat a forráskódkönyvtár 23. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Egy felhasználói felület front end: újra a MultitableDataSetApp

Az automatikusan generált típusok használata meglehetősen egyszerű, ha rutinosan dolgozunk a kapcsolat nélküli modellel. A szöveghez letölthető forráskód tartalmaz egy MultitableDataSetApp-Redux projektet, amely – ahogy a neve is mutatja – a fejezet során korábban létrehozott MultitableDataSetApp projekt egyik verziója.

Az eredeti példa gyengén típusos dataset és sqldataadapter típusokat használ, hogy adatokat mozgasson a táblázatok között. Ez a verzió az autotdaltal, a 17 harmadik iterációját és a varázsló által generált típusokat használja. Bár most nem foglalkozunk azzal, hogy megnevezzük a teljes kódot (mivel többé-kevésbé ugyanaz, mint a projekt első iterációja), néhány fontos megjegyzést azonban érdemes tenni:

- Többé nem kell manuálisan létrehozniunk App.config fájlt vagy a configurationManagert használniunk, hogy megszerezzük a kapcsolatsztinget, ugyanis ezt a settings objektum kezeli.
- Most már erősen típusos osztályokat használunk az autotdaltal és az autolotdal.AutoLotDataSetTableAdapters névtérben belül.
- Többé nem kell manuálisan létrehozniunk vagy konfigurálniunk a tábláink közötti kapcsolatot, ezt ugyanis automatikusan elvégzi a DataSet Designer.

Az utolsó ponttal kapcsolatban figyelni kell arra, hogy a DataSet Designer által a táblakapcsolatoknak adott nevek *különböznek* azoktól, amelyeket mi adtunk a projekt első iterációjában. Ezért a `btngeneratorInfo_Click()` metódust módosítani kell, hogy a helyes kapcsolatlanveket (amelyeket a DataSet Designer tervezői felületén láthatunk) használja, például:

```
private void btngeneratorInfo_Click(object sender,
    System.EventArgs e)
{
    ...
    // Frissíteni kell a kapcsolatot nevet.
    drsorder =
    drscust[0].GetChildRows(
    autolotds.Relations["FK_orders_Customers"]);
    ...
}
```



```
// Frissíteni kell a kapcsolát nevet.  
DataRow[] drsinv =  
    drsororder[0].getParentRows(  
        autolotbs.Relations["FK_Orders-Inventory"]);  
...  
}
```

Forráskód A MultitabbedDataSetApp-Redux kódfrágmenteket a forráskódkönyvtár 23. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Összefoglalás

Ebben a fejezetben az ADO.NET kapcsolat nélküli modelljének részleteivel foglalkoztunk, amelynek központi eleme a dataset. Ez a típus tetszőleges számú tábla, valamint tetszőleges számú opcionális kapcsolat, korlátozás és kifejezés memóriabeli megjelenítésére szolgál. A helyi táblák közötti kapcsolatok kiépítésének előnye az, hogy programozottan navigálhatunk közöttük, miközben azok nem kapcsolódnak a távoli adattárhoz.

A fejezetben megvizsgáltuk az adattílesztő típus szerepét. Ennek a típusnak (és a kapcsolódó selectCommand, insertCommand, updateCommand és deleteCommand tulajdonságoknak) a használataival az illesztő végrehajthatja a dataset módosításait az eredeti adattárban is. Emellett megtanultuk, hogyan navigáljuk egy dataset objektummodelljében manuálisan, valamint erősen típusos objektumok révén, amelyeket főként a Visual Studio 2008 Dataset Designer eszközeivel generáltunk.

A LINQ API programozása

Ebben a fejezetben elkezdjük a *LINQ to ADO.NET* vizsgálatát. Ezt a kifejezést LINQ programozási modellhez kapcsolódó két jellemző leírására használjuk, a LINQ to DataSet és a LINQ to SQL lehetőségekhez. Ezek az API-k teszik lehetővé a LINQ-lekérdezések végrehajtását relációs adatbázisokban, illetve ADO.NET dataset objektumokon. Ezután a LINQ to XML szerepét tekintjük át. A LINQ ezen aspektusa nemcsak azt teszi lehetővé, hogy a lekérdezésoptimátorok készletének használatával adatokat nyerjünk ki XML-dokumentumból, hanem azt is, hogy lényegre törően (sokkal hatékonyabban, mintha a *system.xml.dll* szerelvénybe csomagolt típusokat használná) töltsünk be, mentstünk és hozzunk létre XML-dokumentumokat.

Megjegyzés Ez a fejezet az első kötet 14. fejezetében leírt LINQ programozási modellben való jártasságot feltételezi.

A LINQ to ADO.NET szerepe

A LINQ programozási modell (lásd az előző kötet 14. fejezetét is) lehetővé teszi a programozók számára, hogy olyan erősen típusos lekérdezéskifejezéseket hozzanak létre, amelyeket az adattárolók szűres körére lehet alkalmazni (tömbök, gyűjtemények, adatbázisok, XML-dokumentumok). Függetlenül attól, hogy milyen objektumon hajtjuk végre a LINQ-lekérdezést, mindig lehet ugyanazokat a lekérdezésoptimatorokat használni, és a LINQ to ADO.NET API néhány további típust és infrastruktúrát is biztosít a LINQ-adatbázis integrációjához.

A LINQ to ADO.NET olyan gyűjtőfogalom, amely a LINQ két adatbázis-központú szempontját írja le. Az egyik a LINQ to DataSet. Ez az API lényegében bővítménykészlet a szabványos ADO.NET DataSet programozási modellhez, amely lehetővé teszi, hogy a DataSet, a DataTable és a DataRow típusokon is lehessen LINQ-lekérdezéskifejezéseket végrehajtani. A *system.core.dll* típusának használatán túl a LINQ to DataSet megköveteli, hogy a projekt használja a *system.data.dataextensions.dll* szerelvény kiegészítőit.

A LINQ to ADO.NET másik komponense a LINQ to SQL. Ez az API lehetővé teszi, hogy relációs adatbázisokkal működjünk együtt, mert elvonalkoztat az alapul szolgáló ADO.NET-adattípusoktól (kapcsolatok, paramcsok, adattípusok stb.) az *entitásosztályok* használatára. Ezen az entitásosztályokon keresztül ábrázolhatunk relációs adatokat egy intuitív objektummodellel, valamint manipulálhatjuk az adatokat a LINQ-lekérdezések segítségével. A LINQ to SQL funkcionális a `System.Data.Linq.dll` szerelvényben található.

Megjegyzés A .NET 3.5 óta a LINQ to SQL nem támogatja az `data provider factory` modellel (lásd a 22. fejezetet). Ezért amikor ezt az API-t használjuk, az adatoknak a Microsoft SQL Serveren kell lenniük. Ezzel szemben a LINQ to DataSet API természete „szabadosabb”, ugyanis a kezelt dataset bármilyen relációs adatbázisból származhat.

Programozás a LINQ to DataSettel

Az előző fejezetben láthattuk, hogy a dataset típus a kapcsolat nélküli modell központi eleme, és arra használható, hogy jelképezze a kapcsolódó data-table objektumok gyorstítozárát másolatát, valamint (esetlegesen) a közöttük lévő kapcsolatot. Ehhez kapcsolódó megjegyzés, hogy egy dataset adatait három különböző módon lehet manipulálni:

- indexelőkkkel,
- adattáblázat-olvasókkal,
- erősen típusos adattagokkal.

Amikor a dataset és datatable típusok különböző indexelőit használjuk, akkor meg lehetőségen egyértelműen, de nagyon gyakran típizáltan dolgozhatunk a tárolt adatokkal. Ez a megközelítés azt követeli meg, hogy az adatokat cellák táblázatos blokkjaként kezeljük. Például:

```
static void PrintDataWithIndices(DataTable dt)
{
    // Kírtja a datatable-t.
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
        for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
        {
            Console.WriteLine(dt.Rows[curRow][curCol].ToString() + "\\t");
        }
    }
}
```

```

    }
    }
    console.WriteLine();

```

A `DataTable` típus `CreateDataReader()` metódusa egy olyan másik megközelítést kínál, amely szerint a `DataSet` adatait úgy kezelhetjük, mint sorok lineáris készletének szekvenciális feldolgozását:

```

static void PrintDataWithDataReader(DataTable dt)
{
    // Beolvassuk a DataTableReader típust.
    DataTableReader dtreader = dt.CreateDataReader();
    while (dtreader.Read())
    {
        for (int i = 0; i < dtreader.FieldCount; i++)
        {
            console.WriteLine("{0}\t", dtreader.GetValue(i));
        }
        console.WriteLine();
    }
    dtreader.Close();
}

```

Végül az erősen típusos `DataSet` lehetővé teszi, hogy az objektumban lévő adatokat olyan tulajdonságok használatával kezeljük, amelyek leképezhetők a relációs adatbázis tényleges oszlopneveire.

Erősen típusos objektumok használatával írhatunk olyan kódokat, mint például a következő (lásd a 23. fejezet):

```

static void AddRowWithTypedDataSet()
{
    InventoryTableAdapter invDA = new InventoryTableAdapter();
    AutoLoadDataSet.InventoryDataTable inv = invDA.GetData();
    inv.AddInventoryRow(999, "Ford", "Yellow", "Sai");
    invDA.Update(inv);
}

```

Ezen túlmenően a `LINQ to DataSet` még egy lehetőséget biztosít a tárolt adatok kezelésére a `LINQ`-lekérdezőkifejezések használatával. Az `ADO.NET` `DataSet`-be (és az olyan kapcsolódó típusokba, mint a `DataTable` és `DataRow`) alapvetően nincsen integrálva a szükséges infrastruktúra ahhoz, hogy közvetlenül végrehajtsunk rajta egy `LINQ`-lekérdezt. A következő metódus például fordítási idejű hibát eredményezne:

```

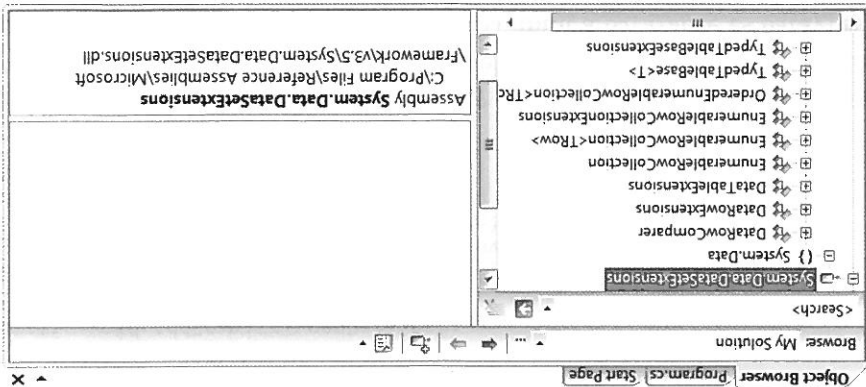
static void LinqOverDataTable()
{
    // Adatok DataTable típusának beolvasása.
    InventoryDALDslLayer dat = new InventoryDALDslLayer(
        @"Data Source=(local)\SQLEXPRESS;" +
        "Initial Catalog=Autolo;Integrated Security=True");
    DataTable data = dat.GetAllInventory();
    // Az 5-nél nagyobb carid-vel rendelkező típusok beolvasása?
    var moreData = from c in data where (int)c["CarID"] > 5 select c;
}

```

Ha lefordítanánk a `LinqOverDataTable()` metódust, akkor a fordító arról tájékoztatna, hogy a `DataTable` típus biztosít egy „lekérdezésiminta-implementációt”. Hasonlóan az `IEnumerable<T>` típust (pl. az `ArrayList`-et) nem implementáló objektumokra vonatkozó LINQ-lekérdezések alkalmazásához, az ADO.NET-objektumokat pedig kompatibilis típusú kell alakítani. Hogy megértjük ennek mikéntjét, meg kell vizsgálnunk a `System.Data.DataTableExtensions`.dll típusait.

A Dataset bővítmények szerepe

A `System.Data.DataTableExtensions.dll` több új taggal bővíti a `System.Data` névteret (lásd a 24.1. ábrát).



24.1. ábra: A `System.Data.DataTableExtensions.dll` szerelvény

A két leghasznosabb tag vitathatatlanul a `DataTableExtensions` és a `DataRowExtensions`. Ahogy a nevük is mutatja, ezek a típusok a `DataTable` és a `DataRow` típusok működését bővíto metódusok készletével egészítik ki. Egy másik kulcsfontosságú típus a `TypedTableBaseExtensions`, amely olyan bővíto metó-

dusokat definiál, amelyeket erősen típusos dataset objektumokra lehet alkalmazni azért, hogy a belső datatable objektumok LINQ-kompatibilitásuk legyenek. A `System.Data.DataSetExtensions`.d11 szerelvény összes többi tagja pusz-
tán az infrastruktúrát szolgálja ki, és nem közvetlen használatra szánják őket.

DataTable LINQ-kompatibilitás használata

A dataset bővítmények használatának bemutatásához töltsük fel, hogy van egy új C# parancssori konzolalkalmazásunk `LinqOverDataSet` névvel. Fontos megjegyezni, hogy amikor a .NET 3.5-öt célzó projekteket készítünk, akkor automatikusan referenciát kapunk a `System.Core`.d11 és `System.Data.DataSetExtensions`.d11 szerelvényekre; ám ebben a példában adjunk hozzá még egy szerelvényreferenciát az `AutoloDAL`.d11 szerelvényre (lásd a 23. fe-
jezetet), és egészítsük ki a kezdeti kódfájlt a következő logikával:

```
using System.Data;
using AutoloDAL.ConnectedLayer;

namespace LinqOverDataSet
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** LINQ over DataSet *****\n");
            // Beolvassuk az Autolo adatbázis Inventory
            // táblájának datatable típusát.
            InventoryDAL layer = new InventoryDAL(layer.Initial
            @Data Source=(local)\SQLEXPRESS;Initial
            Catalog=Autolo;" + "Integrated Security=True");
            datatable data = dal.GetAllInventory();

            // Hívjuk meg az itt következő metódusokat.
            Console.ReadLine();
        }
    }
}
```

Amikor LINQ-kompatibilitás objektumná szeretnénk átalakítani egy `ADO.NET` datatable típust, akkor egyszerűen hívjuk meg az `AsEnumerable()` bőví-
tő metódust, amelyet a `datatableextensions` típus definiál. Ez visszaad egy `Enume-
rableCollection` objektumot, amely egy `DataRow` gyűjteményt tartalmaz. Az `EnumerableCollection` típus használatával az elvártak szerint dolgozhatunk
az egyes sorokkal. Nézzünk egy egyszerű példát:

```

static void PrintAllCars(DataTable data)
{
    // Beolvassuk a DataTable feloszorított változatát.
    Enumerablerowcollection enumeration = data.Asenumerable();

    // Kijelöljük az autoazonosító értékeket.
    foreach (DataRow r in enumeration)
        Console.WriteLine("Car ID = {0}", r["CarID"]);
}

```

Mivel az `Enumerablerowcollection` megvalósítja az `Ienumerable<T>` típust, ezért az is megengedett, hogy az alábbi utasítások egyikével kapjuk meg a visszatérési értéket:

```

// A visszatérési érték tárolása Ienumerable<T> típusként.
Ienumerable<DataRow> enumeration = data.Asenumerable();

// A visszatérési értékek tárolása implicit módon.
var enumeration = data.Asenumerable();

```

Eddig ugyan még nem futtattuk le a LINQ-lekérdezést, ám az `enumdata` objektumon immár végrehajthatunk LINQ-lekérdezéskifejezést. Az `enumerablerowcollection` valóban datarow objektumok gyűjteményét tartalmazza, ezért egy típusindexelőt alkalmazunk az összes alobjektumra, hogy kiterassuk a `carid` oszlop értékét.

A legtöbb esetben nem kell `enumerablerowcollection` típusú változót deklarálnunk azért, hogy tárolja az `Asenumerable()` visszatérési értéket. Ehelyett hívjuk meg ezt a metódust magából a lekérdezéskifejezésből. Akad egy még érdekesebb metódus is, amely visszaadja a `carid`/makes oszlopok lekérdezését a `datatable` összes bejegyzéséből, ahol a `carid` értéke nagyobb mint 5:

```

static void ApplyLinqQuery(DataTable data)
{
    // Képezzünk le egy új eredményhalmazt, amely tartalmazza
    // az ID/color értékeket az 5-nél nagyobb carid-vel rendelkező
    // sorok esetében.
    var cars = from car in data.Asenumerable()
               where
                   (int)car["CarID"] > 5
               select new
               {
                   ID = (int)car["CarID"],
                   color = (string)car["Color"]
               };
}

```



```

Console.WriteLine("Cars with ID greater than 5:");
foreach (var item in cars)
{
    Console.WriteLine("-> CarID = {0} is {1}", item.ID, item.Color);
}
}

```

A DataRowExtensions.Field<T>() bővítő metódus szerepe

A jelenlegi LINQ-lekérdezéskifejezésben hátrányos lehet az, hogy számos kasztolási műveletet és datarow indexelést használunk az eredményhalmaz összeállításához, és ez futásidéjű kivételt eredményezhet, ha nem kompatibilis adat típusra próbálunk kasztolni. Ahhoz, hogy erős tipizálást alkalmazzunk a lekérdezésünkben, használhatjuk a `datarow típus field<T>()` bővítő metódusát. Így megkönnyítjük a lekérdezés típusbiztonságát, ugyanis az adat-típusok kompatibilizálását a rendszer a fordítási időben ellenőrzi. Nézzük meg a következőképpen módosított kódot:

```

var cars = from car in data.AsEnumerable()
            where
                car.Field<int>("CarID") > 5
            select new
            {
                ID = car.Field<int>("CarID"),
                Color = car.Field<string>("Color")
            };

```

Látható, hogy ebben az esetben meghívhatjuk a `field<T>()` metódust, és megadhatunk egy típusparamétert, amely az oszlop alapítusát határozza meg. A metódus argumentumaként átadjuk magát az oszlopnevet. A fordítási időjű ellenőrzés miatt tanácsos a `field<T>()` metódust használni a datarow indexelője helyett `enumerablecollection` feldolgozásakor. Azon túl, hogy meghívjuk az `asenumerable()` metódust, a LINQ-lekérdezés általános formátuma hasonló ahhoz, amelyet már láttunk az előző kötet 14. fejezetében. Ezért most nem térünk ki újra a különböző LINQ-operátorok részleteire. Ha további példákra vagyunk kíváncsiak, nézzük át a „LINQ to DataSet Examples” témakört a .NET Framework 3.5 SDK dokumentációjában.

Új DataTable objektumok feltöltése a LINQ-lekérdezésekből

Könnyedén fel lehet tölteni egy új DataTable adatait egy LINQ-lekérdezés alapján, feltéve, hogy *nem* használunk lekérdezéseket. Amikor van egy olyan eredményhalmazunk, amelyben az alapul szolgáló típus `IDisposable`-t implementál, megkérdezzük, hogy lehet-e a `CopyToDataTable()` metódust az eredményen. Például:

```
static void BuildDataTableFromQuery(DataTable data)
{
    var cars = from car in data.AsEnumerable()
               where
                   car.Field<int>("CarID") > 5
    select car;

    // Használjuk ezt az eredményhalmazt új DataTable létrehozásához.
    DataTable newTable = cars.CopyToDataTable();

    // Készítsük a DataTable-t.
    for (int curRow = 0; curRow < newTable.Rows.Count; curRow++)
    {
        for (int curCol = 0; curCol < newTable.Columns.Count; curCol++)
        {
            Console.WriteLine(
                newTable.Rows[curRow][curCol].ToString() + "\t");
        }
        Console.WriteLine();
    }
}
```

Megjegyzés Az `AsDataView()` metódussal is átalakíthatunk egy LINQ-lekérdezést `DataView` típusúvá.

Ez a megközelítés akkor lehet nagyon hasznos, ha a LINQ-lekérdezés eredményét adatkötés során forrásként szeretnénk használni. A Windows Forms `DataGridView` (valamint az `ASP.NET GridView`) vezérlőelem például támogatja a `DataSource` tulajdonságot. LINQ-eredményt az alábbiak szerint köthetünk egy `DataGridView`-hez:

```
// Tegyük fel, hogy a myDataGrid GUI-alapú táblázatobjektum.
myDataGrid.DataSource = (from car in data.AsEnumerable()
                          where
                              car.Field<int>("CarID") > 5
                          select car).CopyToDataTable();
```

A következőkben térünk át a LINQ to SQL-re.

Programozás a LINQ to SQL használatával

A LINQ to SQL olyan API, amely lehetővé teszi jól megformázott LINQ-lekérdezéskifejezések alkalmazását relációs adatbázisokban lévő adatokra. A LINQ to SQL több olyan típust kínál (a `system.data.linq.d11` szerezélyenben), amelyek megkönnyítik a kommunikációt a kód és a fizikailagadatbázis-motor között.

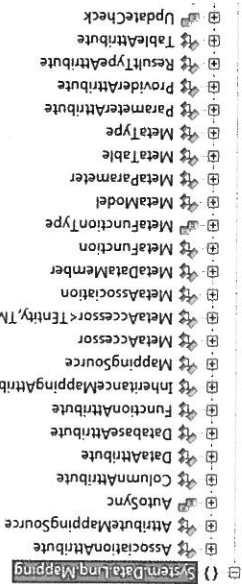
A LINQ to SQL legfőbb célja az, hogy konzisztenciát biztosítson a relációs adatbázisok és a velük folytatott munkára használt programozási logika számára. Ahelyett például, hogy az adatbázis-lekérdezéseket nagy sztringekkel jellepezzünk, használhatunk erősen típusos LINQ-lekérdezéseket. Továbbá ahelyett, hogy a relációs adatokat rekordok folyamanként kellene kezelnünk, dolgozhatunk az adatokkal szabványos objektumorientált programozási módszerekkel. Mivel a LINQ to SQL lehetővé teszi az adatelérés integrálását közvetlenül a C#-forráskódban, nagymértékben csökkén annak a szükségessége, hogy manuálisan hozzuk létre egyedi osztályok tucatjait, valamint azokat az adatelérési könyvtárakat, amelyek elrejtik szem elől az ADO.NET csúnya kódjait.

Amikor LINQ to SQL felhasználásával programozunk, nyoma sincs az olyan megszokott ADO.NET-típusoknak, mint az `sqlconnection`, az `sqlcommand` vagy az `sqlDataAdapter`. A LINQ-lekérdezéskifejezések, a (hamarosan ismertendő) entitásszállyok és a `datacontext` típus segítségével végrehajthatjuk a kívánt adatbázis-műveleteket (letrhozás, eltávolítás, módosítás és törés), valamint definiálhatunk tranzakciós kontextusokat, létrehozhatunk új adatbázis-entitásokat (vagy egész adatbázisokat), meghívhatunk tárolt eljárásokat, és elvégezhetünk egyéb adatbázis-centrikus tevékenységeket. Ezenfelül a LINQ to SQL típusok (példaként megint a `datacontext` említhetjük) feladata, hogy megvalósítsák a standard ADO.NET-adattípusok integrációját. A `datacontext` egyik túlterhelt konstruktora bemennetként például egy `idbconnection` objektumot vár, amely az összes ADO.NET-kapcsolatobjektum által támogatott közös interfész. Ilyen módon a már meglevő ADO.NET adatelérési könyvtárak integrálhatók a C# 2008 LINQ-lekérdezéskifejezésekbe (és fordítva). Valójában – ami a Microsoftot illeti – a LINQ to SQL egy szerte az ADO.NET-család egyik új tagja.

Az entitásosztályok szerepe

Ha a LINQ to SQL-t szeretnénk használni az alkalmazásainkban, akkor az első lépés az *entitásosztályok* definiálása. Röviden: az entitásosztályok olyan típusok, amelyek azokat a relációs adatokat jelképezik, amelyekkel dolgozni szeretnénk. Programozási szempontból az entitásosztályok olyan osztálydefiniciók, amelyeket különböző LINQ to SQL attribútumokkal (például [Table] és [Column]) jelölünk, amelyek leképezzhetők fizikai táblákra egy adott adatbázisban. A LINQ to SQL attribútumok többségének definícióját a system.data.Linq.Mapping névtér foglalja magában (lásd a 24.2. ábrát).

Ahogy azt hamarosan látni fogjuk, a .NET Framework 3.5 SDK (valamint a Visual Studio 2008) olyan eszközöket biztosít, amelyek automatizálják az alkalmazás számára szükséges entitás típusok elkészítését. Ezt megelőzően az első LINQ to SQL példánk fogja bemutatni, hogy hogyan hozzuk létre entitásosztályokat.



24.2. ábra: A System.Linq.Mapping névtér számos LINQ to SQL attribútumot definiál

A DataContext típus szerepe

Ha meghatározzuk az entitásosztályokat, akkor a DataContext típussal átadhatjuk a lekérdezési kifejezéseket a relációs adatbázisnak. Ez a LINQ to SQL-specifikus osztálytípus felelős a LINQ-lekérdezéskifejezések megfelelő SQL-lekérde-

zésekre történő fordításáért, valamint az adott adatbázissal történő kommunikációért. Bizonyos tekintetben a DataContext hasonló egy ADO.NET-kapcsolat-objektumra: szükség van egy kapcsolatsztringre. A tipikus kapcsolatotobjektummal ellentétben azonban a DataContext típus számos olyan taggal rendelkezik, amelyek a lekérdezéskifejezések eredményeit az általunk meghatározott entitásosztályokba képezik le.

Továbbá a DataContext típus egy factoryminitát definíál a kódban használt entitásosztályok példányainak megszerzésére. Ha megszerztünk egy entitáspéldányt, akkor a kívánt módon változtathatjuk meg annak állapotát (re-kordok hozzáadása és frissítése stb.), és a módosított objektumot feldolgozásra visszaküldhetjük. Így módon a DataContext hasonló az ADO.NET adatillesztő típushoz.

Egy egyszerű LINQ to SQL példa

Először nézzük meg a LINQ to SQL használatának egy egyszerű példáját, amely a 22. fejezetben létrehozott AutoLot adatbázis Inventory táblájával dolgozik. Ebben a példában nem használjuk az AutoLotDAL.d11 könyvtárat, hanem minden kódot manuálisan készítünk el. Hozzunk létre egy új páranccsori konzolalkalmazást SimpleLINQtoSQLapp névvel, valamint adjunk egy referenciát a System.Data.Linq.d11 szerezvényre.

Ezután adjunk hozzá egy új C#-osztályfájl (Inventory.cs). Ez a fájl fogja definiálni az entitásosztályunkat, amely megköveteli a típus kiegészítését különböző LINQ-központú attribútumokkal; ezért mindenképpen adjuk meg, hogy a System.Data.Linq.Mapping és System.Data.Linq.névtereket használjuk. A következőkben nézzük meg az Inventory típus definícióját:

```
[Table]
public class Inventory
{
    [Column]
    public string Make;
    [Column]
    public string Color;
    [Column]
    public string PetName;
    // Az elsődleges kulcs azonosítása.
    [Column(IsPrimaryKey = true)]
    public int CarID;
```

```

public override string ToString()
{
    return string.Format("ID = {0}; Make = {1}; Color = {2};
        PetName = {3}", carID, Make.Trim(),
        color.Trim(), PetName.Trim());
}
}

```

Az entitásosztályunk megkapta a [Table] attribútumot, amíg a nyilvános mezők mindegyikét megjelöltük a [Column] attribútummal. A nevek mindkét esetben közvetlen leképezései a fizikai adatbázistáblának. Ez azonban nem szigorú követelmény, hiszen a tableattribute és a columnattribute típusok támogatják a Name tulajdonságot, amely lehetővé teszi az adattábla programozott reprezentációjának elkülönítését magától a fizikai táblától. A carID mezőt tovább minősítettük a columnattribute típus IsPrimaryKey tulajdonságának beállításával, a megnevezett tulajdonságszintaxis segítségével.

Az egyszerűség kedvéért minden mezőt publikusan deklaráltunk. Ha erősebb egyébszerezésre van szükségünk, akkor definiálhatunk nyilvános tulajdonságokba (vagy akár automatikus tulajdonságokba) csomagolt privát mezőket. Ekkor nem a mezők, hanem a *tulajdonságok* lesznek a [Column] attribútummal megjelölve.

Egy entitásosztály bármennyi olyan tagot tartalmazhat, amelyek nem képezhetők le a reprezentált adattáblára. Ami a LINQ-futásidő ill. a rendszer csak a LINQ to SQL attribútumokkal megjelölt elemeket használja az adatcsere során. Ez az inventory osztálydefinició például a ToString() egyedi megvalósítását biztosítja ahhoz, hogy lehetővé tegye az alkalmazás számára az állapot gyors megjelölését.

Ha már rendelkezünk entitásosztállyal, használhatjuk a DataContext típust, hogy elküldjük (és lefordítsuk) a LINQ-lekérdezéskifejezéseinket a megadott adatbázisnak. Tanulmányozzuk az alábbi Main() metódust, amely megjeleníti az AutoLot adatbázis Inventory táblájában lévő összes elemet:

```

class Program
{
    const string cnstr =
        @"Data Source=(local)\SQLEXPRESS;Initial Catalog=AutoLot;" +
        "Integrated Security=True";
    static void Main(string[] args)
    {
        Console.WriteLine("***** LINQ to SQL Sample App *****\n");
        // DataContext objektum létrehozása.
        DataContext db = new DataContext(cnstr);
    }
}

```

```
// Most hozzunk létre egy Table<> típust.
Table<Inventory> invTable = db.GetTable<Inventory>();

// Az adatok megjelenítése LINQ-lekérdezés segítségével.
Console.WriteLine("> Contents of Inventory table from Autolot
database:\n");
foreach (var car in from c in invTable select c)
    Console.WriteLine(car.ToString());
Console.ReadLine();
}
```

Amikor létrehozunk egy DataContext típust, meg kell adnunk a megfelelő kapcsolatsztringet, amely itt egyszerű sztringkonstansként jelenik meg. Ezt nyugodtan eltávolíthatjuk az alkalmazáskonfigurációs fájlban, és/vagy használhatjuk az SqlConnectionStringBuilder típust az objektumorientáltabb kézeles érdekében.

Ezután megszerezzük az Inventory entitásosztályunk egy példányát a DataContext típus generikus GetTable<T>() módszerével, ahol az entitásosztályt adjuk meg típusparaméterként. Végezetül létrehozunk egy LINQ-lekérdezéskifejezést, és végrehajtuk azt az invTable objektumon. A végeredmény az Inventory tábla minden elemének megjelenítése lesz.

Erősen típusos DataContext létrehozása

Míg első példánk erősen típusos volt az adatbázis-lekérdezés szempontjából, némileg elkülönítettük a DataContext típust és az általa fenntartott Inventory entitásosztályt. Ennek orvoslásához általában érdemes létrehozunk egy olyan osztályt, amely kibővíti a kezelt táblák mindegyike számára tagváltozókat definiáló DataContext típust. Illesztünk be egy új osztályt AutolotData-Base névvel, adjuk meg, hogy a System.Core és a System.Data.Linq névtérket használjuk, majd implementáljuk a típust az alábbiak szerint:

```
class AutolotDatabase : DataContext
{
    public Table<Inventory> Inventory;

    public AutolotDatabase(string connectionString)
        : base(connectionString)
    {
    }
}
```

Az új osztálytípussal most már egyszerűsíthetjük a `main()` metódus kódját:

```
static void main(string[] args)
{
    Console.WriteLine("***** LINQ to SQL Sample App *****\n");

    // Hozzunk létre egy AutolotDatabase objektumot.
    AutolotDatabase db = new AutolotDatabase(cnst);

    // Most már használhatjuk az AutolotDatabase Inventory mezőjét.
    Console.WriteLine(">-> Contents of Inventory Table from Autolot
    database:\n");
    foreach (var car in from c in db.Inventory select c)
    {
        Console.WriteLine(car.ToString());
    }
}
```

Az erősen típusos adatkönyvezet létrehozásának egyik meglepő jellemzője, hogy a `DataContext`-ból származó típus (jelen példában az `AutolotDatabase`) közvetlenül nem hozza létre a `Table<T>` tagválasztókat, és nincs nyoma a várt `getTable()` metódushívásnak. A fordítás során azonban, amikor végigítérünk a LINQ-eredményhalmazon, a `DataContext` a háttérben létrehozza a `Table<T>` típust.

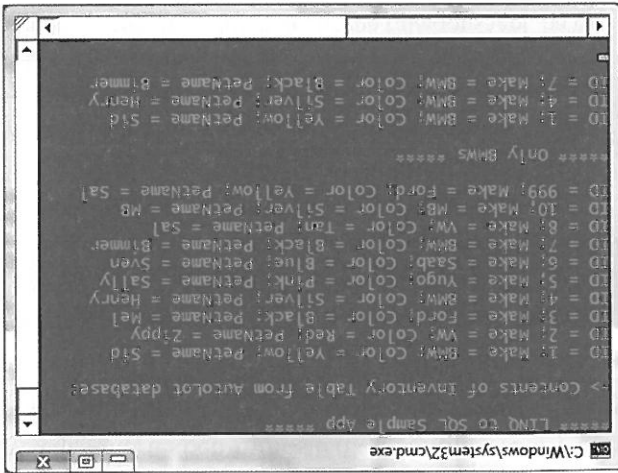
A LINQ-lekérdezések természetesen használhatók adott eredményhalmaz megszerzésére is. Tegyük fel, hogy létrehoztunk a következő segédmetódust, amelyet kilépés előtt meghívunk a `main()` metódusból (ez a metódus paraméterként vár egy `AutolotDatabase` példányt):

```
static void ShowOnlyBimmers(AutolotDatabase db)
{
    Console.WriteLine("***** Only BMWs *****\n");

    // Megkapjuk a BMW-ket.
    var bimmers = from s in db.Inventory
    where s.Make == "BMW"
    orderby s.CarID
    select s;

    foreach (var c in bimmers)
    {
        Console.WriteLine(c.ToString());
    }
}
```

A 24.3. ábra mutatja az első LINQ to SQL példánk kimenetét.



24.3. ábra: Első bepillantás a LINQ to SQL használatába

Forráskód A SimpleInqToSqlApp kódrajzokat a forráskódkönyvtár 24. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A [Table] és [Column] attribútumok: további részletek

Az entitásosztályok különböző attribútumokkal rendelkeznek, amelyeket a LINQ to SQL használ, hogy le tudja fordítani az objektumokon végzett lekérdezést adatbázis SQL-lekérdezésekre. Csak a [Table] és a [Column] attribútumokat fogjuk használni; mindazonáltal további attribútumok is léteznek azoknak a metódusoknak a megjelölésére, amelyek végrehajják az SQL besztűró, módosító és törölő parancsait. Ezenfelül a LINQ to SQL attribútumok mind-egyike meghatároz egy olyan tulajdonságkészletet, amely tovább részletezi, hogy a LINQ to SQL futtatómotor hogyan dolgozza fel a megjelölt elemet.

A [Table] attribútum nagyon egyszerű, ugyanis egyetlen tulajdonságot definiál: ez a Name. Ez teszi lehetővé, hogy elkülönítsük az entitásosztály nevét a fizikai táblázattól.

Ha nem állítjuk be a Name tulajdonságot a [Table] attribútum használatakor, a LINQ to SQL feltételezi, hogy az entitásosztály és adatbázistábla nevei megegyeznek.

A [Column] attribútum kicsivel tartalmasabb, mint a [Table]. Az IsPrimary-key tulajdonságon túl a ColumnAttribute további olyan tagokat definiál, ame-

Ilyek lehetővé teszik, hogy teljes mértékben minősítsük az entitásosztály összes mezejét, és azt, hogy hogyan képezhetők le a fizikai adatbázisból egy adott oszlopra. A 24.1. táblázat az érdekesebb tulajdonságokat mutatja be.

ColumnAttribute	Jelentés
-----------------	----------

CanBeNull	Ez a tulajdonság azt jelzi, hogy az oszlop tartalmazhat null értéket.
-----------	---

DbType	Az adatmezők deklarációja alapján a LINQ to SQL automati- matikusan kikövetkezteti, milyen tulajdonságokat kell az adatbázismotornak átadni. Emiatt általában csak akkor kell közvetlenül beállítani a DbType tulajdonságot, ha di- namikusan hozunk létre adatbázisokat a DataContext fi- pus CreateDatabase() metódusának használatával.
--------	---

IsDbGenerated	A tulajdonság azt állítja be, hogy az adatbázis automati- kusan generálja egy mező értékét.
---------------	--

IsVersion	Ez a tulajdonság határozza meg, hogy az oszlop adatbázis- időbélyeg, esetleg verziószám-e, vagy sem. A verziószá- mok növekednek, az időbélyegoszlopok pedig frissülnek minden alkalommal, amikor a kapcsolódó sor módosul.
-----------	--

UpdateCheck	Ez a tulajdonság vezérli, hogy a LINQ to SQL hogyan ke- zelje az adatbázis-ütközéseket az optimista konkurencia- szabályozás révén.
-------------	---

24.1. táblázat: A [Column] attribútum néhány tulajdonsága

Entitásosztályok generálása az SqlMetal.exe használatával

Az első LINQ to SQL példánk meglehetősen egyszerű volt, részben annak a ténynek köszönhetően, hogy a DataContext típusunk egyetlen adattáblával dolgozott. Elképzelhető azonban, hogy a termékcsintű LINQ to SQL alkal-
mazások több, egymással kapcsolatos adattáblával dolgoznak, s ezek mind-
egyike oszlopok tucatjait definiálhatja. Ilyen esetekben unalmas lenne manu-
álisan létrehozni minden egyes kívánt entitásosztályt. Szerencsére kétéle-
mód is megoldhatjuk ezeket a típusoknak az automatikus generálását.

Az első lehetőség az sqlmetal.exe parancssori segédprogram használata, amelyet a Visual Studio 2008 parancssorából indíthatunk el. Ez az eszköz úgy automatizálja az entitásozstályok létrehozását, hogy generál egy megfelelő C#-osztálytípust az adatbázis metaadataiból. Bár az eszköz több parancssori paraméterrel rendelkezik, a 24.2. táblázat csak a lényegesebb kapcsolatokat mutatja be.

sqlmetal.exe parancssori paraméter	Jelentés
------------------------------------	----------

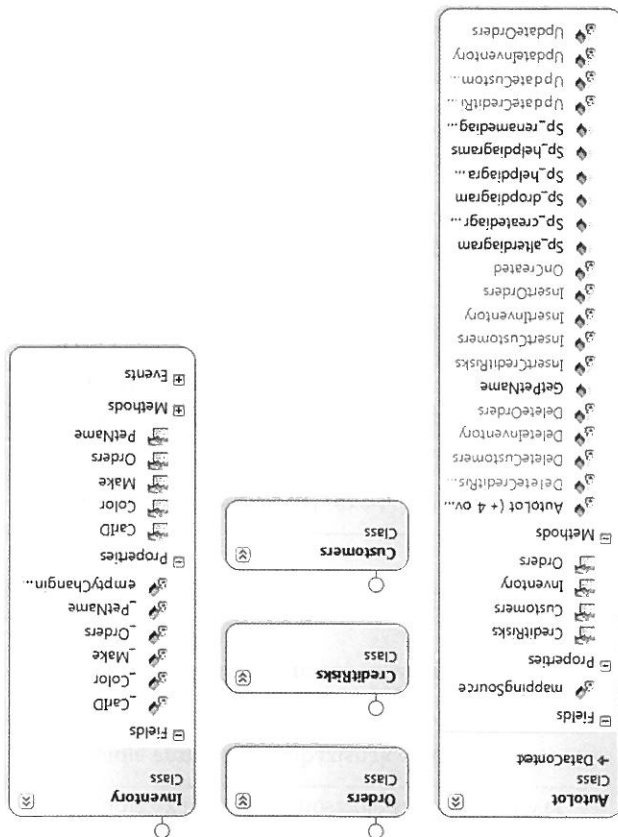
/server	Megadja az adatbázist hosztoló kiszolgáló nevét.
/database	Megadja annak az adatbázisnak a nevét, amelyből ki kell olvasni a metaadatokat.
/user	Megadja a felhasználónevet, amellyel be kell jelentkezni a kiszolgálóra.
/password	Megadja a jelszót, amellyel be kell jelentkezni a kiszolgálóra.
/views	Tájékoztatja az sqlmetal.exe-t, hogy létező adatbázisnévzetek alapján generáljon kódot.
/functions	Tájékoztatja az sqlmetal.exe-t, hogy készítse el az adatbázisfüggvények megfelelőit.
/procs	Tájékoztatja az sqlmetal.exe-t, hogy készítse el a tárolt eljárások megfelelőit.
/code	Tájékoztatja az sqlmetal.exe-t, hogy C#-kódként adja ki az eredményeket (vagy VB-kódként, ha beállítottuk a /language kapcsolót).
/language	Megadja, hogy a rendszer milyen nyelven definiálja a generált típusokat.
/namespace	Megadja névteret a generált típusok definiálásához.

24.2. táblázat: Az sqlmetal.exe parancs paramétereit

A következő utasítás például az Autolot adatbázis minden táblájához generál entitásozstályokat, metódust készít a GetPetName tárolt eljáráshoz, és az Autolotadatbázis névtérbe csomagolja a generált C#-kódot (természetesen egy sorba kell beírni a Visual Studio 2008 parancssorában):

```
sqlmeta1/server:(local)\SQL EXPRESS /database:Autolot
namespace:AutolotDatabase /code:autolotDB.cs /sprocs
```

Ha végrehajtottuk az utasítást, hozzáunk létre egy új paramcssoriatkalkalmazás-projektet `lingwtshq\metalgeneadcode` nével, illetőleg adjunk referenciát a `system.data.linq.dll` szerezélevényre, és adjuk hozzá az `autolotdb.cs` fájlt a projekthez a `Project > Add Existing Item` menüpont segítségével. Ezenfelül, szűnjünk be egy új osztálydiagramot a projektkibe (`Project > Add New Item`), és bontsuk ki a generált osztályokat (lásd a 24.4. ábrát).



24.4. ábra: Az `sqlmetal.exe` által generált entitáshozszoztályok

Van tehát egy új, a DataContextteret kibővíítő típusunk, amely tartalmazza a tulajdonságokat a megadott adatbázisban lévő összes adattábla számára (továbbá a getPetName() tárolt eljárást a megfigyző névvel rendelkező nyilvános metódus képviseli). Mielőtt elkezdzenk használni ezeket az új típusokat, vizsgáljuk meg egy kicsit alaposabban ezt az automatizkusan generált kódot.

A generált entitásosztályok

Az sq1metal.exe az AutoLot adatbázis minden táblája (Inventory, Customers, Orders, CreditRisks) számára külön entitásosztályt definiált, és minden osztópont beágyazott egy tulajdonságba. Továbbá az egyes entitásosztályok két interfészt implementálnak (INotifyPropertyChanged és INotifyProperty-changed), amelyek mindegyike egy-egy eseményt hárít meg:

```
namespace System.Data.Linq
{
    public interface INotifyPropertyChanged
    {
        // Az esemény akkor következik be, mielőtt egy tulajdonság
        // értéke megváltozik.
        event PropertyChangedEventHandler PropertyChanged;
    }
}

namespace System.ComponentModel
{
    public interface INotifyPropertyChanging
    {
        // Az esemény akkor következik be, amikor egy tulajdonság értéke
        // megváltozott.
        event PropertyChangedEventHandler PropertyChanged;
    }
}
```

Ezek az interfészek összesen két eseményt definiálnak (PropertyChanging és PropertyChanged), amelyek a System.ComponentModel névtérben meghatározott PropertyChangedEventHandler metódusreferenciával működnek együtt. Ez a metódusreferencia meghívható bármilyen metódust, amely egy objektumot vár első paraméterként, míg a második paraméter egy PropertyChangedEventArgs. A fenti interfészek miatt minden entitásosztály támogatja a következő tagokat:

```
[Table(Name="Inventory")]
public partial class Inventory : INotifyPropertyChanging,
    INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    ...
}
```

Ha megvizsgáljuk a három entitásosztály tulajdonságainak megvalósítását, láthatjuk, hogy a set hatókör kivált minden eseményt azok számára, akik a tulajdonság értékének a változását figyelik. Példaként nézzük meg az Inventory típus PetName tulajdonságát:

```

[Column(Storage="_PetName", DbType="VarChar(50)")]
public string PetName
{
    get
    {
        return this._PetName;
    }
    set
    {
        if ((this._PetName != value))
        {
            this.OnPetNameChanging(value);
            this.SendPropertyChanging();
            this._PetName = value;
            this.SendPropertyChanged("PetName");
            this.OnPetNameChanged();
        }
    }
}

```

A set hatókör meghívja az `OnPetNameChanging()` és az `OnPetNameChanged()` metódusokat az entitásosztály-típuson, hogy valójában ezek váltásák ki az eseményeket. Ezeket a tagokat azonban *reszleges metódusokként* definiáltuk, amelyek könnyű eseménykezelést hajtanak végre (lásd az előző kötet 13. fejezetét), lehetővé téve az érdekelteket hívó számára, hogy biztosítsa a szükséges megvalósítást (ha nem, akkor a fordítás során a rendszer eltávolítja őket a típusdefiniációból):

```

partial void OnPetNameChanging(string value);
partial void OnPetNameChanged();

```

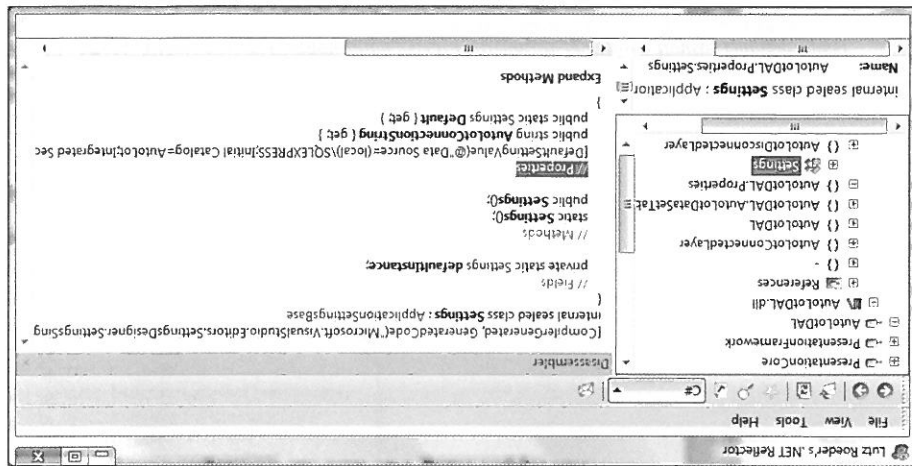
Kapcsolatok definiálása entitásosztályok használatával

Azon túl, hogy tulajdonságokat és segédmetódusokat definiál az adatbázis oszlopainak megjelenítésére, az `sqlmetal.exe` segédprogram az `EntityType>` típusal modellt az összefüggő táblázatok közötti kapcsolatokat. Az Autolót adatbázis definiált három, egymáshoz kapcsolódó belső táblát, amelyeket elsődleges és idegen kulcsok kapcsolnak össze (lásd a 22. fejezet). Ahelyett, hogy SQL-központú join szintaxis megírására kényszerülnénk a táblázatok közötti navigáláshoz, a LINQ to SQL lehetővé teszi, hogy az objektumcentrikus C# pont operátorral navigáljunk.

Egészen pontosan a megjelöltösen hosszú VisualDataGridV1EWApp.Properties.Settings.AutoLotConnectionString értéket látjuk a kapcsolatsztring nevéként. Még furcsább az a tény, hogy ha átfutjuk a generált kódot, nem találunk referenciát a ConfigurationManager típusra, hogy kiolvashassuk az értéket a <connectionstring> elemről. Azt azonban láthatjuk, hogy az automatikusan generált adatillesztő objektumot (lásd részletesen később) részben a következő zó privát segédfüggvény meghívásával hoztuk létre:

```
private void InitConnection()
{
    this._connection =
        new global::System.Data.SqlClient.SqlConnection();
    this._connection.ConnectionString = global::
        VisualDataGridV1EWApp.Properties.Settings.Default.
        AutoLotConnectionString;
}
```

A Connectionstring tulajdonságot a settings.Default meghívásával állítjuk be. Minden Visual Studio 2008 projekt típus fenntart az egész alkalmazásra kiterjedő beállításokat, amelyek metaadatként égneek bele a szerelvénybe, amikor lefordítjuk az alkalmazást. Ha megnyitunk egy lefordított alkalmazást a reflector.exe használatával (lásd az előző kötet 2. fejezetét), akkor megnevezhetjük ezt a belső típust (lásd a 23.22. ábrát).



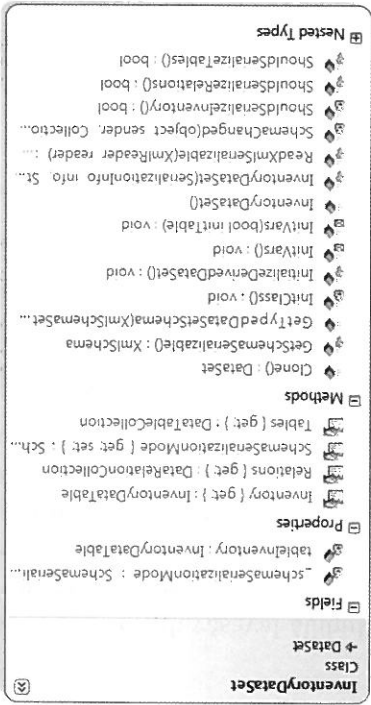
23.22. ábra: A Settings objektum tartalmaz egy beágyazott kapcsolatsztringértéket

A fentiek ismeretében lehetséges lenne úgy telepíteni az alkalmazást, hogy nem adjuk oda a *.config fájlt, mivel az alapértelmezés szerint a beágyazott értéket használjuk, ha nincsen jelen kliensoldali *.config fájl.

Megjegyzés A Visual Studio 2008 beállításaiiban a programozási modellek teljes ismertetése túlmutat a fejezet hatáskörén. Ha szeretnének többet megtudni ezekről, olvassuk el a „NET Framework 3.5 SDK dokumentációjának „Managing Application Settings” című témakörét.

A generált DataSet vizsgálata

Nézzük meg a generált kód néhány alapvető aspektusát. Először szúrjunk be egy új osztálydiagramot a projektbe, amelyhez válasszuk ki a projekt ikonját a Solution Explorerben, majd kattintsunk a View Class Diagram gombra. A be-menet alapján a varázsló létrehoz egy új dataset típust, amelynek ebben az esetben InventoryDataSet a neve. Az osztály számos hasznos tagot definiál, amelyek közül az Inventory tulajdonság a legfontosabb (lásd a 23.23. ábrát).



23.23. ábra: A Data Source Configuration Wizard létrehozott egy erősen típusos DataSet típust

Ha a Solution Explorerben kattintunk az InventoryDataSet.xsd fájlra, akkor betöltjük a Visual Studio 2008 DataSet Designer eszközt (erről részlete-gebben később lesz szó). Ha ezen a tervezőn belül bárhova kattintunk a jobb egérgombbal, és kiválasztjuk a View Code lehetőséget, megjelölésen üres részleges osztálydefiniációt láthatunk:


```
public partial class InventoryDataSet {
    {
    }
}
```

Az események a tervező által karbantartott fájlon (InventoryDataSet.Designer.cs) belül zajlanak. Ha a Solution Explorerrel megnyitjuk ezt a fájlt, látható, hogy az InventoryDataSet valójában kibővíti a dataset osztálytípust. Amikor mi (vagy a varázsló) létrehozunk egy olyan osztályt, amely kibővíti a dataset típust, akkor *erősen típusos* datasetet valósítunk meg. Az erősen típusos dataset objektumok használatainak egyik előnye az, hogy több olyan tulajdonságot tartalmaznak, amelyek közvetlenül az adatbázis tábláinak neveire képezhetők le. Így ahelyett, hogy bele kéne ásnunk magunkat a táblák gyűjteményébe a Tables tulajdonság segítségével, egyszerűen használhatjuk az Inventory tulajdonságot. Vizsgáljuk meg az alábbi kódresztet, amely az érthető-ség kedvéért megjegyzéseket is tartalmaz:

```
// Ez mind a tervező által létrehozott kód.
public partial class InventoryDataSet : global::System.Data.DataSet
{
    // InventoryDataTable típusú tagváltozó.
    private InventoryDataTable tableInventory;

    // Minden konstruktor meghív egy InitClass() segédmetódust.
    {
        public InventoryDataSet()
        {
            this.InitClass();
        }

        // Az InitClass() előkészíti a datasetet, és hozzáadja az
        // InventoryDataTable osztályt a Tables gyűjteményhez.
        private void InitClass()
        {
            this.DataSetName = "InventoryDataSet";
            this.Prefix = "";
            this.Namespace = "http://tempuri.org/InventoryDataSet.xsd";
            this.EnforceConstraints = true;
            this.SchemaSerializationMode =
                global::System.Data.SchemaSerializationMode.IncludeSchema;
            this.tableInventory = new InventoryDataTable();
            base.Tables.Add(this.tableInventory);
        }
    }
}
```

megfelelő).

Az egyedi datarow típus sokkal kevésbé egzotikus, mint a generált dataset vagy datatable. Ez az osztály kibővíti a datarow osztályt (lásd 23.25. ábra), és olyan tulajdonságokat nyújt, amelyek közvelelő az Inventory tábla sémájára

Megjegyzés Az erősen típusos datatable számos olyan eseményt is definiál, amelyekkel megfigyelhetjük a táblázat adatainak módosításait.

lása mellett).

használatával (vonzó alternatíva a row és column indexelők manuális navigálásához). Sokkal érdekesebb, hogy ez az erősen típusos datatable osztály egy olyan metóduskészletet definiál (lásd a 23.24. ábrát), amely sorok beszűrését, hely-

rekorrdok aktuális számát. valamint egy egyedi indexelőt és egy count tulajdonságot, hogy beolvassa a táblát az alapul (CartIDColumn, ColorColumn, MakeColumn és PetNameColumn), olyan tulajdonságkészletet definiál, amely a fizikai Inventory tábla oszlopneve- pusd, mint a most megvizsgált erősen típusos dataset tagváltozója) egy- egy erősen típusos datarow osztályt, amelyek mindegyikét beágyazta az Inven- torydataset osztályba. Az Inventorydataset osztály (amely ugyanolyan ti- pusú, mint a most megvizsgált erősen típusos dataset tagváltozója) egy- A fentiekhez hasonlóan, a varázsló létrehozott egy erősen típusos datatable és

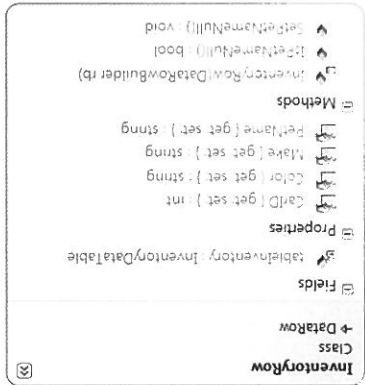
A generált DataTable és DataRow típusok vizsgálata

Amellett, hogy egyesíti a datatable objektum fenntartásának részleteit, a ter- vező által generált erősen típusos dataset tartalmazhat hasonló logikát, hogy feltájon olyan datarelation objektumokat (ezekkel jelenleg nem rendelke- zünk), amelyek a táblák közötti kapcsolatot képviselik.

```
// Az frásvédeett Inventory tulajdonság visszaadja
public InventoryDataTable tagváltozót.
{
    get { return this.tableInventory; }
}
```



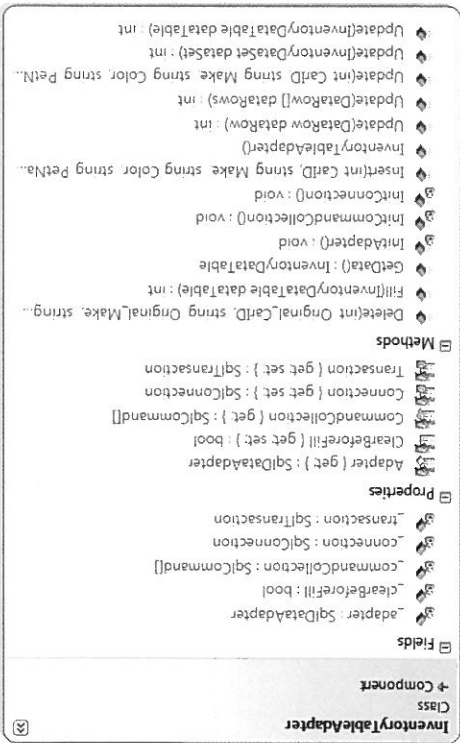
23.24. ábra: Az egyedi DataTable típus



23.25. ábra: Az egyedi DataRow típus

A generált adatillesztő

Az erős típusok megléte a kapcsolat nélküli típusok számára a Data Source Configuration Wizard használatának komoly előnye, ugyanis az erős típusos osztályok manuális hozzáadása meglehetősen hosszadalmas (bár kivitelezhető) lenne. A varázsló szerencsére egyedi adatillesztő objektumot generál, amely képes feltölteni és módosítani az InventoryDataSet és az InventoryDataTable osztálytípusokat (lásd a 23.26. ábrát).



23.26. ábra: Egyedi adatillesztő, amely erősen típusos típusokkal működik

Az automatikusan generált InventoryTableAdapter típus karbantartja az SqlCommand objektumok gyűjteményét, amelyek mindegyike rendelkezik az SqlParameter objektumok teljes felületével (ez jelentős időmegtakarítást eredményez). Emellett ez az egyedi adatillesztő tulajdonságkészletet biztosít az alapul szolgáló kapcsolat-, tranzakció- és adatillesztő objektumok kinyerésére, valamint egy tulajdonságot az egyes parancstípusokat képviselő tömb megszerzésére. Ennek nyílvaáváló előnye az, hogy nem nekünk kell megírni a kódot.

Generált típusok használata a kódban

Ha megvizsgáljuk az `urlapból` származtatott típus `Load` eseménykezelőjét, látható, hogy az egyedi adattílllesztő `Fill()` metódusa indításakor meghívódik, és átadja az egyedi dataset által fenntartott egyedi datatábla típusát:

```
private void MainForm_Load(object sender, EventArgs e)
{
    this.InventoryTablAdapter.Fill(this.InventoryDataSet.Inventory);
}
```

Ugyanezt az egyedi adattílllesztő objektumot használhatjuk arra, hogy frissítsük a táblázat módosításait. Adjunk az `urlap` felhasználói felületéhez egy `Button` vezérlőelemet (`btnUpdateInventory`), készítsünk eseménykezelőt a `Click` eseményhez, és írjuk bele a következő kódot:

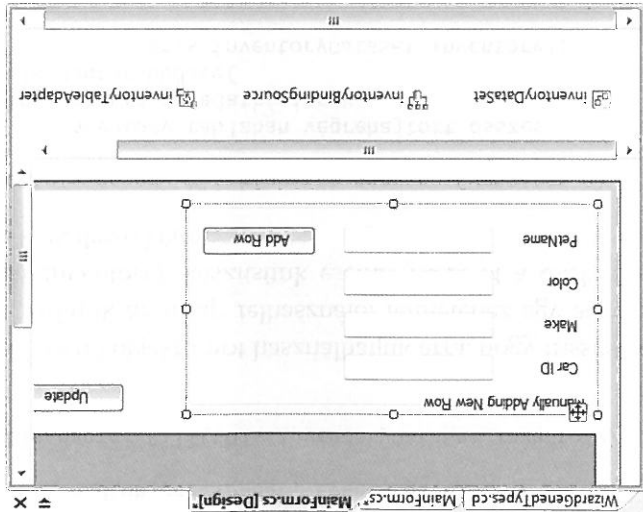
```
private void btnUpdateInventory_Click(object sender, EventArgs e)
{
    // Ez visszaadja az Inventory táblában végrehajtott összes
    // módosítást felidőgzőzsra az adatbázisnak.
    this.InventoryTablAdapter.Update(
        this.InventoryDataSet.Inventory);
    // Megkapjuk a táblázatot friss másolatát.
    this.InventoryTablAdapter.Fill(this.InventoryDataSet.Inventory);
}
```

Futassuk megint az alkalmazást; adjunk hozzá, töröljünk vagy módosítsunk rekordot a táblázatban; végül kattintsunk az `Update` gombra. Ha megint futatjuk a programot, láthatjuk, hogy a változások megjelentek. Az összes erősen osztályt közvetlenül felhasználhatjuk a kódukbán (nagyjából) ugyanúgy, ahogy azt már megtettük a korábbiakban. Tette-lezzük fel például, hogy olyan felhasználói felület-tartományal frissítettük az `urlapunkat` (lásd a 23.27. ábrát), amely lehetővé teszi a felhasználó számára, hogy új rekord beírását szövegmezők segítségével (ez voltaképpen feleslegesen ebben a példában, ugyanis ezt a `dataGridView` megteszi helyettünk). Az új `Button` vezérlőelem `Click` eseménykezelőjébe beírhatjuk az alábbi kódot:

```
private void btnAddRow_Click(object sender, EventArgs e)
{
    // Lekérdezzük az adatokat a szövegdobozokból
    int id = int.Parse(txtCarID.Text);
    string make = txtMake.Text;
    string color = txtColor.Text;
    string petName = txtPetName.Text;
}
```

```
// Egyedi ílllesztő használataval beszűnünk egy sort.
inventoryTableAdapter.Insert(id, make, color, petname);

// Újra feltöltjük a táblázatot.
this.inventoryTableAdapter.Fill(this.inventoryDataSet.inventory);
}
```



23.27. ábra: Egyszerű frissítés az újrap hűpusához

Manuálisan is hozzáadhatunk egy új sort:

```
private void btnAddRow_Click(object sender, EventArgs e)
{
    // Új sor beolvasása.
    InventoryDataSet.InventoryRow newRow =
        inventoryDataSet.Inventory.NewInventoryRow();
    newRow.CarID = int.Parse(txtCarID.Text);
    newRow.Make = txtMake.Text;
    newRow.Color = txtColor.Text;
    newRow.PetName = txtPetName.Text;
    inventoryDataSet.Inventory.AddInventoryRow(newRow);

    // Egyedi ílllesztő használataval beszűnünk egy sort.
    inventoryTableAdapter.Update(inventoryDataSet.inventory);

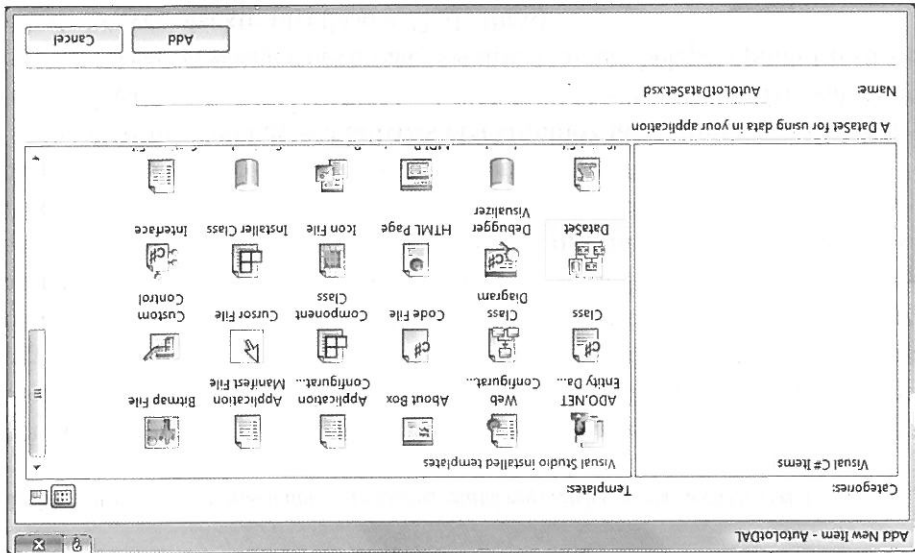
    // Újra feltöltjük a táblázatot.
    this.inventoryTableAdapter.Fill(this.inventoryDataSet.inventory);
}
```

Forráskód A VisualDataGridViewApp kódfájlokat a forráskódkönyvtár 23. fejezetének al-könyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Az automatikusan generált kód elválasztása a felhasználófelület-rétegről

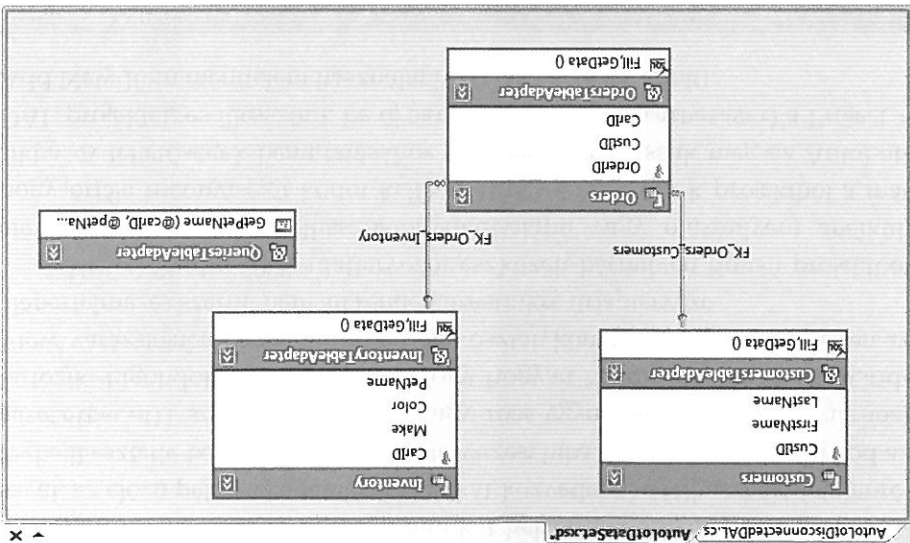
Végül mutassunk rá, hogy míg a DataGridView által elindított Data Source Configuration Wizard elvégezte a munkát helyettünk rengeteg kód létrehozásával, az előző példa az adatelérési logikát közvetlenül a felhasználófelület-rétegbe illesztette be – ez pedig súlyos tervezési hiba. Ideális esetben ez a kód az AutoLotDAL.d11 szerezésvényünkhöz (vagy más egyéb adatelérési könyvtárhoz) tartozzik. Elgondolkozhatunk azon, hogy hogyan használhatjuk a DataGrid-View varázslójában generált kódot egy osztálykönyvtár-projektben, hiszen az alapértelmezés szerint nem áll rendelkezésünkre uralptervező.

A Visual Studio 2008 adattervezői eszközeit bármilyen típusú projektből elérhetjük (legyen az felhasználófelület-alapú vagy másmilyen) anélkül, hogy óriási mennyiségű kódot kellene átmásolnunk egyik projektből a másikba. A lehetőségek bemutatásához még egyszer nyissuk meg az AutoLotDAL projektet, és illesszünk be új dataset típust (AutoloDataset) a Project > Add New Item menüelem használatával (lásd a 23.28. ábrát).



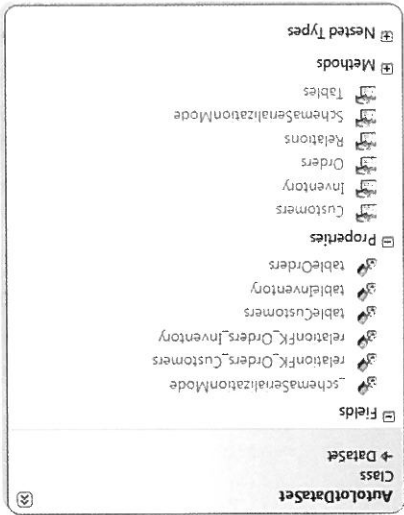
23.28. ábra: Új DataSet beszúrása

Ekkor megnyílik egy üres Dataset Designer felület. A Server Explorer segítségével kapcsolódjunk egy adott adatbázishoz (elvileg már kapcsolódunk az AutoLothoz), és húzzunk át minden adatbázis-objektumot (most ne törődjünk a CreditRisk táblával), amelyet a felületen szereznénk generálni. A 23.29. ábrán láthatjuk az AutoLot minden olyan egyedi aspektusát, amelyről most gondoskodunk.

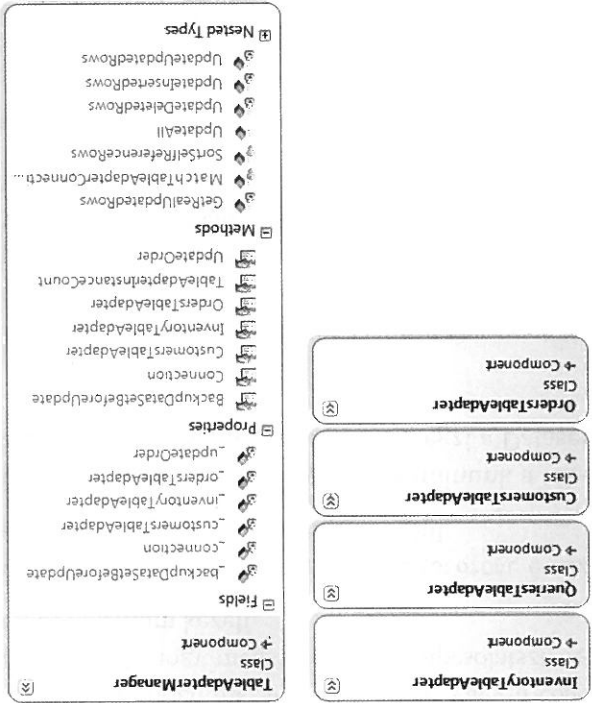


23.29. ábra: Az egyedi erősen típusos típusaink, ezúttal egy osztálykönyvtár-projektben belül

Ha rápillantunk a létrehozott kódra, erősen típusos dataset, datacube és datarow típusok egy új köteget láthatjuk, valamint minden egyes táblához egy egyedi adattílesztő objektumot. Mivel az autolotdataset típus tartalmaz olyan kódot, amely feltölti és frissíti az Autolot adatbázis minden tábláját, az automatikusan generált kód mennyisége több mint 3000 sor. Ennek nagy része azonban csak infrastruktúra, amelyet figyelmen kívül hagyhatunk. Ahogy a 23.30. ábrán látható, az autolotdataset típus az előző inventorydataset típus létrehozásához nagyon hasonlóan készült. Emellett minden egyes adatbázis-objektumhoz találhatóunk egyedi adattílesztő objektumot, amelyet áthúztunk a Dataset Designer felületére, valamint egy TableAdapterManager nevű hasznos típust, amely belépési pontot biztosít minden egyes objektumba (lásd a 23.31. ábrát).



23.30. ábra: Az AutoloDataSet



23.31. ábra: Az automatikusan generált adatillesztő objektumok

Forráskód Az AutoloDAL (Part 3) kódfrájlokat a forráskódkönyvtár 23. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Egy felhasználói felület front end: újra a MultitabledDataSetApp

Az automatikusan generált típusok használata meglehetősen egyszerű, ha rutinosan dolgozunk a kapcsolat nélküli modellel. A szöveghez letölthető forráskód tartalmaz egy MultitabledDataSetApp-Redux projektet, amely – ahogy a neve is mutatja – a fejezet során korábban létrehozott MultitabledDataSetApp projekt egyik verziója.

Az eredeti példa gyengén típusos dataset és sqldataadapter típusokat használ, hogy adatokat mozgasson a táblázatok között. Ez a verzió az autotdaltal. d11 harmadik iterációját és a varázsló által generált típusokat használja. Bár most nem foglalkozunk azzal, hogy megnezzük a teljes kódot (mivel többé-kevésbé ugyanaz, mint a projekt első iterációja), néhány fontos megjegyzést azonban érdemes tenni:

- Többé nem kell manuálisan létrehozniunk App.config fájlt vagy a configurationManagert használniunk, hogy megszerezzük a kapcsolatsztrínget, ugyanis ezt a settings objektum kezeli.
- Most már erősen típusos osztályokat használunk az autotdaltal és az autolotdal.AutolotDataSetTableadapters névtérben belül.
- Többé nem kell manuálisan létrehozniunk vagy konfigurálniunk a tábláink közötti kapcsolatot, ezt ugyanis automatikusan elvégzi a DataSet Designer.

Az utolsó ponttal kapcsolatban figyelni kell arra, hogy a DataSet Designer által a táblakapcsolatoknak adott nevek *különböznek* azoktól, amelyeket mi adtunk a projekt első iterációjában. Ezért a btnGetOrderInfo_Click() metódust módosítani kell, hogy a helyes kapcsolatneveket (amelyeket a DataSet Designer tervezői felületén láthatunk) használja, például:

```
private void btnGetOrderInfo_Click(object sender,
    System.EventArgs e)
{
    ...
    // Frissíteni kell a kapcsolatot nevet.
    drsOrder =
        drsCust[0].GetChildRows(
            autolotDS.Relations["FK_Orders_Customers"]);
    ...
}
```

```
// Frissíteni kell a kapcsolatot nevet.  
DataRow[] drsinv =  
    drsorder[0].getParendRows(  
        autolots.Relations["FK_Orders_Inventory"]);  
...  
}
```

Forráskód A MultitabledDataSetApp-Redux kódfrágmenteket a forráskódkönyvtár 23. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Összefoglalás

Ebben a fejezetben az ADO.NET kapcsolat nélküli modelljének részleteivel foglalkoztunk, amelynek központi eleme a dataset. Ez a típus tetszőleges számu tábla, valamint tetszőleges számu opcionális kapcsolat, korlátozás és kifejezés memóriabeli megjelenítésére szolgál. A helyi táblák közötti kapcsolatok kiépítésének előnye az, hogy programozottan navigálhatunk közöttük, miközben azok nem kapcsolódnak a távoli adattárolóhoz.

A fejezetben megvizsgáltuk az adatillesztő típus szerepét. Ennek a típusnak (és a kapcsolódó selectcommand, insertcommand, updatecommand és deletecommand tulajdonságoknak) a használataival az illesztő végrehajthatja a dataset módosításait az eredeti adattárban is. Emellett megtanultuk, hogyan navigáljuk egy dataset objektummodelljében manuálisan, valamint erősen típusos objektumok révén, amelyeket főként a Visual Studio 2008 Dataset Designer eszközeivel generáltunk.

A LINQ API programozása

Ebben a fejezetben elkezdjük a *LINQ to ADO.NET* vizsgálatát. Ezt a kifejezést LINQ programozási modellhez kapcsolódó két jellemző leírására használjuk, a LINQ to DataSet és a LINQ to SQL lehetőségekhez. Ezek az API-k teszik lehetővé a LINQ-lekérdezések végrehajtását relációs adatbázisokban, illetve ADO.NET dataset objektumokon. Ezután a LINQ to XML szerepét tekintjük át. A LINQ ezen aspektusa nemcsak azt teszi lehetővé, hogy a lekérdezésoptimátorok készsletének használatával adatokat nyerjünk ki XML-dokumentumból, hanem azt is, hogy lényegre törően (sokkal hatékonyabban, mint ha `system.xml.dll` szerelvénybe csomagolt típusokat használunk) töltsünk be, mentstünk és hozzunk létre XML-dokumentumokat.

Megjegyzés Ez a fejezet az első kötet14. fejezetében leírt LINQ programozási modellben való jártasságot feltételezi.

A LINQ to ADO.NET szerepe

A LINQ programozási modell (lásd az előző kötet 14. fejezetét is) lehetővé teszi a programozók számára, hogy olyan erősen típusos lekérdezéskifejezéseket hozzanak létre, amelyeket az adattárolók széles körére lehet alkalmazni (tömbök, gyűjtemények, adatbázisok, XML-dokumentumok). Függelentől attól, hogy milyen objektumon hajtjuk végre a LINQ-lekérdezést, mindig lehet ugyanazokat a lekérdezésoptimatorokat használni, és a LINQ to ADO.NET API néhány további típust és infrastruktúrát is biztosít a LINQ-adatbázis integrációjához.

A LINQ to ADO.NET olyan gyűjtőfogalom, amely a LINQ két adatbázis-központú szempontját írja le. Az egyik a LINQ to DataSet. Ez az API lényegében bővítménykészlet a szabványos ADO.NET DataSet programozási modellhez, amely lehetővé teszi, hogy a dataset, a datatable és a datarow típusokon is lehessen LINQ-lekérdezéskifejezéseket végrehajtani. A `system.core.dll` típusainak használatán túl a LINQ to DataSet megköveteli, hogy a projekt használja a `system.data.dataextensions.dll` szerelvény kiegészítőitípusait.

A LINQ to ADO.NET másik komponense a LINQ to SQL. Ez az API lehetővé teszi, hogy relációs adatbázisokkal működjünk együtt, mert elvonatkoztat az alapul szolgáló ADO.NET-adattípusoktól (kapcsolatok, parancsok, adatillesztők stb.) az *entitásmosztályok* használatára. Ezen az entitásmosztályokon keresztül ábrázolhatunk relációs adatokat egy intuitív objektummodellel, valamint manipulálhatjuk az adatokat a LINQ-lekérdezések segítségével. A LINQ to SQL funkcionális a `System.Data.Linq.dll` szerelvényben található.

Megjegyzés A .NET 3.5 óta a LINQ to SQL nem támogatja az `data provider factory` modellt (lásd a 22. fejezetet). Ezért amikor ezt az API-t használjuk, az adatoknak a Microsoft SQL Serveren kell lenniük. Ezzel szemben a LINQ to DataSet API természetesen „szabadosabb”, ugyanis a kezelt DataSet bármilyen relációs adatbázisból származhat.

Programozás a LINQ to DataSettel

Az előző fejezetben láthattuk, hogy a DataSet típus a kapcsolat nélküli modell központi eleme, és arra használható, hogy jelképezze a kapcsolódó DataTable objektumok gyorsítottátárazott másolatát, valamint (esetlegesen) a közöttük lévő kapcsolatokat. Ehhez kapcsolódó megjegyzés, hogy egy DataSet adatait három különböző módon lehet manipálni:

- indexelőkkel,
- adattáblázat-olvasókkal,
- erősen típusos adattagokkal.

Amikor a DataSet és DataTable típusok különböző indexelőit használjuk, akkor meg lehetőségen egyértelműen, de nagyon gyakran tipizáltan dolgozhatunk a tárolt adatokkal. Ez a megközelítés azt követeli meg, hogy az adatokat cellák táblázatos blokkjaként kezeljük. Példaul:

```
static void PrintDataWithIndexes(DataTable dt)
{
    // Kijelöl a DataTable-t.
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
        for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
        {
            Console.WriteLine(dt.Rows[curRow][curCol].ToString() + "\\t");
        }
    }
}
```

```

        console.WriteLine();
    }
}

```

A `DataTable` típus `CreateDataReader()` metódusa egy olyan másik megközelítést kínál, amely szerint a `DataSet` adatait úgy kezelhetjük, mint sorok lineáris készleteinek szekvenciális feldolgozását:

```

static void PrintDataWithDataReader(DataTable dt)
{
    // Beolvassuk a DataTableReader típust.
    DataTableReader dtreader = dt.CreateDataReader();
    while (dtreader.Read())
    {
        for (int i = 0; i < dtreader.FieldCount; i++)
        {
            console.WriteLine("{0}\t", dtreader.GetValue(i));
        }
        console.WriteLine();
    }
    dtreader.Close();
}

```

Végül az erősen típusos `DataSet` lehetővé teszi, hogy az objektumban lévő adatokat olyan tulajdonságok használatával kezeljük, amelyek leképezhetők a relációs adatbázis tényleges oszlopneveire.

Erősen típusos objektumok használatával írhatunk olyan kódokat, mint például a következő (lásd a 23. fejezet):

```

static void AddRowWithTypedDataSet()
{
    InventoryTableAdapter invDA = new InventoryTableAdapter();
    AutoLoadDataSet dataSet = invDA.GetData();
    inv.AddInventoryRow(999, "Ford", "Yellow", "Sail");
    invDA.Update(inv);
}

```

Ezen túlmenően a LINQ to DataSet még egy lehetőséget biztosít a tárolt adatok kezelésére a LINQ-lekérdezőkifejezések használatával. Az `ADO.NET` `DataSet`-be (és az olyan kapcsolódó típusokba, mint a `DataTable` és `DataRowView`) alapvetően nincsen integrálva a szükséges infrastruktúra ahhoz, hogy közvetlenül végrehajtsunk rajta egy LINQ-lekérdezést. A következő metódus például fordítási idejű hibát eredményezne:

```

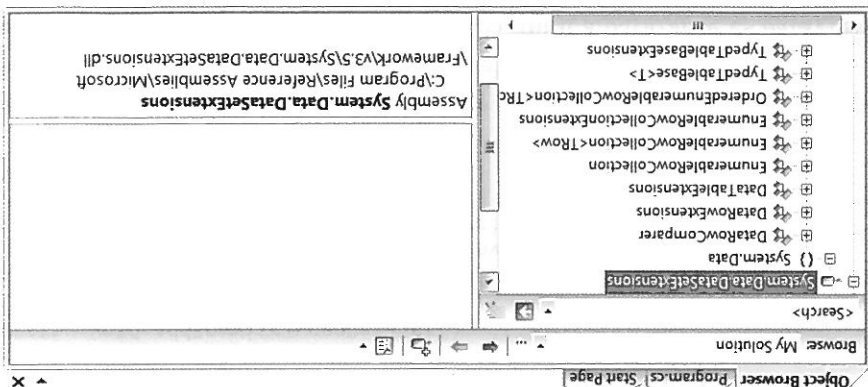
static void LinqOverDataTable()
{
    // Adatok DataTable típusának beolvasása.
    InventoryDALInventoryLayer dal = new InventoryDALInventoryLayer(
        @"Data Source=(local)\SQLEXPRESS;" +
        "Initial Catalog=Autolo;Integrated Security=True");
    DataTable data = dal.GetAllInventory();
    // Az 5-nél nagyobb carid-vel rendelkező típusok beolvasása?
    var moreData = from c in data where (int)c["CarID"] > 5 select c;
}

```

Ha lefordítanánk a `LinqOverDataTable()` metódust, akkor a fordító arról tájékoztatna, hogy a `DataTable` típus biztosít egy „lekérdezésiminta-implementációt”. Hasonlóan az `IEnumerable<T>` típust (pl. az `ArrayList`-et) nem implementáló objektumokra vonatkozó LINQ-lekérdezések alkalmazásához, az ADO.NET-objektumokat pedig kompatibilis típusúak kell alakítani. Hogy megértjük ennek mikéntjét, meg kell vizsgálnunk a `System.Data.DataTable`-típusait.

A DataSet bővítmények szerepe

A `System.Data.DataSetExtensions.dll` több új taggal bővíti a `System.Data` névteret (lásd a 24.1. ábrát).



24.1. ábra: A `System.Data.DataSetExtensions.dll` szerelvény

A két leghasznosabb tag vitathatatlanul a `DataTableExtensions` és a `DataRowExtensions`. Ahogy a nevük is mutatja, ezek a típusok a `DataTable` és a `DataRow` típusok működését bővíti metódusok készletével egészítik ki. Egy másik kulcsfontosságú típus a `TypedTableBaseExtensions`, amely olyan bővíti metó-

dusokat definiál, amelyeket erősen típusos dataset objektumokra lehet alkalmazni. A `system.data.dataextensions.dll` szerelvény összes többi tagja pusz-
tán az infrastruktúrát szolgálja ki, és nem közvetlen használatra szánják őket.

DataTable LINQ-kompatibilis használata

A dataset bővítmények használatának bemutatásához tételezzük fel, hogy van egy új C# parancssori konzolalkalmazásunk `linqoverdataset` névvel. Fontos megjegyezni, hogy amikor a .NET 3.5-öt célzó projekteket készítünk, akkor automatikusan referenciát kapunk a `system.core.dll` és `system.data.dataextensions.dll` szerelvényekre; ám ebben a példában adjunk hozzá még egy szerelvényreferenciát az `autolot.dll` szerelvényre (lásd a 23. fe-
jezetet), és egészítsük ki a kezdeti kóddal a következő logikával:

```
using System.Data;
using AutolotDisconnectedLayer;

namespace LinqOverDataSet
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** LINQ over DataSet *****\n");
            // Beolvassuk az Autolot adatbázis aktuális Inventory
            // táblájának DataTable típusát.
            InventoryDALDisLayer dal = new InventoryDALDisLayer(
                @"Data Source=(local)\SQLExpress;Initial
                Catalog=Autolot;" + "Integrated Security=True");
            DataTable data = dal.GetAllInventory();

            // Hívjuk meg az itt következő metódusokat.
            Console.ReadLine();
        }
    }
}
```

Amikor LINQ-kompatibilis objektumná szeretnénk átalakítani egy `ADO.NET DataTable` típust, akkor egyszerűen hívjuk meg az `AsEnumerable()` bőví-
tődést, amelyet a `dataextensions` típus definiál. Ez visszaad egy `Enume-
rableCollection` objektumot, amely egy `DataRow` gyűjteményt tartalmaz. Az
`EnumerableCollection` típus használatával az elvártak szerint dolgozhatunk
az egyes sorokkal. Nézzünk egy egyszerű példát:

```

static void PrintAllCarIDs(DataTable data)
{
    // Beolvassuk a DataTable felismerhető változatát.
    EnumerableViewCollection enumdata = data.AsEnumerable();

    // Kijelöljük az azonosító értékeit.
    foreach (DataRow r in enumdata)
        Console.WriteLine("Car ID = {0}", r["CarID"]);
}

```

Mivel az `EnumerableViewCollection` megvalósítja az `IEnumerable<T>` típust, ezért az is megengedett, hogy az alábbi utasítások egyikével kapjuk meg a visszatérési értéket:

```

// A visszatérési érték tárolása IEnumerableView<T> típusként.
IEnumerable<DataRow> enumdata = data.AsEnumerable();

// A visszatérési értékek tárolása implicit módon.
var enumdata = data.AsEnumerable();

```

Eddig ugyan még nem futtattuk le a LINQ-lekérdezést, ám az `enumdata` objektumon immár végrehajthatunk LINQ-lekérdezéskifejezést. Az `EnumerableViewCollection` valóban `DataRow` objektumok gyűjteményét tartalmazza, ezért egy típusindexelőt alkalmazunk az összes alobjektumra, hogy kijeressük a `CarID` oszlop értékét.

A legtöbb esetben nem kell `EnumerableViewCollection` típusú változót deklarálnunk azért, hogy tárolja az `AsEnumerable()` visszatérési értéket. Ehelyett hívjuk meg ezt a metódust magából a lekérdezéskifejezésből. Akad egy még érdekesebb módus is, amely visszaadja a `CarID`/makes oszlopok leképezését a `DataTable` összes bejegyzéséből, ahol a `CarID` értéke nagyobb mint 5:

```

static void ApplyLINQQuery(DataTable data)
{
    // Képezzünk le egy új eredményhalmazt, amely tartalmazza
    // az ID/color értékeket az 5-nél nagyobb CarID-vel rendelkező
    // sorok esetében.
    var cars = from car in data.AsEnumerable()
    where
        (int)car["CarID"] > 5
    select new
    {
        ID = (int)car["CarID"],
        color = (string)car["Color"]
    };
}

```

```

Console.WriteLine("Cars with ID greater than 5:");
foreach (var item in cars)
{
    Console.WriteLine("<-> CarID = {0} is {1}", item.ID, item.Color);
}
}

```

A DataRowExtensions.Field<T>() bővítő metódus szerepe

A jelenlegi LINQ-lekérdezéskifejezésben hátrányos lehet az, hogy számos kasztolási műveletet és datarow indexelést használunk az eredményhalmaz összeállításához, és ez futásidőjelel eredményezhet, ha nem kompatibilis adat típusra próbálunk kasztolni. Ahhoz, hogy erős tipizálást alkalmazzunk a lekérdezésünkben, használhatjuk a datarow típus Field<T>() bővítő metódusát. Így megnöveljük a lekérdezés típusbiztonságát, ugyanis az adat-típusok kompatibilitását a rendszer a fordítási időben ellenőrzi. Nézzük meg a következőképpen módosított kódot:

```

var cars = from car in data.AsEnumerable()
            where
                car.Field<int>("CarID") > 5
            select new
            {
                ID = car.Field<int>("CarID"),
                Color = car.Field<string>("Color")
            };

```

Látható, hogy ebben az esetben meghívhatjuk a Field<T>() metódust, és megadhatunk egy típusparamétert, amely az oszlop alapítusát határozza meg. A metódus argumentumaként átadjuk magát az oszlopnevet. A fordítási időben ellenőrzés miatt tanácsos a Field<T>() metódust használni a datarow indexelője helyett Enumerable.Cast<T>() metódust, a LINQ-lekérdezés Ázom túl, hogy meghívjuk az AsEnumerable() metódust, a LINQ-lekérdezés általános formátuma hasonló ahhoz, amelyet már láttunk az előző kötet 14. fejezetében. Ezért most nem térünk ki újra a különböző LINQ-operátorok részleteire. Ha további példákra vagyunk kíváncsiak, nézzük át a „LINQ to DataSet Examples” témakört a .NET Framework 3.5 SDK dokumentációjában.

Új DataTable objektumok feltöltése a LINQ-lekérdezésekből

Könnnyedén fel lehet tölteni egy új DataTable adatait egy LINQ-lekérdezés alapján, feltéve, hogy *nem* használunk lekérdezéseket. Amikor van egy olyan eredményhalmazunk, amelyben az alapul szolgáló típus `IEnumerable<T>` típusként lehet ábrázolni, meghívhatjuk a `CopyToDataTable<T>` metódust az eredményen. Például:

```
static void BuildDataTableFromQuery(DataTable data)
{
    var cars = from car in data.AsEnumerable()
               where
                   car.Field<int>("CarID") > 5
    select car;

    // Használjuk ezt az eredményhalmazt új DataTable létrehozásához.
    DataTable newtable = cars.CopyToDataTable();

    // Készítsük a DataTable-t.
    for (int currow = 0; currow < newtable.Rows.Count; currow++)
    {
        for (int curcol = 0; curcol < newtable.Columns.Count; curcol++)
        {
            Console.WriteLine(
                newtable.Rows[currow][curcol].ToString().Trim() + "\t");
        }
        Console.WriteLine();
    }
}
```

Megjegyzés Az `AsDataTable<T>` (`<T>`) bővítő metódussal is átalakíthatunk egy LINQ-lekérdezést `DataTable` típusúvá.

Ez a megközelítés akkor lehet nagyon hasznos, ha a LINQ-lekérdezés eredményét adatkötes során forrásként szeretnénk használni. A Windows Forms `DataGridView` (valamint az ASP.NET `GridView`) vezérlőelem például támogatja a `DataSource` tulajdonságot. LINQ-eredményt az alábbiak szerint köthetünk egy `DataGridView`-hez:

```
// Tegyük fel, hogy a mydatagrid GUI-alapú táblázatobjektum.
mydatagrid.DataSource = (from car in data.AsEnumerable()
                          where
                              car.Field<int>("CarID") > 5
                          select car).CopyToDataTable();
```

A következőkben térünk át a LINQ to SQL-re.

Programozás a LINQ to SQL használatával

A LINQ to SQL olyan API, amely lehetővé teszi jól megformázott LINQ-lekérdezések kifejezéseinek alkalmazását relációs adatbázisokban lévő adatokra. A LINQ to SQL több olyan típust kínál (a `system.data.linq.dll` szerelvényben), amelyek megkönnyítik a kommunikációt a kód és a fizikaiadatbázis-motor között.

A LINQ to SQL legfőbb célja az, hogy konzisztenciát biztosítson a relációs adatbázisok és a velük folytatott munkára használt programozási logika számára. Ahelyett például, hogy az adatbázis-lekérdezéseket nagy sztringekkel jelképezzünk, használnánk erősen típusos LINQ-lekérdezéseket. Továbbá ahelyett, hogy a relációs adatokat rekordok folyamanként kellene kezelnünk, dolgozhatunk az adatokkal szabványos objektumorientált programozási módszerekkel. Mivel a LINQ to SQL lehetővé teszi az adatelérés integrálását közvetlenül a C#-forráskódba, nagymértékben csökkenni fog a szükségesség, hogy manuálisan hozzuk létre egyedi osztályok tucait, valamint azokat az adatelérési könyvtárakat, amelyek elrejtik szem elől az ADO.NET csúnya kódjait.

Amikor LINQ to SQL felhasználásával programozunk, nyoma sincs az olyan megszokott ADO.NET-típusoknak, mint az `SqlConnection`, az `SqlCommand` vagy az `SqlDataAdapter`. A LINQ-lekérdezések kifejezések, a (harmadosan ismeretendő) entitásosztályok és a `DataContext` típus segítségével végrehajthatjuk a kívánt adatbázis-műveleteket (leérkezés, eltávolítás, módosítás és törlés), valamint definiálhatunk tranzakciós kontextusokat, létrehozhatunk új adatbázis-entitásokat (vagy egész adatbázisokat), meghívhatunk tárolt eljárásokat, és elvégezhetünk egyéb adatbázis-centrikus tevékenységeket.

Ezenfelül a LINQ to SQL típusok (példaként megint a `DataContext`-et említhetjük) feladata, hogy megvalósítsák a standard ADO.NET-adattípusok integrációját. A `DataContext` egyik túlterhelt konstruktora bemenneként például egy `IdbConnection` objektumot vár, amely az összes ADO.NET-kapcsolatobjektum által támogatott közös interfész. Ilyen módon a már meglévő ADO.NET adatelérési könyvtárak integrálhatók a C# 2008 LINQ-lekérdezésekbe (és fordítva). Valójában – ami a Microsofttól illet – a LINQ to SQL egyszerűen az ADO.NET-család egyik új tagja.

Az entitásosztályok szerepe

Ha a LINQ to SQL-t szeretnénk használni az alkalmazásainkban, akkor az első lépés az *entitásosztályok* definiálása. Röviden: az entitásosztályok olyan típusok, amelyek azokat a relációs adatokat jelképezik, amelyekkel dolgozni szeretnénk. Programozási szempontból az entitásosztályok olyan osztálydefiniciók, amelyeket különböző LINQ to SQL attribútumokkal (például [Table] és [Column]) jelölünk, amelyek leképezzhetők fizikai táblákra egy adott adatbázisban. A LINQ to SQL attribútumok többségének definícióját a `System.Data.Linq.Mapping` névtér foglalja magában (lásd a 24.2. ábrát).

Ahogy azt hamarosan látni fogjuk, a .NET Framework 3.5 SDK (valamint a Visual Studio 2008) olyan eszközöket biztosít, amelyek automatizálják az alkalmazás számára szükséges entitás típusok elkészítését. Ezt megelőzően az első LINQ to SQL példánk fogja bemutatni, hogy hogyan hozzunk létre entitásosztályokat.



24.2. ábra: A `System.Data.Linq.Mapping` névtér számos LINQ to SQL attribútumot definiál

A DataContext típus szerepe

Ha meghatároztuk az entitásosztályokat, akkor a `DataContext` típusal átdahatjuk a lekérdezési kifejezéseket a relációs adatbázisnak. Ez a LINQ to SQL-specifikus osztálytípus felelős a LINQ-lekérdezések megfelelő SQL-lekérde-

zésekre történő fordításáért, valamint az adott adatbázissal történő kommunikációért. Bizonyos tekintetben a DataContext hasonló egy ADO.NET-kapcsolatobjektumra: szüksége van egy kapcsolatstríngre. A típusú kapcsolatobjektummal ellentétben azonban a DataContext típus számos olyan taggal rendelkezik, amelyek a lekérdezéskifejezések eredményeit az általunk meghatározott entitásosztályokba képezik le.

Továbbá a DataContext típus egy factoryminitét definiál a kódban használt entitásosztályok példányainak megszerzésére. Ha megszeroznénk egy entitáspéldányt, akkor a kívánt módon változtathatjuk meg annak állapotát (rektorok hozzáadása és frissítése stb.), és a módosított objektumot feldolgozásra visszaküldhetjük. Így módon a DataContext hasonló az ADO.NET adattílesztő típushoz.

Egy egyszerű LINQ to SQL példa

Először nézzük meg a LINQ to SQL használatának egy egyszerű példáját, amely a 22. fejezetben létrehozott Autolot adatbázis Inventory táblájával dolgozik. Ebben a példában nem használjuk az AutolotDAL.dll könyvtárat, hanem minden kódot manuálisan készítünk el. Hozzunk létre egy új páranccsori konzolalkalmazást SimpleLINQtoSQLApp névvel, valamint adjunk egy referenciát a System.Data.Linq.dll szerezőlvényre.

Ezután adjunk hozzá egy új C#-osztályfájl (Inventory.cs). Ez a fájl fogja definiálni az entitásosztályunkat, amely megköveteli a típus kiegészítését különböző LINQ-központú attribútumokkal; ezért mindenképpen adjuk meg, hogy a System.Data.Linq.Mapping és System.Data.Linq névtérket használjuk. A következőkben nézzük meg az Inventory típus definícióját:

```
[Table]
public class Inventory
{
    [Column]
    public string Make;
    [Column]
    public string Color;
    public string PetName;
    // Az elsődleges kulcs azonosítása.
    [Column(IsPrimaryKey = true)]
    public int CarID;
```

```

    public override string ToString()
    {
        return string.Format("ID = {0}; Make = {1}; Color = {2};
                               PetName = {3}", carID, Make, Trim(),
                               Color, Trim(), PetName, Trim());
    }
}

```

Az entitásosztályunk megkapta a [Table] attribútumot, amíg a nyilvános mezők mindegyikét megjelöltük a [Column] attribútummal. A nevek mindkét esetben közvetlen leképezései a fizikai adatbázistáblának. Ez azonban nem szigorú követelmény, hiszen a tableattribute és a columnattribute típusok támogatják a name tulajdonságot, amely lehetővé teszi az adattábla programozott reprezentációjának elkülönítését magától a fizikai táblától. A carID mezőt tovább minősítettük a columnattribute típus isprimarykey tulajdonságának beállításával, a megnevezett tulajdonságszintaxis segítségével.

Az egyszerűség kedvéért minden mezőt publikusan deklaráltunk. Ha erősebb egységbe zárásra van szükségünk, akkor definiálhatunk nyilvános tulajdonságokba (vagy akár automatikus tulajdonságokba) csomagolt privát mezőket. Ekkor nem a mezők, hanem a *tulajdonságok* lesznek a [column] attribútummal megjelölve.

Egy entitásosztály bármennyi olyan tagot tartalmazhat, amelyek nem képezhetők le a reprezentált adattáblára. Ami a LINQ-futásidőt illeti, a rendezer csak a LINQ to SQL attribútumokkal megjelölt elemeket használja az adatcsere során. Ez az inventory osztálydefinició például a ToString() egyedi megvalósítását biztosítja ahhoz, hogy lehetővé tegye az alkalmazás számára az állapot gyors megjelentését.

Ha már rendelkezünk entitásosztállyal, használhatjuk a DataContext típust, hogy elküldjük (és lefordítsuk) a LINQ-lekérdezéskifejezéseinket a megadott adatbázisnak. Tanulmányozzuk az alábbi main() metódust, amely megjelenti az AutoLot adatbázis Inventory táblájában lévő összes elemet:

```

class Program
{
    const string cnstr =
        @"Data Source=(local)\SQLEXPRESS;Initial Catalog=AutoLot;" +
        "Integrated Security=True";
    static void Main(string[] args)
    {
        Console.WriteLine("***** LINQ to SQL Sample App *****\n");
        // DataContext objektum létrehozása.
        DataContext db = new DataContext(cnstr);
    }
}

```



```
// Most hozzunk létre egy Table<> típust.
Table<Inventory> invTable = db.GetTable<Inventory>();

// Az adatok megjelenítése LINQ-lekérdezés segítségével.
Console.WriteLine("-> Contents of Inventory Table from Autolot
    database:\n");
foreach (var car in from c in invTable select c)
    Console.WriteLine(car.ToString());
Console.ReadLine();
}
```

Amikor létrehozunk egy DataContext típust, meg kell adnunk a megfelelő kapcsolatsztringet, amely itt egyszerű sztringkonstansként jelenik meg. Ezt nyugodtan eltarolhatjuk az alkalmazáskonfigurációs fájlban, és/vagy használhatjuk az SqlConnectionStringBuilder típust az objektumorientáltabb kezelés érdekében.

Ezután megcserezzük az Inventory entitásosztályunk egy példányát a DataContext típus generikus GetTable<T>() módszerének meghívásával, ahol az entitásosztályt adjuk meg típusparaméterként. Végezetül létrehozunk egy LINQ-lekérdezéskifejezést, és végrehajljuk azt az invTable objektumon. A végeredmény az Inventory tábla minden elemének megjelenítése lesz.

Erősen típusos DataContext létrehozása

Míg első példánk erősen típusos volt az adatbázis-lekérdezés szempontjából, némileg elkülönböztettük a DataContext típust és az általa fenntartott Inventory entitásosztályt. Ennek orvoslásához általában érdemes létrehozunk egy olyan osztályt, amely kibővíti a kezelt táblák mindegyike számára tagváltozókat definiáló DataContext típust. Illesztünk be egy új osztályt AutolotData-base névvel, adjuk meg, hogy a System.Core és a System.Data.Linq névtérket használjuk, majd implementáljuk a típust az alábbiak szerint:

```
class AutolotDatabase : DataContext
{
    public Table<Inventory> Inventory;
    public AutolotDatabase(string connectionString)
    {
        base(connectionString);
    }
}
```

Az új osztálytípussal most már egyszerűsíthetjük a `main()` módszer kódját:

```
static void Main(string[] args)
{
    Console.WriteLine("***** LINQ to SQL Sample App *****\n");

    // Hozzunk létre egy AutolotDatabase objektumot.
    AutolotDatabase db = new AutolotDatabase(connectionString);

    // Most már használhatjuk az AutolotDatabase Inventory mezőjét.
    Console.WriteLine("> contents of Inventory Table from Autolot
    database:\n");
    foreach (var car in db.Inventory select c)
    {
        Console.WriteLine(car.ToString());
    }
}
```

Az erősen típusos adatkörnyezet létrehozásának egyik meglepő jellemzője, hogy a `DataContext`-ból származó típus (jelen példában az `AutolotDatabase`) közvetlenül nem hozza létre a `Table<T>` tagválasztókat, és nincs nyoma a várt `getTable()` módszerhívásnak. A fordítás során azonban, amikor végigiterálunk a LINQ-eredményhalmazon, a `DataContext` a háttérben létrehozza a `Table<T>` típust.

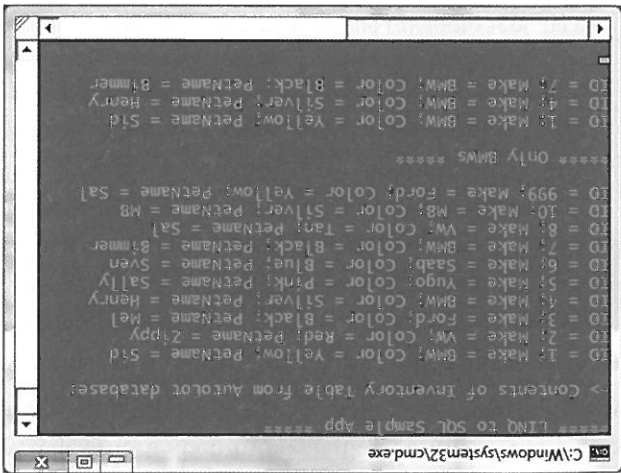
A LINQ-lekérdezések természetesen használhatóak adott eredményhalmaz megszerzésére is. Tegyük fel, hogy létrehoztuk a következő segédmetódust, amelyet kilépés előtt meghívunk a `main()` metódusból (ez a módszer paraméterként vár egy `AutolotDatabase` példányt):

```
static void ShowOnlyBimmers(AutolotDatabase db)
{
    Console.WriteLine("***** only BMWs *****\n");

    // Megkapjuk a BMW-ket.
    var bimmers = from s in db.Inventory
        where s.Make == "BMW"
        orderby s.CarID
        select s;

    foreach (var c in bimmers)
    {
        Console.WriteLine(c.ToString());
    }
}
```

A 24.3. ábra mutatja az első LINQ to SQL példánk kimenetét.



24.3. ábra: Előb példánál is a LINQ to SQL használatában

Forráskód A SimpleInqToSqlApp kódfájlokat a forráskódkönyvtár 24. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xiv. oldalát.

A [Table] és [Column] attribútumok: további részletek

Az entitásosztályok különböző attribútumokkal rendelkeznek, amelyeket a LINQ to SQL használ, hogy le tudja fordítani az objektumokon végzett lekérdezést adatbázis SQL-lekérdezésekre. Csak a [Table] és a [Column] attribútumokat fogjuk használni; mindazonáltal további attribútumok is léteznek azoknak a metódusoknak a megjelölésére, amelyek végrehajják az SQL beszürt, módosító és törölő parancsait. Ezenfelül a LINQ to SQL attribútumok mind-egyike meghatároz egy olyan tulajdonságkészletet, amely további részletezi, hogy a LINQ to SQL futtatómotor hogyan dolgozza fel a megjelölt elemet.

A [Table] attribútum nagyon egyszerű, ugyanis egyetlen tulajdonságot definiál: ez a name. Ez teszi lehetővé, hogy elkülönítsük az entitásosztály nevét a fizikai táblázattól.

Ha nem állítjuk be a name tulajdonságot a [Table] attribútum használatakor, a LINQ to SQL feltételezi, hogy az entitásosztály és adatbázis tábla nevei megegyeznek.

A [Column] attribútum kicsivel tartalmasabb, mint a [Table]. Az IsPrimary-key tulajdonságon túl a ColumnAttribute további olyan tagokat definiál, ame-

Ilyek lehetővé teszik, hogy teljes mértékben minősítsük az entitásosztály összes mezőjét, és azt, hogy hogyan képezhetők le a fizikai adatbázisból egy adott oszlopára. A 24.1. táblázat az érdekesebb tulajdonságokat mutatja be.

ColumnAttribute	Jelentés
-----------------	----------

CanBeNull	Ez a tulajdonság azt jelzi, hogy az oszlop tartalmazhat null értéket.
-----------	---

DbType	Az adatmezők deklarációja alapján a LINQ to SQL automati- matikusan kikövetkezteti, milyen tulajdonságokat kell az adatbázismotoroknak átadni. Emiatt általában csak akkor kell közvetlenül beállítani a DbType tulajdonságot, ha di- namikusan hozunk létre adatbázisokat a DataContext-ti- pus CreateDatabase() metódusának használatával.
--------	---

IsDbGenerated	A tulajdonság azt állítja be, hogy az adatbázis automati- kusan generálja egy mező értékét.
---------------	--

IsVersion	Ez a tulajdonság határozza meg, hogy az oszlop adatbázis- időbélyeg, esetleg verziószám-e, vagy sem. A verziószá- mok növekednek, az időbélyegoszlopok pedig frissülnek minden alkalommal, amikor a kapcsolódó sor módosul.
-----------	--

UpdateCheck	Ez a tulajdonság vezérli, hogy a LINQ to SQL hogyan ke- zelje az adatbázis-ültközéseket az optimista konkurencia- szabályozás révén.
-------------	--

24.1. táblázat: A [Column] attribútum néhány tulajdonsága

Entitásosztályok generálása az SqlMetal.exe használatával

Az első LINQ to SQL példánk megtehetően egyszerű volt, részben annak a ténynek köszönhetően, hogy a DataContext típusunk egyetlen adattáblával dolgozott. Elképzelhető azonban, hogy a termékészintű LINQ to SQL alkal-
mazások több, egymással kapcsolatos adattáblával dolgoznak, s ezek mind-
egyike oszlopok tucatjait definiálhatja. Ilyen esetekben unalmas lenne manu-
álisan létrehozni minden egyes kívánt entitásosztályt. Szerencsére kétéle-
módon is megoldhatóak ezeknek a típusoknak az automatikus generálását.

Az első lehetőség az sqlmetal.exe parancssori segédprogram használatát, amelyet a Visual Studio 2008 parancssorából indíthatunk el. Ez az eszköz úgy automatizálja az entításozstályok létrehozását, hogy generál egy megfelelő C#-osztálytípust az adatbázis metaadataiból. Bár az eszköz több parancssori paraméterrel rendelkezik, a 24.2. táblázat csak a lényegesebb kapcsolatokat mutatja be.

sqlmetal.exe parancssori paraméter
Jelentés

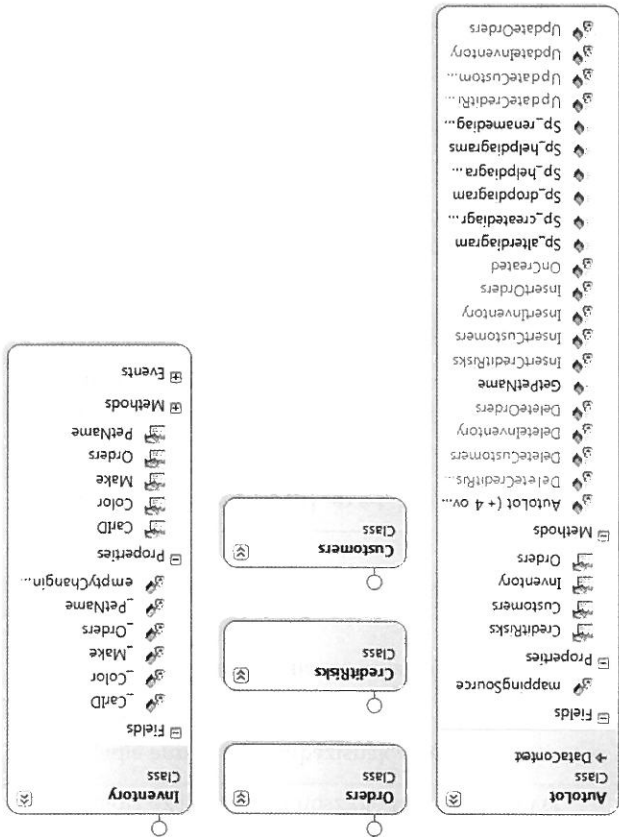
/server	Megadja az adatbázist hosztoló kiszolgáló nevét.
/database	Megadja annak az adatbázisnak a nevét, amelyből ki kell olvasni a metaadatokat.
/user	Megadja a felhasználónévet, amellyel be kell jelentkezni a kiszolgálóra.
/password	Megadja a jelszót, amellyel be kell jelentkezni a kiszolgálóra.
/views	Tájékoztató az sqlmetal.exe-t, hogy létező adatbázisnévzetek alapján generáljon kódot.
/functions	Tájékoztató az sqlmetal.exe-t, hogy készítse el az adatbázisfüggvények megfelelőit.
/procs	Tájékoztató az sqlmetal.exe-t, hogy készítse el a tárolt eljárások megfelelőit.
/code	Tájékoztató az sqlmetal.exe-t, hogy C#-kódként adja ki az eredményeket (vagy VB-kódként, ha beállítottuk a /language kapcsolót).
/language	Megadja, hogy a rendszer milyen nyelven definiálja a generált típusokat.
/namespace	Megadja névteret a generált típusok definiálásához.

24.2. táblázat: Az sqlmetal.exe parancs paraméterei

A következő utasítás például az AutoLot adatbázis minden táblájához generál entításozstályokat, metódust készít a getName tárolt eljáráshoz, és az Autolotdatabase névtérbe csomagolja a generált C#-kódot (természetesen egy sorba kell beírni a Visual Studio 2008 parancssorában):

```
sqlmetal /server:(local)\SQLXPRESS /database:Autolot
        /namespace:AutolotDatabase /code:autoLotDB.cs /spprocs
```

Ha végrehajtottuk az utasítást, hozzunk létre egy új parancssorialkalmazás-projektet `linqwithsqlmetalgeneratedcode` névvel, illetőleg adjunk referenciát a `system.data.linq.dll` szerezélyenyre, és adjuk hozzá az `autolotdb.cs` fájlt a projektkehez a `Project > Add Existing Item` menüpont segítségével. Ezenfelül szúrjunk be egy új osztálydiagramot a projektbe (`Project > Add New Item`), és bontsuk ki a generált osztályokat (lásd a 24.4. ábrát).



24.4. ábra: Az `sqlmetal.exe` által generált entitásszabványok

Van tehát egy új, a `DataContext`-et kibővítő típusunk, amely tartalmazza a tulajdonságokat a megadott adatbázisban lévő összes adattábla számára (továbbá a `getPetName()` tárolt eljárást a megnevező névvel rendelkező nyilvános metódus képviseli). Mielőtt elkezdeneünk használni ezeket az új típusokat, vizsgáljuk meg egy kicsit alaposabban ezt az automatikusan generált kódot.

A generált entitásosztályok

Az sqLmetaL.exe az AutoLot adatbázis minden táblája (Inventory, Customers, Orders, CreditsRisks) számára külön entitásosztályt definiált, és minden oszlopot beágyazott egy tulajdonságba. Továbbá az egyes entitásosztályok két interfészt implementálnak (INotifyPropertyChanged és INotifyProperty-changed), amelyek mindegyike egy-egy eseményt hataroz meg:

```
namespace System.Data.Linq
{
    public interface INotifyPropertyChanged
    {
        // Az esemény akkor következik be, mielőtt egy tulajdonság
        // értéke megváltozik.
        event PropertyChangedEventHandler PropertyChanged;
    }

    namespace System.ComponentModel
    {
        public interface INotifyPropertyChanged
        {
            // Az esemény akkor következik be, amikor egy tulajdonság értéke
            // megváltozott.
            event PropertyChangedEventHandler PropertyChanged;
        }
    }
}
```

Ezek az interfészek összesen két eseményt definiálnak (PropertyChanged és PropertyChanged), amelyek a System.ComponentModel névtérben meghatározott PropertyChangedEventHandler metódusreferenciával működnek együtt. Ez a metódusreferencia meghívható bármilyen metódust, amely egy objektumot vár első paraméterként, míg a második paraméter egy PropertyChangedEventArgs.

A fenti interfészek miatt minden entitásosztály támogatja a következő tagokat:

```
[TableName("Inventory")]
public partial class Inventory : INotifyPropertyChanged,
    INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    public event PropertyChangedEventHandler PropertyChanged;
    ...
}
```

Ha megvizsgáljuk a három entitásosztály tulajdonságainak megvalósítását, láthatjuk, hogy a set hatókör kivált minden eseményt azok számára, akik a tulajdonság értékének a változását figyelik. Példaként nézzük meg az Inventory típus PetName tulajdonságát:

Kapcsolatok definiálása entitásosztályok használatával

Azon túl, hogy tulajdonságokat és segédmezőket definiál az adattábla oszlopainak megjelölésére, az `sqlmeta1.exe` segédprogram az `EntitySet<T>` típusalattal definiált hármas, egymáshoz kapcsolódó belső táblát, amelyeket el-sődleges és idegen kulcsok kapcsolnak össze (lásd a 22. fejezet). Ahelyett, hogy SQL-központú join szintaxis megírására kényyszerüljünk a táblázatok közötti navigáláshoz, a LINQ to SQL lehetővé teszi, hogy az objektumcentrikus C# pont operátorral navigáljunk.

A set hatókör meghívja az `onPetNamechanged()` és az `onPetNamechanged()` metódusokat az entitásosztály-típuson, hogy valójában ezek váltsák ki az eseményeket. Ezeket a tagokat azonban *reszleges metódusokként* definiáltuk, amelyek könnyű eseménykezelést hajtának végre (lásd az előző kötet 13. fejezetét), lehetővé téve az érdekelteket hívó számára, hogy biztosítsa a szükséges megvalósítást (ha nem, akkor a fordítás során a rendszer eltávolítja őket a típusdefiniációból):

```
[Column(Storage="_PetName", DbType="VarChar(50)")]
public string PetName
{
    get
    {
        return this._PetName;
    }
    set
    {
        if ((this._PetName != value))
        {
            this.OnPetNamechanged(value);
            this.SendPropertyChanging();
            this._PetName = value;
            this.OnPetNamechanged();
        }
    }
}
```


Az illesztett táblakapcsolat érdekében a szülő-entitásosztály tulajdonságaként hivatkozhat a gyermektáblára. Ezt a tulajdonságot az [Association] attribútummal jelöljük, hogy létrehozzunk egy *társítási kapcsolatot* a táblák között a megegyező oszlopértékek révén. Nézzük meg például a customer típus számára generált (részleges) kódot, amelyben bármennyi rendelés lehet:

```
[Table(Name="Customers")]
public partial class Customers :
    INotifyPropertyChanged, INotifyPropertyChanging
{
    private EntitySet<Orders> _orders;

    [Association(Name="FK_Orders_Customers", Storage="Orders",
        otherkey="CustID", deleteRule="NO ACTION")]
    public EntitySet<Orders> orders
    {
        get { return this._orders; }
        set { this._orders.Assign(value); }
    }
    ...
}
```

A LINQ to SQL futtatómotor számára ezen a ponton lesz nyilvánvaló az, hogy az orders tulajdonság olyan tag, amely lehetővé teszi a Customers táblából az Orders táblába történő navigálást az otherkey tulajdonság által definiált oszlop révén. Az EntitySet<T> tagválasztózt arra használjuk, hogy jelképezze az adott kapcsolatot „egy a többhöz” (one-to-many) természetét.

Az erősen típusos DataContext

Az sqlmetal.exe által generált kód utolsó figyelmeztető elem a DataContext-tól származtatott típus. Az előző példában létrehozott AutoloDatabase osztályhoz hasonlóan minden táblát Table<T>-kompatibilis tulajdonság jelképez. Ez az osztály emellett rendelkezik olyan konstruktorokkal, amelyek egyike paraméterként várja az IDbConnection osztályt megvalósító objektumot, amely egy ADO.NET-kapcsolatobjektumot jelképez (a LINQ to SQL és az ADO.NET-típusok keverhetők egyetlen alkalmazáson belül). Emellett ez a DataContextból származó osztály mutatja, hogyan dolgoztunk az adatbázisban meghatározott tárolt eljárásokkal. Mivel megadtuk az /sprocs kapcsolót a sqlmetal.exe-nek, találunk egy GetPetName() nevű metódust:

```

[Function(Name="dbo.GetPetName")]
[return: Parameter(DbType="Int")]
public int GetPetName([Parameter(DbType="Int")]
    System.Nullable<int> carID,
    [Parameter(DbType="char(10)")]
    ref string petName)
{
    IExecuteResult result = this.ExecuteMethodCall(this,
        ((MethodInfo)(MethodInfo.GetCurrentMethod()))),
        carID, petName);
    petName = ((string)(result.GetParameterValue(1)));
    return ((int)(result.ReturnValue));
}

```

A `GetPetName()` metódus a `[Function]` attribútumokkal és a `[return:]` attribútumminősítővel rendelkezik. Továbbá az összes paraméter rendelkezik a `[Parameter]` attribútummal. A megvalósítás az örökölte `ExecuteMethodCall()` metódust (és reflexiós szolgáltatást) használja, hogy biztosítsa a hívó számára a tároltejelrás-hívást, valamint az eredmény visszaadását.

A generált típusok használata

A továbbiakban vizsgáljuk meg a `Program` típus alábbi implementációját, amely meghívja a tároltejelrásunkat:

```

class Program
{
    const string cnstr =
        @"Data Source=(local)\sql EXPRESS;Initial Catalog=Autolot;" +
        "Integrated Security=true";

    static void Main(string[] args)
    {
        Console.WriteLine("***** More Fun with Linq to SQL *****\n");
        Autolot carsDB = new Autolot(cnstr);
        InvokeStoredProc(carsDB);
        Console.ReadLine();
    }

    private static void InvokeStoredProc(Autolot carsDB)
    {
        int carID = 0;
        string petName = "";
        Console.WriteLine("Enter ID: ");
        carID = int.Parse(Console.ReadLine());
    }
}

```

```
// Meghívja a tárolt eljárást, és kírja a becenevet.
carpdb.getPetname(carID, ref petname);
console.WriteLine("Car ID {0} has the petname: {1}",
    carID, petname);
}
```

A LINQ to SQL teljes mértékben elrejt előlünk az alapul szolgáló tárolt eljárást logikáját. Itt nem kell manuálisan sqlCommand objektumot létrehozni, feltölteni a paramétergyűjteményt vagy meghívni az executeNonQuery() metódust. Ehelyett egyszerűen meghívjuk a DataContextból származtatott típus getPetname() metódusát. A kimeneti paraméterek referenciaparaméterként jelennek meg, ezért a C# ref kulcsszó használatával kell meghívni őket. Tetelezzük fel, hogy van egy másik segédüggyényünk (amelyet szintén a main() metódusból hívunk meg), ennek neve PrintOrderForCustomer(). Ez a metódus kír néhány megrendelésinformációt a megadott ügyféllel kapcsolatosan, valamint az ügyfél keresztnévét:

```
static void PrintOrderForCustomer(AutoLot carsDB)
{
    int custID = 0;
    Console.WriteLine("Enter customer ID: ");
    custID = int.Parse(Console.ReadLine());

    var customerOrders =
        from cust in carsDB.Customers
        from o in cust.Orders
        where cust.CustID == custID
        select new { cust, o };

    Console.WriteLine("***** Order Info for Customer ID: {0}. *****",
        custID);
    foreach (var q in customerOrders)
    {
        console.WriteLine("{0} {1} is order ID # {2}.",
            q.cust.FirstName.Trim(),
            q.cust.LastName.Trim(),
            q.o.OrderID);
        console.WriteLine("{0} bought car ID # {1}.",
            q.cust.FirstName.Trim(), q.o.CarID);
    }
}
```

A 24.5. ábra mutatja azt a kimenetet, amikor az 1-es azonosítójú ügyféllel kapcsolatban kérdezzünk le.

Entitáosztályok létrehozása a Visual Studio 2008 használatával

Végül hozzunk létre új parancssori konzolalkalmazást `LinqToSqlCrud` néven, valamint adjunk hozzá egy referenciát a `System.Data.Linq.dll` szerelvényre. Ezáltal ahelyett, hogy az `sqlmetal.exe` segítségével generálnánk az entitáosztályainkat, a Visual Studio 2008-ra bizzuk a „píszkos munkát”. Ehhez válasszunk a `Project > Add New Item` menüpontot, majd adjunk hozzá egy új LINQ to SQL Classes elemet `Autolotobjects` néven (lásd a 24.6. ábrát).

Forráskód A `LinqWithSqlMetalGeneratedCode` kódfájlokat a forráskódkönyvtár 24. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

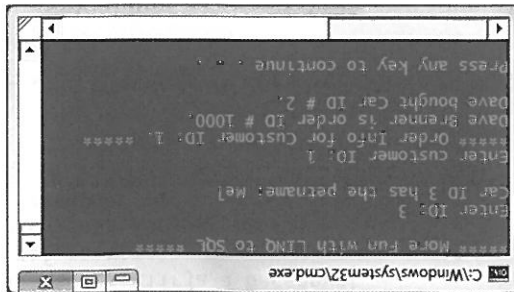
```
SELECT [t0].[FirstName], [t0].[LastName], [t0].[CustID],
       [t1].[OrderID], [t1].[CarID], [t1].[CustID] AS [CustID2]
FROM [Customers] AS [t0], [Orders] AS [t1]
WHERE ([t0].[CustID] = @p0) AND ([t1].[CustID] = [t0].[CustID])
```

Ha újra lefuttatjuk a programot, azt tapasztalhatjuk, hogy a lekérdezéskifejezésünk sztringformátumú értéke az alapul szolgáló SQL-lekérdezést mutatja:

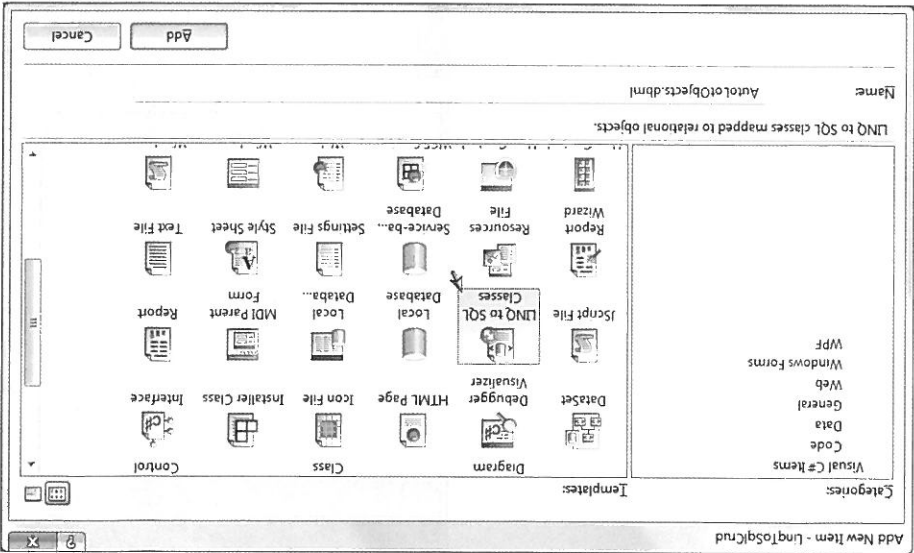
```
console.WriteLine("\ncustomerOrders as a string: {0}",
customerOrders);
```

A LINQ to SQL előnye ismételten az, hogy következetes, objektumalapú modellel használatával dolgozhatunk a relációs adatbázisokkal. A LINQ-lekérdezéskifejezésünk megvilágításához adjuk hozzá a következő utasítást a `PrintOrderForCustomer()` módszer végéhez:

24.5. ábra: Egy adott ügyfél rendelési információinak kitirása



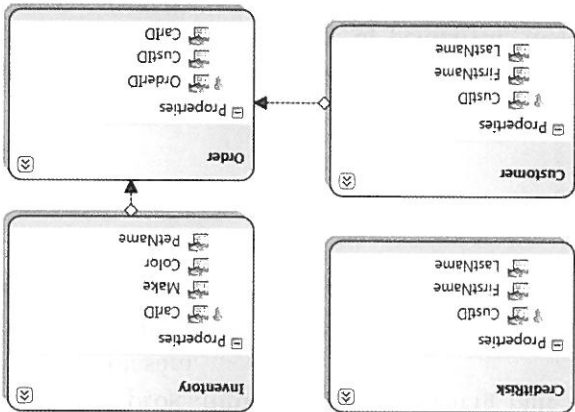
24. fejezet: A LINQ API programozása



24.6. ábra: A LINQ to SQL Classes elem ugyanazokat a feladatokat végzi el, mint az sqmtool.exe

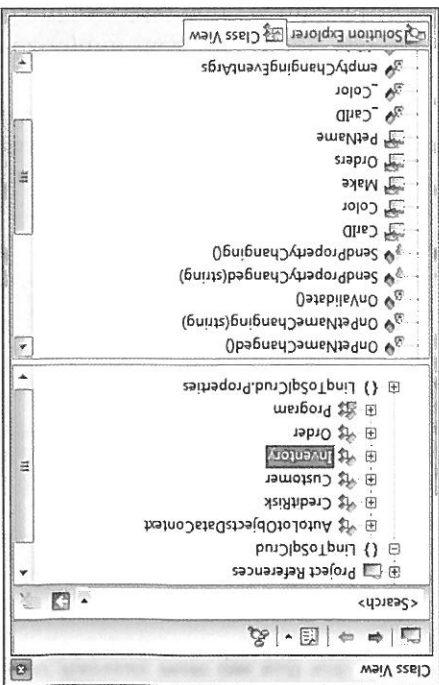
Nyissuk meg a Server Explorer-t, és győződjünk meg róla, hogy van aktív kapcsolatunk az Autolo adatbázissal (ha nincs, akkor jobb gombbal kattintunk a Data Connections ikonra, és válasszuk az Add Connection-t). Jelöljük ki az összes táblát, és húzzuk át őket a LINQ to SQL tervezőfelületre. Ekkor a

képernyő a 24.7. ábrához hasonlít.



24.7. ábra: Entitásosztályok létrehozása a LINQ to SQL tervezővel

Miután lefordítottuk, a Solution Explorerben nyissuk meg a kapcsolódó *.cs fájlt (lásd a 24.8. ábrát).



24.8. ábra: A generált névter a tartalmazott típusokkal

Végignézve a generált C#-kódot, látható, hogy ugyanezt az általános kódot generálta az `sqlmetal.exe` parancssori segédprogram is. Továbbá a vizuális LINQ to SQL tervező hozzáadott a projektünkhez egy `app.config` fájlt a szükséges kapcsolatszttringadatok tárolására.

Most, hogy minden generált osztály rendelkezésre áll, nézzük meg a program osztályt, amely az Inventory táblán bemutatja az adatok beszurását, módosítását és törlését.

Új elemek beszurása

Ahhoz, hogy új elemet szúrjunk be egy relációs adatbázisba, csak annyit kell tennünk, hogy létrehozzuk az adott entitásosztály egy új példányát, majd hozzáadjuk a `datacontext` által fenntartott `table<T>` típushoz, és meghívjuk rajta a `submitchanges()` metódust. Az alábbi `insertNewCars()` metódus hozzáad két elemet az Inventory táblához. Az első megközelítés közvetlenül beállítja az Inventory entitásosztály minden mezőjét, míg a második a tömörebb objektuminitializáló szintaxist használja:

```

static void InsertNewCars(AutoLotobjectsDataContext ctx)
{
    Console.WriteLine("***** Adding 2 cars *****");
    int newCarID = 0;
    Console.WriteLine("Enter ID for Betty: ");
    newCarID = int.Parse(Console.ReadLine());

    // Új sor hozzáadása "terjengős" szintaxisal.
    Inventory newCar = new Inventory();
    newCar.Make = "Yugo";
    newCar.Color = "Pink";
    newCar.PetName = "Betty";
    newCar.CarID = newCarID;
    ctx.Inventoryes.InsertOnSubmit(newCar);
    ctx.SubmitChanges();

    Console.WriteLine("Enter ID for Henry: ");
    newCarID = int.Parse(Console.ReadLine());

    // Új sor hozzáadása a "tömör" objektuminitializáló szintaxis
    // használatával.
    newCar = new Inventory { Make = "BMW", Color = "Silver",
    PetName = "Henry", CarID = newCarID };
    ctx.Inventoryes.InsertOnSubmit(newCar);
    ctx.SubmitChanges();
}

```

Létező elemek módosítása

Egy elem módosítása meglehetősen egyszerű. A LINQ-lekérdezés alapján nyerjük ki az első elemet, amely megfelel a keresési feltételnek. Miután frissítettük az objektum állapotát, hívjuk meg a `SubmitChanges()` metódust.

```

static void UpdateCar(AutoLotobjectsDataContext ctx)
{
    Console.WriteLine("***** updating color of 'Betty' *****");
    // Módosítsuk Betty színét világos rózsaszínné.
    var betty = (from c in ctx.Inventoryes
    where c.PetName == "Betty"
    select c).First();
    betty.Color = "Green";
    ctx.SubmitChanges();
}

```

Létező elemek törlése

Végül, ha törölni szeretnénk egy elemet a relációsadatbázis-táblából, egyszerűen hozzunk létre egy LINQ-lekérdezést, hogy megtaláljuk azt az elemet, amelyre nincs szükségünk, és távolítsuk el a DataContext megfelelő `Table<T>` tagváltozójából a `DeleteOnSubmit()` metódus segítségével. Ezután ismét a `SubmitChanges()` metódust kell megívni.

```
static void DeleteCar(AutoLotObjectContext ctx)
{
    int carToDelete = 0;
    Console.WriteLine("Enter ID of car to delete: ");
    carToDelete = int.Parse(Console.ReadLine());

    // Adott autó eltávolítása.
    ctx.Inventories.DeleteOnSubmit((from c in ctx.Inventories
    where c.CarID == carToDelete
    select c).First());
    ctx.SubmitChanges();
}
```

Most már meghívhatunk minden metódust a `main()` metódusból, hogy ellenőrizzük a kimenetet:

```
static void Main(string[] args)
{
    Console.WriteLine("***** CRUD with LINQ to SQL *****\n");
    const string cnstr =
        @"Data Source=(local)\SQLEXPRESS;Initial Catalog=AutoLot;" +
        "Integrated Security=True";
    AutoLotObjectContext ctx =
        new AutoLotObjectContext(cnstr);
    InsertNewCars(ctx);
    UpdateCar(ctx);
    DeleteCar(ctx);
    Console.ReadLine();
}
```

Ezzel befejeztük a LINQ to SQL áttekintését. Természetesen az itt leírtaknál jóval többet el lehetne mondani; ám ennyi elegendő ahhoz, hogy belismerüljünk a részletekbe.

Forráskód A `LinqToSqlCrud` kódfrágákat a forráskódkönyvtár 24. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xiv. oldalát.

XML-dokumentumok kezelése a LINQ to XML használatával

Végezetül bemutathatjuk a LINQ to XML szerepét, amely azt lehetővé teszi, hogy LINQ-lekérdezéseket alkalmazzunk XML-dokumentumokon. Az XML-adatok reprezentációja milyen beágyazódott a .NET keretrendszerbe.

Az alkalmazások és a webalapú konfigurációs fájlok XML-ként tárolják az adatokat. Az ADO.NET adatbázisok (vagy betölthetők) XML-ként az adatokat. A Windows Presentation Foundation egy XML-alapú nyelvet (XAML) használ az asztali felületek ábrázolására, a Windows Communication Foundation (valamint a .NET-remoting API) szintén számos beállításról tárol olyan jól formázott sztringként, amelyet XML-nek hívunk.

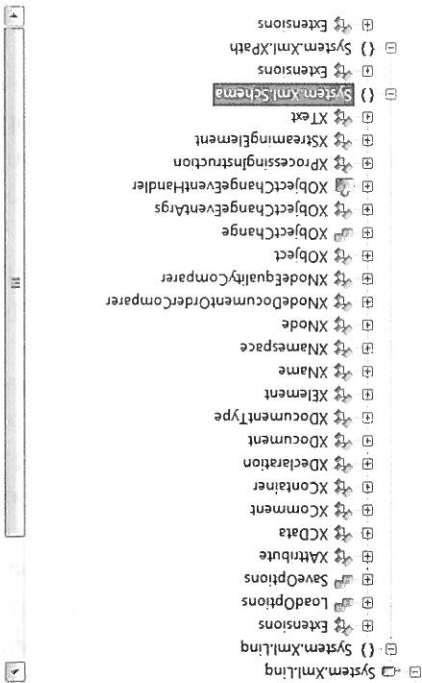
Noha az XML valóban mindenhol jelen van, programozása mindig is unalmas, terjedős és összetett volt, ha valaki nem volt tisztában számos XML-technológiával (XPath, XQuery, XSLT, DOM, SAX stb.). A .NET platform kezdete óta a Microsoft egy olyan speciális szerelvényt biztosít az XML-dokumentumok programozásához, amelynek `system.xml.dll` a neve. Ezen a bináris fájlban belül számos névtér és típus található az XML-programozási módszerekhez, valamint néhány .NET-specifikus XML API, mint például az `XmlReader/XmlWriter` modellek.

LINQ to XML: egy jobb DOM

Ahogy a LINQ to SQL célja az, hogy a relációs adatbázisok kezelését közvetlenül a .NET programozási nyelvekbe integrálja, úgy a LINQ to XML is ezt a célt tűzi ki az XML-adatok feldolgozása terén. Nemcsak olyan eszközként használhatjuk a LINQ to XML-t, amellyel LINQ-lekérdezésekkel megszereshetünk adatreszhalmozásokat egy meglevő XML-dokumentumból, hanem ugyanezt az API-t használhatjuk XML-adatok létrehozására, módosítására és elemzésére is. Emiatt a LINQ to XML-re gondolhatunk úgy is, mint egy „jobb DOM” programozási modellre. Emellett, ahogy a LINQ to SQL együttműködhet az ADO.NET-típusokkal, a LINQ to XML is dolgozhat a `system.xml.dll` szerelvények számos tagjával.

A System.Xml.Linq névtér

Az alapvető LINQ to XML szerelvény (System.Xml.Linq.d11) nagyon kevés típust definiál a következő három különböző névtérben: System.Xml.Linq, System.Xml.Schema és System.Xml.XPath (lásd a 24.9. ábrát).



24.9. ábra: A System.Xml.Linq.all névtérei

Az alapnévtér, a System.Xml.Linq olyan jól kezelhető típuskészletet tartalmaz, amely egy XML-dokumentum különböző aspektusait jelképezi (elemeket és attribútumait, XML-névtérreket, XML-megjegyzéseket, feldolgozási utasításokat stb.). A 24.3. táblázat mutatja be a System.Xml.Linq névtér alapvető tagjait.

A System.Xml.Linq tagja		Jelölés
XAttribute	Egy adott XML-elem XML-attribútumát jelképezi.	
XComment	XML-megjegyzést jelképez.	
XDeclaration	Egy XML-dokumentum nyitó deklarációját jelképezi.	
XDocument	Egy XML-dokumentum összességét jelképezi.	

Az inventory XElement objektumának konstruktora valójában további XElement és attribute objektumok fája. Utasításaink gondos behúzásával a forráskód megjelenése hasonló magához az XML-dokumentumhoz. Ha a main() metódusból meghívjuk a metódusunkat, a 24.10. ábrán látható kimenetet fogjuk kapni.

```
static void createFunctionalXMLElement()
{
    // "Funkcionális" megközelítés egy
    // XML-elem létrehozásához a memóriában.
    XElement inventory =
        new XElement("Inventory",
            new XElement("Car", new Attribute("ID", "1"),
                new XElement("Color", "Green"),
                new XElement("Make", "BMW"),
                new XElement("PetName", "Stan")
            )
        );
    // A toString() meghívása az XElementünkön.
    Console.WriteLine(inventory);
}
```

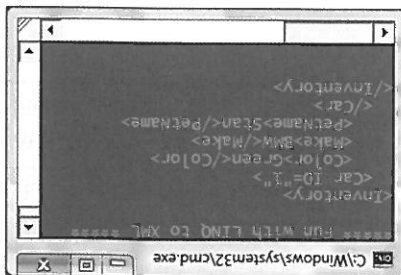
valósítással:
Program osztályunkhoz createFunctionalXMLElement() névvel, az alábbi megadatok formátumába. Ennek bemutatásához adjunk hozzá egy metódust a programozási modell leképezhető majdnem közvetlenül a jól formált XML-Ellel nemcsak nagymértékben csökkenni a szükséges kód mennyisége, de a val, a LINQ to XML lehetővé teszi, hogy ezt "DOM-mentesen" tegyük meg. Ilyett, hogy létrehoznánk egy dokumentumot a nagyon terjedős DOM API-XML-dokumentumot LINQ-val csak *funkcionális módon* lehet kezelni. Így ahe- Az eredeti .NET XML programozási modellel (system.xml.dll) ellentétben

XML-dokumentumok létrehozása programozottan

Ennek (és más) típusoknak a vizsgálatához hozzunk létre egy parametreszt kon-zolalkalmazást LinqToXmlBasis névvel, majd importáljuk a System.Xml.Linq névteret a kezdeti kódjába.

24.3. táblázat: A System.Xml.Linq névter néhány tagja

XMLElement	Egy adott elemet jelképez egy XML-dokumentumban.
XNamespace	Nagyon egyszerű módszert biztosít XML-névterek definiálására és hivatkozására.



24.10. ábra: XML-dokumentum létrehozásának funkcionális megközelítése

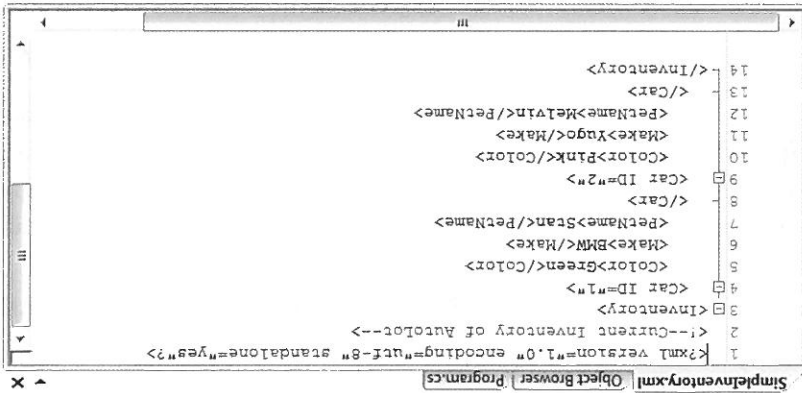
Ahhoz, hogy egy teljes XML-dokumentumot hozzunk létre a memóriában (megjegyzésekkel, feldolgozási utasításokkal, nyitó deklarációkkal stb.), megadhatjuk az objektumfát egy `xdocument` típus konstruktorába. Gondoljunk végig a következő `createFunctionalXmlDoc()` metódust, amely először létrehoz egy dokumentumot a memóriában, majd elmenti egy helyi fájlba:

```
static void createFunctionalXmlDoc()
{
    xdocument inventoryDoc =
        new xdocument(
            new XElement("Inventory",
                new XElement("Car", new XAttribute("ID", "1"),
                    new XElement("Color", "Green"),
                    new XElement("Make", "BMW"),
                    new XElement("PetName", "Stan")),
                new XElement("Car", new XAttribute("ID", "2"),
                    new XElement("Color", "pink"),
                    new XElement("Make", "Yugo"),
                    new XElement("PetName", "Melvin"))
            );
    // A dokumentum megjelenítése és mentése a lemezre.
    Console.WriteLine(inventoryDoc);
    inventoryDoc.Save("SimpleInventory.xml");
}
```

A 24.11. ábra mutatja a Visual Studio 2008-ban megnyitott `SimpleInventory.xml` fájl.

Az `XElement` és az `XDocument` típusok meghatároznak egy olyan konstruktor, amely `xname` típust vessz fel első paraméterként és objektumok paramétertömbjét másodikként. Az `xname` típust a LINQ to XML-ben arra használjuk, hogy ábrázoljuk vele (nyilvánvalóan) a létrehozandó elem nevét, míg az ob-

jeltek paramétertömbje tartalmazhat bármennyi további LINQ to XML típust (xcomment, xprocessinginstruction, xelement, xattribute stb.), illetve egy szerű sztringeket (elemtartalomhoz) vagy az Ienumerable típust implementáló objektumot.



24.11. ábra: A rögzített XML-dokumentum

Dokumentumok létrehozása LINQ-lekérdezésekből

Ami az utolsó pontot illeti, tétellezzük fel, hogy van egy névtelen típusból álló névtelen tömbünk, amely egy egyszerű car osztályt képvisel. Készíthetünk egy LINQ-lekérdezést, amely kiválogat minden név/érték párt a tömbből, hogy dinamikusan létrehozhassunk egy új XElement típust:

```
static void CreateXmlDocFromArray()
{
    // Anonymous típusok névtelen tömbjének létrehozása.
    var data = new [] {
        new { PetName = "Melvin", ID = 10 },
        new { PetName = "Pat", ID = 11 },
        new { PetName = "Danny", ID = 12 },
        new { PetName = "Clunker", ID = 13 }
    };

    // Most listázzuk az egész tömböt, hogy létrehozzunk
    // egy XElement objektumot.
    XElement vehicles =
        new XElement("Inventory",
            from c in data
            select new XElement("Car",
                new XElement("ID", c.ID),
                new XElement("Name", c.PetName),
                new XElement("Make", c.Make),
                new XElement("Color", c.Color)
            )
        );
}
```

A LINQ to XML-lel kapcsolatban a következőkben azt vizsgáljuk meg, hogy hogyan navigálhatunk egy adott dokumentumban ahhoz, hogy meghatározzuk bizonyos elemek/attribútumok helyét.

Navigálás egy memóriában lévő dokumentumban

Forráskód A `LinqToXml` Basics kódfájlokat a forráskódkönyvtár 24. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xiv. oldalát.

```
}
Console.WriteLine(myDoc);
XDocument myDoc = XDocument.Load(@"SimpleInventory.xml");
// A SimpleInventory.xml fájl betöltése.

Console.WriteLine(newElement);
XElement newElement = XElement.Parse(myElement);
</Car>;
<Make>Yugo</Make>
<Color>Yellow</Color>
@"<Car ID = '3'>
string myElement =
// XElement létrehozása stringből.
}
static void LoadExampleXML()
```

mindkét megközelítést bemutatja: adatokból vagy külső fájlokból. Vizsgáljuk meg a következő metódust, amely sokat, amelyek lehetővé teszik egy XML-objektummodell feltöltését string-Az `XElement` és az `XDocument` típusok támogatják a `Load()` és `Parse()` metódusokat, amelyek lehetővé teszik egy XML-objektummodell feltöltését string-

XML-tartalom betöltése és elemzése

```
}
Console.WriteLine(vehicles);
);
)
select new XElement("Car",
    new XAttribute("ID", c.ID),
    new XElement("PetName", c.PetName)
)
```

A LINQ to XML objektummodell számos olyan metódust biztosít, amelyekkel programozottan navigálhatunk egy dokumentumban, ugyanígy a LINQ-lekérdezéskifejezéseket is használhatjuk erre a célra.

Rövid példaként először hozzuk létre új parancssori konzolalkalmazást `NavigationWithLinqToXml` néven, és importáljuk a `System.Xml.Linq` névtérrel. Ezután adjunk hozzá egy új XML-dokumentumot aktuális projektünkhöz `Inventory.xml` névvel, ez bejegyzések kis gyűjteményét tárolja a gyöke `<Inventory>` elemen belül, amelynek lehetséges tartalma a következő:

```
<?xml version="1.0" encoding="utf-8"?>
<Inventory>
  <Car carID="0">
    <make>Ford</make>
    <color>Blue</color>
    <petName>Chuck</petName>
  </Car>
  <Car carID="1">
    <make>VW</make>
    <color>Silver</color>
    <petName>Mary</petName>
  </Car>
  <Car carID="2">
    <make>Yugo</make>
    <color>Pink</color>
    <petName>Gipper</petName>
  </Car>
  <Car carID="55">
    <make>Ford</make>
    <color>Yellow</color>
    <petName>Max</petName>
  </Car>
  <Car carID="98">
    <make>BMW</make>
    <color>Black</color>
    <petName>Zippy</petName>
  </Car>
</Inventory>
```

Jelöljük ki ezt a fájlt a `Solution Explorer`-ben, és használjuk a `Properties` ablakot, hogy állítsuk a `Copy to Output Directory` tulajdonságot `Copy Always` értékre (ezzel biztosítjuk, hogy a fájl másolata bekerüljön a `Bin` mappába). Végül módosítsuk a `main()` metódust, hogy betöltsse ezt a fájlt a memóriába az `XElement.Load()` metódus használatával. A helyi `doc` változót különböző segédmetódusoknak adjuk át, hogy különböző módszerekkel módosítsuk az adatokat:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with LINQ to XML *****\n");
    // Az Inventory.xml dokumentum betöltése a memóriába.
    XElement doc = XElement.Load("Inventory.xml");

    // Ezek mindegyikét mi fogjuk létrehozni...
    PrintAllPetNames(doc);
    Console.WriteLine();
    GetAllFords(doc);
    Console.ReadLine();
}

```

A `PrintAllPetNames()` módszer bemutatja az `XElement` használatát. A `Descendants()` módszer lehetővé teszi egy olyan adott aelem meghatározását, amelyhez el akarunk navigálni annak érdekében, hogy LINQ-lekérdezéskifejezést alkalmazzunk. Ekkor kiválasztunk minden `PetName` értéket, és kiírjuk a tartalmukat a konzolra:

```

static void PrintAllPetNames(XElement doc)
{
    var petNames = from pn in doc.Descendants("PetName")
                    select pn.Value;

    foreach (var name in petNames)
        Console.WriteLine("{0}", name);
}

```

A `GetAllFords()` módszer nagyon hasonló. A bejövő `XElement` alapján definiálunk egy `where` operátort, és kiválasztjuk az összes `XElement`-et, ahol a `make` elem értéke „Ford”:

```

static void GetAllFords(XElement doc)
{
    var fords = from c in doc.Descendants("Make")
                where c.Value == "Ford"
                select c;

    foreach (var f in fords)
        Console.WriteLine("Name: {0}", f);
}

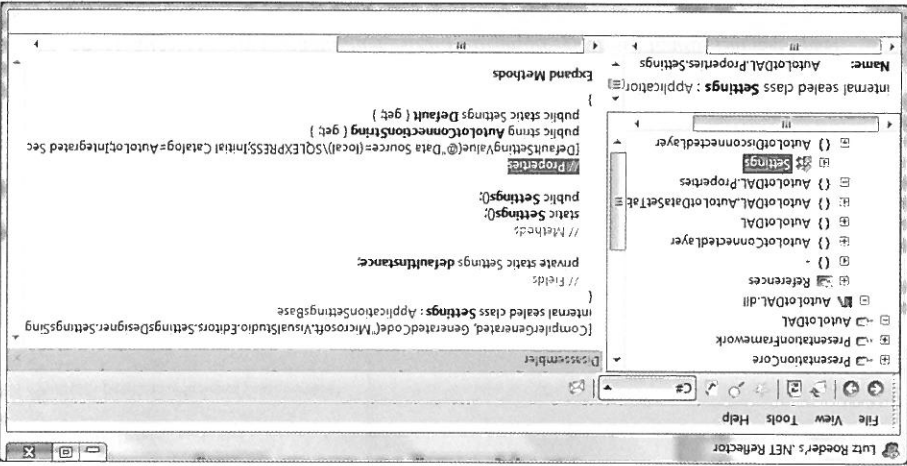
```

A 24.12. ábra mutatja a program kimenetét.

Egészen pontosan a megjelölésen hosszú VisualDataGridVizualizáció.Properties.Settings.AutoLoadConnectionString értéket látjuk a kapcsolatsztring nevéként. Még furcsább az a tény, hogy ha átfutjuk a generált kódot, nem találunk referenciát a ConfigurationManager típusra, hogy kiolvashassuk az értéket a <connectionstring> elemről. Azt azonban láthatjuk, hogy az automatikusan generált adaillesztő objektumot (lásd részletesen később) részben a következő privát segédfüggvény meghívásával hoztuk létre:

```
private void InitConnection()
{
    this._connection =
        new global::System.Data.SqlClient.SqlConnection();
    this._connection.ConnectionString = global::
        VisualDataGridVizualizáció.Properties.Settings.Default.
        AutoLoadConnectionString;
}
```

A connectionString tulajdonságot a settings.Default meghívásával állítjuk be. Minden Visual Studio 2008 projekt típus fenntart az egész alkalmazásra kiterjedő beállításokat, amelyek metaadatként égneke bele a szerelvénybe, amikor lefordítjuk az alkalmazást. Ha megnyitunk egy lefordított alkalmazást a reflector.exe használatával (lásd az előző kötet 2. fejezetét), akkor megnevezhetjük ezt a belső típust (lásd a 23.22. ábrát).



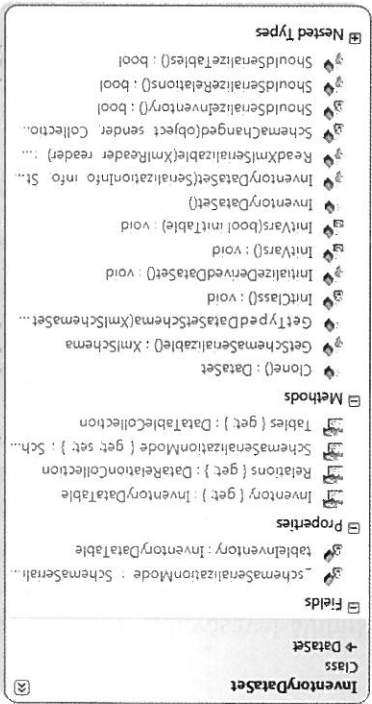
23.22. ábra: A Settings objektum tartalmaz egy beágyazott kapcsolatsztringértéket

A fentiek ismeretében lehetséges lenne úgy telepíteni az alkalmazást, hogy nem adjuk oda a *.config fájlt, mivel az alapértelmezés szerint a beágyazott értéket használjuk, ha nincsen jelen kliensoldali *.config fájl.

Megjegyzés A Visual Studio 2008 beállítási programozási modellek teljes ismertetése túlmutat a fejezet hatáskörén. Ha szeretnének többet megtudni ezekről, olvassuk el a „NET Framework 3.5 SDK dokumentációjának „Managing Application Settings” című témakörét.

A generált DataSet vizsgálata

Nézük meg a generált kód néhány alapvető aspektusát. Először szűnjünk be egy új osztálydiagramot a projektbe, amelyhez válasszuk ki a projekt ikonját a Solution Explorerben, majd kattintsunk a View Class Diagram gombra. A be-menet alapján a varázsló létrehoz egy új dataset típust, amelynek ebben az esetben InventoryDataSet a neve. Az osztály számos hasznos tagot definiál, amelyek közül az Inventory tulajdonság a legfontosabb (lásd a 23.23. ábrát).



23.23. ábra: A Data Source Configuration Wizard létrehozott egy erősen típusos DataSet típust

Ha a Solution Explorerben kattintunk az InventoryDataSet.xsd fájlra, akkor betöltjük a Visual Studio 2008 DataSet Designer eszközt (erről részlete-ebben később lesz szó). Ha ezen a tervezőn belül bárhova kattintunk a jobb egérgombbal, és kiválasztjuk a View Code lehetőséget, megtehetjük a következőt:

```
public partial class InventoryDataSet {
    {
    }
}
```

Az események a tervező által karbantartott fájlban (InventoryDataSet.Designer.cs) belül zajlanak. Ha a Solution Explorerrel megnyitjuk ezt a fájlt, látható, hogy az InventoryDataSet valójában kibővíti a DataSet osztálytípust. Amikor mi (vagy a varázsló) létrehozunk egy olyan osztályt, amely kibővíti a DataSet típust, akkor *erősen típusos* datasetet valósítunk meg. Az erősen típusos dataset objektumok használatának egyik előnye az, hogy több olyan tulajdonságot tartalmaznak, amelyek közveletlenül az adatbázistáblák neveire képezhetők le. Így ahelyett, hogy bele kéne ásnunk magunkat a táblák gyűjteményébe a Tables tulajdonság segítségével, egyszerűen használhatjuk az Inventory tulajdonságot. Vizsgáljuk meg az alábbi kódreszletet, amely az érthetőség kedvéért megjegyzéseket is tartalmaz:

```
// Ez mind a tervező által létrehozott kód.
public partial class InventoryDataSet : global::System.Data.DataSet
{
    // InventoryDataTable típusú tagváltozó.
    private InventoryDataTable tableInventory;

    // Minden konstruktor meghív egy InitClass() segédmódszert.
    public InventoryDataSet()
    {
        ..
        this.InitClass();
    }

    // Az InitClass() előkészíti a DataSetet, és hozzáadja az
    // InventoryDataTable osztályt a Tables gyűjteményhez.
    private void InitClass()
    {
        this.DataSetName = "InventoryDataSet";
        this.Prefix = "";
        this.Namespace = "http://tempuri.org/InventoryDataSet.xsd";
        this.EnforceConstraints = true;
        this.SchemaSerializationMode =
            global::System.Data.SchemaSerializationMode.IncludeSchema;
        this.tableInventory = new InventoryDataTable();
        base.Tables.Add(this.tableInventory);
    }
}
```

```
// Az írásvédett Inventory tulajdonság visszaadja
// az InventoryDataTable tagváltozót.
public InventoryDataTable Inventory
{
    get { return this.tableInventory; }
}
```

Amellett, hogy egyesíti a DataTable objektum fenntartásának részleteit, a tervező által generált erősen típusos dataset tartalmazhat hasonló logikát, hogy feltárjon olyan dataRelation objektumokat (ezekkel jelenleg nem rendelke-zünk), amelyek a táblák közötti kapcsolatot képviselik.

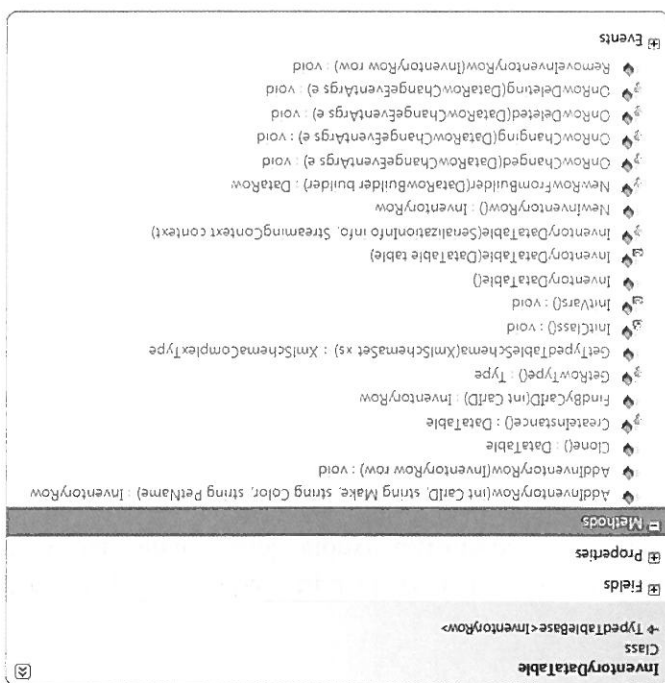
A generált DataTable és DataRow típusok vizsgálata

A fentiekhez hasonlóan, a varázsló létrehozott egy *erősen típusos* DataTable és egy *erősen típusos* DataRow osztályt, amelyek mindegyikét beágyazta az *InventoryDataSet* osztályba. Az *InventoryDataTable* osztály (amely ugyanolyan ti-pusú, mint a most megvizsgált erősen típusos dataset tagváltozója) egy olyan tulajdonságkészletet definiál, amely a fizikai *Inventory* tábla oszlopneve-in alapul (*CarIDColumn*, *ColorColumn*, *MakeColumn*, *PetNameColumn*), valamint egy egyedi indexelőt és egy *count* tulajdonságot, hogy beolvassa a rekordok aktuális számát.

Sokkal érdekesebb, hogy ez az erősen típusos DataTable osztály egy olyan metoduskészletet definiál (lásd a 23.24. ábrát), amely sorok beszúrását, hely-meghatározását és törlését teszi lehetővé a táblában erősen típusos tagok használatával (vonzó alternatíva a *row* és *column* indexelők manuális navigá-lása mellett).

Megjegyzés Az erősen típusos DataTable számos olyan eseményt is definiál, amelyekkel megfigyelhetjük a táblázat adatainak módosításait.

Az egyedi DataRow típus sokkal kevésbé egzotikus, mint a generált dataset vagy DataTable. Ez az osztály kibővíti a DataRow osztályt (lásd 23.25. ábra), és olyan tulajdonságokat nyújt, amelyek közvetlenül az *Inventory* tábla sémájára képezhetők le (természetesen a tulajdonságok típusa az oszlopok típusának megfelelő).



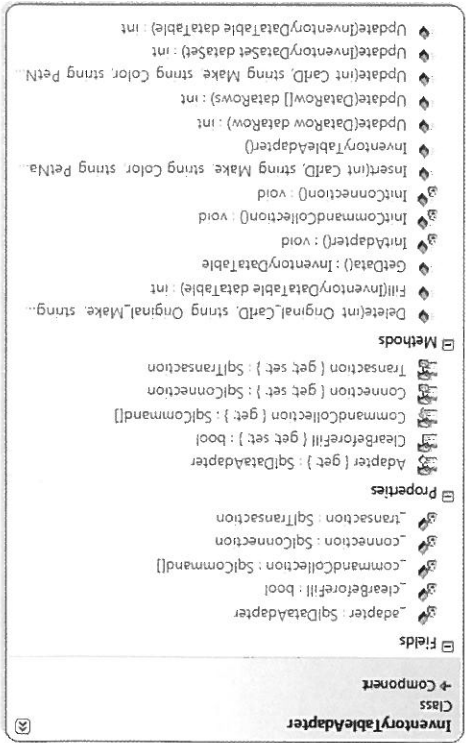
23.24. ábra: Az egyedi DataTable típus



23.25. ábra: Az egyedi DataRow típus

A generált adatillesztő

Az erős típusok meglete a kapcsolat nélküli típusok számára a Data Source Configuration Wizard használatainak komoly előnye, ugyanis az erős típusos osztályok manuális hozzáadása meglehetősen hosszadalmas (bár kivitelezhető) lenne. A varázsló szerencsére egyedi adatillesztő objektumot generál, amely képes feltölteni és módosítani az InventoryDataSet és az InventoryDataTable osztálytípusokat (lásd a 23.26. ábrát).



23.26. ábra: Egyedi adatillesztő, amely erősen típusos típusokkal működik

Az automatikusan generált InventoryTableAdapter típus karbantartja az SqlCommand objektumok gyűjteményét, amelyek mindegyike rendelkezik az SqlParameter objektumok teljesen feltöltött készletével (ez jelentős időmegtakarítást eredményez). Emellett ez az egyedi adatillesztő tulajdonságkészletet biztosít az alapul szolgáló kapcsolat-, tranzakció- és adatillesztő objektumok kinyerésére, valamint egy tulajdonságot az egyes parancstípusokat képviselő tömb megszerzésére. Ennek nyilvánvaló előnye az, hogy nem nekünk kell megírni a kódot.

Generált típusok használata a kódban

Ha megvizsgáljuk az űrlapból származtatott típus Load eseménykezelőjét, látható, hogy az egyedi adattípus `F111()` metódusa indításakor meghívódik, és átadja az egyedi dataset által fenntartott egyedi datatábla típusát:

```
private void MainForm_Load(object sender, EventArgs e)
{
    this.InventoryTabAdapter.F111(this.Inventory);
}
```

Ugyanezt az egyedi adattípus objektumot használhatjuk arra, hogy frissítsük a táblázat módosításait. Adjunk az űrlap felhasználói felületéhez egy `Button` vezérlőelemet (`btnUpdateInventory`), készítsünk eseménykezelőt a `Click` eseményhez, és írjuk bele a következő kódot:

```
private void btnUpdateInventory_Click(object sender, EventArgs e)
{
```

```
    // Ez visszaadja az Inventory táblában végrehajtott összes
    // módosítást feldolgozásra az adatbázisnak.
    this.InventoryTabAdapter.Update(
```

```
        this.InventoryDataSet.Inventory);
```

```
    // Megkapjuk a táblázat friss másolatát.
```

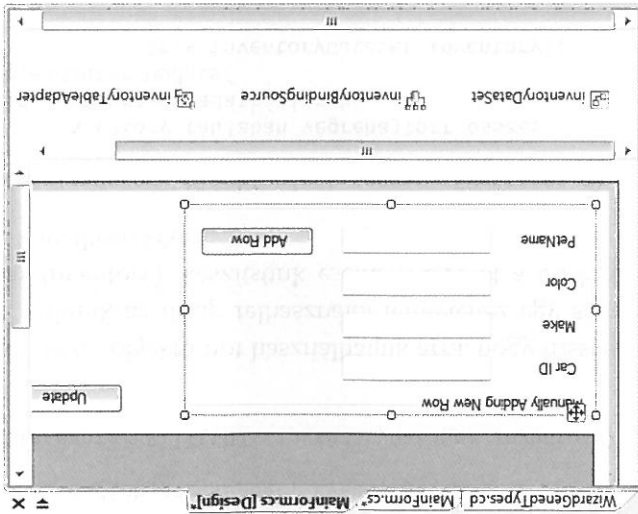
```
    this.InventoryTabAdapter.F111(this.InventoryDataSet.Inventory);
}
```

Futassuk megint az alkalmazást; adjunk hozzá, töröljünk vagy módosítsunk rekordot a táblázatban; végül kattintsunk az `Update` gombra. Ha megint futtatjuk a programot, láthatjuk, hogy a változások megjelentek. Az összes erősen típusos osztályt közvetlenül felhasználhatjuk a kódukbán (nagyjából) ugyanúgy, ahogy azt már megtettük a korábbiakban. Tételünk fel például, hogy olyan felhasználói felület-tartományal frissítettük az űrlapunkat (lásd a 23.27. ábrát), amely lehetővé teszi a felhasználó számára, hogy új rekord beírását szövegmezők segítségével (ez voltaképpen felesleges ebben a példában, ugyanis ezt a `DataGridView` megteszi helyettünk). Az új `Button` vezérlőelem `Click` eseménykezelőjébe beírhatjuk az alábbi kódot:

```
private void btnAddRow_Click(object sender, EventArgs e)
{
    // Lekérdezzük az adatokat a szövegdobozokból
    int id = int.Parse(txtCarID.Text);
    string make = txtMake.Text;
    string color = txtColor.Text;
    string petName = txtPetName.Text;
```

```
// Egyedi ílllesztő használataival beszűnünk egy sort.
inventoryTableAdapter.Insert(id, make, color, petName);

// Újra feltöltjük a táblázatot.
this.inventoryTableAdapter.Fill(this.inventoryDataSet.inventory);
}
```



23.27. ábra: Egyszerű frissítés az újlap típusához

Manuálisan is hozzáadhatunk egy új sort:

```
private void btnAddRow_Click(object sender, EventArgs e)
{
    // új sor beolvasása.
    InventoryDataSet.InventoryRow newRow =
        inventoryDataSet.Inventory.NewInventoryRow();
    newRow.CarID = int.Parse(txtCarID.Text);
    newRow.Make = txtMake.Text;
    newRow.Color = txtColor.Text;
    newRow.PetName = txtPetName.Text;
    inventoryDataSet.Inventory.AddInventoryRow(newRow);

    // Egyedi ílllesztő használataival beszűnünk egy sort.
    inventoryTableAdapter.Update(this.inventoryDataSet.inventory);

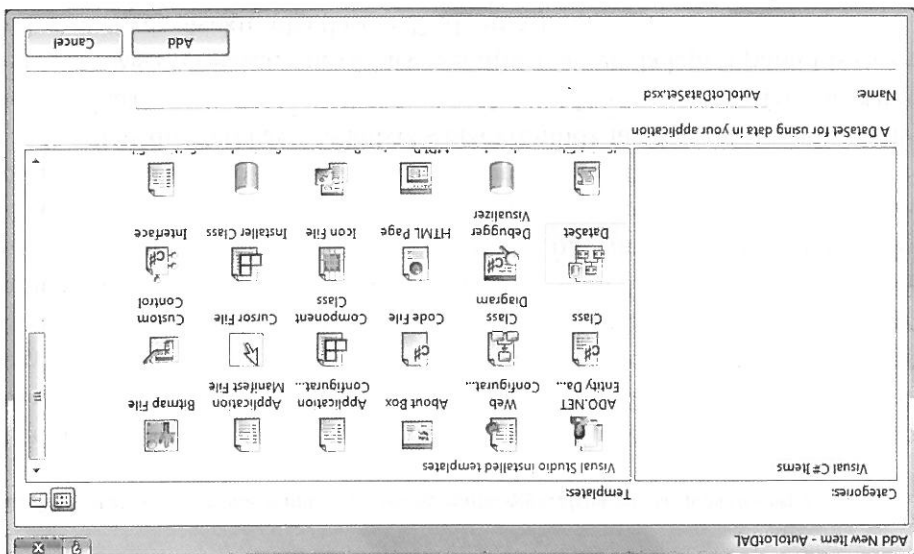
    // Újra feltöltjük a táblázatot.
    this.inventoryTableAdapter.Fill(this.inventoryDataSet.inventory);
}
```

Forráskód A VisualDataGridViewApp kódfájlokat a forráskódkönyvtár 23. fejezetének al-könyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Az automatikusan generált kód elválasztása a felhasználófelület-rétegről

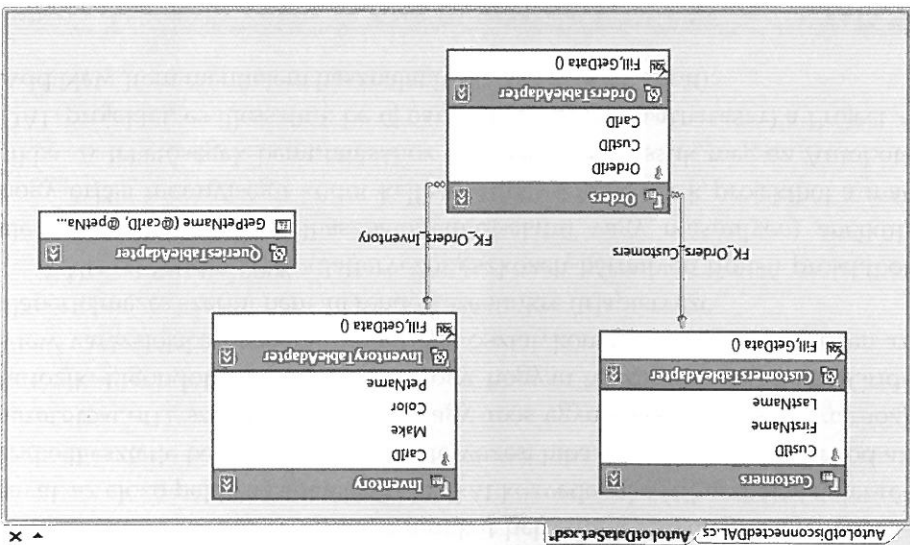
Végül mutassunk rá, hogy míg a DataGridView által elindított Data Source Configuration Wizard elvégezte a munkát helyettünk rengeteg kód létrehozásával, az előző példa az adatelérési logikát közvetlenül a felhasználófelület-rétegbe illesztette be – ez pedig súlyos tervezési hiba. Ideális esetben ez a kód az AutolotDAL.dll szerverelvényünkhöz (vagy más egyéb adatelérési könyvtárhoz) tartozik. Elgondolkozhatunk azon, hogy hogyan használhatjuk a DataGridView varázslójában generált kódot egy osztálykönyvtár-projektben, hiszen az alapértelmezés szerint nem áll rendelkezésünkre újraprojekt.

A Visual Studio 2008 adattervezői eszközöket bármilyen típusú projektből elérhetjük (legyen az felhasználófelület-alapú vagy másmilyen) anélkül, hogy őrjátsi mennyiségű kódot kellene átmásolnunk egyik projektből a másikba. A lehetőségek bemutatásához még egyszer nyissuk meg az AutolotDAL projektet, és illesszünk be új dataset típust (autolotdataset) a Project > Add New Item menüelem használatával (lásd a 23.28. ábrát).



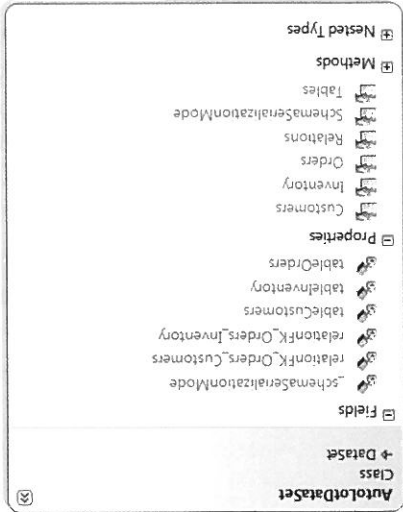
23.28. ábra: Új DataSet beszúrása

Ekkor megnyílik egy üres Dataset Designer felület. A Server Explorer segítségével kapcsolódjunk egy adott adatbázishoz (elvilleg már kapcsolódunk az AutoLothoz), és húzzunk át minden adatbázis-objektumot (most ne tördeljünk a CreditRisk táblával), amelyet a felületen szerezniünk generálni. A 23.29. ábrán láthatjuk az AutoLot minden olyan egyedi aspektusát, amelyről most gondoskodunk.

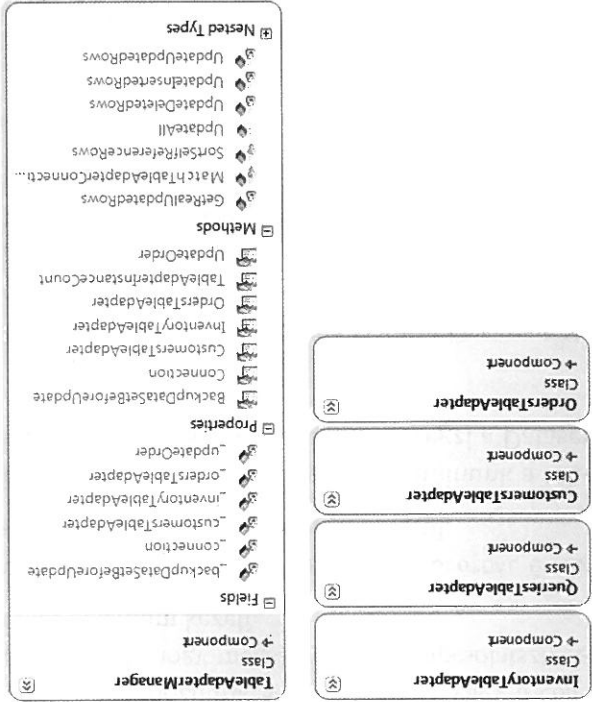


23.29. ábra: Az egyedi erősen típusos típusaink, ezáltal egy osztálykönyvtár-projekten belül

Ha rápillantunk a létrehozott kódra, erősen típusos dataset, dataable és datarow típusok egy új köteget láthatjuk, valamint minden egyes táblához egy egyedi adatillesztő objektumot. Mivel az autolotdataset típus tartalmaz olyan kódot, amely feltölti és frissíti az Autolot adatbázis minden tábláját, az automatikusan generált kód mennyisége több mint 3000 sor. Ennek nagy része azonban csak infrastruktúra, amelyet figyelmen kívül hagyhatunk. Ahogy a 23.30. ábrán látható, az autolotdataset típus az előző inventorydataset típus létrehozásához nagyon hasonlóan készült. Emellett minden egyes adatbázis-objektumhoz található egyedi adatillesztő objektumot, amelyet áthúztunk a Dataset Designer felületére, valamint egy TableAdapterManager nevű hasznos típust, amely belépési pontot biztosít minden egyes objektumba (lásd a 23.31. ábrát).



23.30. ábra: Az AutoloDataset



23.31. ábra: Az automatikusan generált adattílesztó objektumok

Forráskód Az AutoloDAL (Part 3) kódrájlokát a forráskódkönyvtár 23. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtáról lásd a Bevezetés xlv. oldalát.

Egy felhasználói felület front end: újra a MultitabledDataSetApp

Az automatikusan generált típusok használata megkezdődött, ha rutinosan dolgozunk a kapcsolat nélküli modellel. A szöveghez letölthető forráskód tartalmaz egy MultitabledDataSetApp-Redux projektet, amely – ahogy a neve is mutatja – a fejezet során korábban létrehozott MultitabledDataSetApp projekt egyik verziója.

Az eredeti példa gyengén típusos datasejt és sqldataadapter típusokat használ, hogy adatokat mozgasson a táblázatok között. Ez a verzió az autolotdal.d11 hármadik iterációját és a varázsló által generált típusokat használja. Bár most nem foglalkozunk azzal, hogy megnevezzük a teljes kódot (mivel többé-kevésbé ugyanaz, mint a projekt első iterációja), néhány fontos megjegyzést azonban érdemes tenni:

- Többé nem kell manuálisan létrehozni a config fájl vagy a configurationManagert használni, hogy megszerezzük a kapcsolatszttinget, ugyanis ezt a settings objektum kezeli.
- Most már erősen típusos osztályokat használunk az autolotdal és az autolotdal.AutolotDataSetTabLeadapters névtérben belül.
- Többé nem kell manuálisan létrehozni a konfigurálunk a tábláink közötti kapcsolatot, ezt ugyanis automatikusan elvégzi a DataSet Designer.

Az utolsó ponttal kapcsolatban figyelni kell arra, hogy a DataSet Designer által a táblakapcsolatoknak adott neveket *különböznek* azokról, amelyeket mi adunk a projekt első iterációjában. Ezért a btngeneratorInfo_Click() metódust módosítani kell, hogy a helyes kapcsolata neveket (amelyeket a DataSet Designer tervezői felületen láthatunk) használja, például:

```
private void btngeneratorInfo_Click(object sender,
    System.EventArgs e)
{
    // Frissíteni kell a kapcsolatot nevet.
    drsorder =
    drscust[0].GetChildRows(
    autolotds.Relations["FKOrders_Customers"]);
    ...
}
```

```
// Frissíteni kell a kapcsolat nevét.  
DataRow[] drsInV =  
    drsOrder[0].getParentRows(  
        autolotDS.Relations["FK_Orders_Inventory"]);  
...  
}
```

Forráskód A MultitabbedDataSetApp-Redux kódfájlokat a forráskódkönyvtár 23. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Összefoglalás

Ebben a fejezetben az ADO.NET kapcsolat nélküli modelljének részleteivel foglalkoztunk, amelynek központi eleme a dataset. Ez a típus tetszőleges számú tábla, valamint tetszőleges számú opcionális kapcsolat, korlátozás és kifejezés memóriabeli megjelenítésére szolgál. A helyi táblák közötti kapcsolatok kiépítésének előnye az, hogy programozottan navigálhatunk közöttük, miközben azok nem kapcsolódnak a távoli adattárolóhoz.

A fejezetben megvizsgáltuk az adatillesztő típus szerepét. Ennek a típusnak (és a kapcsolódó selectcommand, insertcommand, updatecommand és deletecommand tulajdonságoknak) a használataival az illesztő végrehajthatja a dataset módosításait az eredeti adattárban is. Emellett megtanultuk, hogyan navigáljuk egy dataset objektummodelljében manuálisan, valamint erősen típusos objektumok révén, amelyeket főként a Visual Studio 2008 Dataset Designer eszközeivel generáltunk.

A LINQ API programozása

Ebben a fejezetben elkezdjük a *LINQ to ADO.NET* vizsgálatát. Ezt a kifejezést a LINQ programozási modellhez kapcsolódó két jellemző leírására használjuk, a LINQ to DataSet és a LINQ to SQL lehetőségekhez. Ezek az API-k teszik lehetővé a LINQ-lekérdezések végrehajtását relációs adatbázisokban, illetve ADO.NET dataset objektumokon. Ezután a LINQ to XML szerepét tekintjük át. A LINQ ezen aspektusa nemcsak azt teszi lehetővé, hogy a lekérdezésope-rátorok készenlethez használataival adatokat nyerjünk ki XML-dokumentum-ból, hanem azt is, hogy lényegre törően (sokkal hatékonyabban, mintha a `system.xml.dll` szerelvénybe csomagolt típusokat használják) töltsünk be, mentssünk és hozzunk létre XML-dokumentumokat.

Megjegyzés Ez a fejezet az első kötet 14. fejezetében leírt LINQ programozási modellben való jártasságot feltételezi.

A LINQ to ADO.NET szerepe

A LINQ programozási modell (lásd az előző kötet 14. fejezetét is) lehetővé teszi a programozók számára, hogy olyan erősen típusos lekérdezéskifejezéseket hoz-zanak létre, amelyeket az adattárolók széles körére lehet alkalmazni (tömbök, gyűjtemények, adatbázisok, XML-dokumentumok). Függelendül attól, hogy mi-lyen objektumon hajtjuk végre a LINQ-lekérdezést, mindig lehet ugyanazokat a lekérdezésope-rátorokat használni, és a LINQ to ADO.NET API néhány további típust és infrastruktúrát is biztosít a LINQ-adatbázis integrációjához.

A LINQ to ADO.NET olyan gyűjtőfogalom, amely a LINQ két adatbázis-központú szempontját írja le. Az egyik a LINQ to DataSet. Ez az API lényegé-ben bővítménykészlet a szabványos ADO.NET DataSet programozási model-lhez, amely lehetővé teszi, hogy a DataSet, a DataTable és a DataRow típusokon is lehessen LINQ-lekérdezéskifejezéseket végrehajtani. A `system.core.dll` típusa-inak használatán túl a LINQ to DataSet megköveteli, hogy a projekt használja a `system.data.dataextensions.dll` szerelvény kiegészítőtypusait.

A LINQ to ADO.NET másik komponense a LINQ to SQL. Ez az API lehetővé teszi, hogy relációs adatbázisokkal működjünk együtt, mert elvonatkoztat az alapul szolgáló ADO.NET-adattípusoktól (kapcsolatok, parancsok, adatillesztők stb.) az *entitásosztályok* használatára révén. Ezekben az entitásosztályokon keresztül ábrázolhatunk relációs adatokat egy intuitív objektummodellel, valamint manipulálhatjuk az adatokat a LINQ-lekérdezések segítségével. A LINQ to SQL funkcionálitás a `System.Data.Linq.dll` szerezélyben található.

Megjegyzés A .NET 3.5 óta a LINQ to SQL nem támogatja az `data provider factory` modellel (lásd a 22. fejezetet). Ezért amikor ezt az API-t használjuk, az adatoknak a Microsoft SQL Server (vagy más) adatbázisból származniuk kell. Ezzel szemben a LINQ to DataSet API természetesen „szabadadosabb”, ugyanis a kezelt dataset bármilyen relációs adatbázisból származhat.

Programozás a LINQ to DataSettel

Az előző fejezetben láthattuk, hogy a dataset típus a kapcsolat nélküli modell központi eleme, és arra használható, hogy jelképezze a kapcsolódó data-table objektumok gyorstítozárát másolatát, valamint (esetlegesen) a közöttük lévő kapcsolatokat. Ehhez kapcsolódó megjegyzés, hogy egy dataset adatait három különböző módon lehet manipulálni:

- indexelőkkel,
- adattáblázat-olvasókkal,
- erősen típusos adattagokkal.

Amikor a dataset és a data-table típusok különböző indexelést használnak, akkor meg lehetetlen egyértelműen, de nagyon gyakran típizáltan dolgozni a tárolt adatokkal. Ez a megközelítés azt követeli meg, hogy az adatokat cellák táblázatos blokkjaként kezeljük. Például:

```
static void PrintDataWithIndexes(DataTable dt)
{
    // Kérjük a DataAdapter-t.
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
        for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
        {
            Console.WriteLine(dt.Rows[curRow][curCol].ToString() + " \\t");
        }
    }
}
```



```

        Console.WriteLine();
    }
}

```

A `DataTable` típus `Createdatareader()` metódusa egy olyan másik megközelítést kínál, amely szerint a dataset adatait úgy kezelhetjük, mint sorok lineáris készletének szekvenciális feldolgozását:

```

static void PrintDatawithDatareader(DataTable dt)
{
    // Beolvassuk a DataTableleADER típust.
    DataAdapter datareader = dt.CreateDatareader();
    while (datareader.Read())
    {
        for (int i = 0; i < datareader.FieldCount; i++)
        {
            Console.WriteLine("{0}\t", datareader.GetValue(i));
        }
        Console.WriteLine();
    }
    datareader.Close();
}

```

Végül az erősen típusos dataset lehetővé teszi, hogy az objektumban lévő adatokat olyan tulajdonságok használatával kezeljük, amelyek lekepezhetők a relációs adatbázis tényleges oszlopneveire. Erősen típusos objektumok használatával írhatunk olyan kódokat, mint például a következő (lásd a 23. fejezet):

```

static void AddresswithTypeddataset()
{
    InventoryTableAdapter invDA = new InventoryTableAdapter();
    AutoLoadDataSet inventoryDataTable = invDA.GetData();
    invDA.AddInventoryRow(999, "Ford", "Yellow", "Sail");
    invDA.Update(inv);
}

```

Ezen túlmenően a LINQ to DataSet még egy lehetőséget biztosít a tárolt adatok kezelésére a LINQ-lekérdezőkifejezések használatával. Az ADO.NET DataSetbe (és az olyan kapcsolódó típusokba, mint a `DataRow` és `DataRowView`) alapvetően nincsen integrálva a szükséges infrastruktúra ahhoz, hogy közvetlenül végrehajtsunk rajta egy LINQ-lekérdezőt. A következő metódus például fordítási idejű hibát eredményezne:

```

static void LinqOverDataTable()
{
    // Adatok DataTable típusának beolvasása.
    InventoryDALDataLayer dal = new InventoryDALDataLayer(
        @"Data Source=(local)\SQLEXPRESS;" +
        "Initial Catalog=AutoloT;Integrated Security=True");
    DataTable data = dal.GetAllInventory();

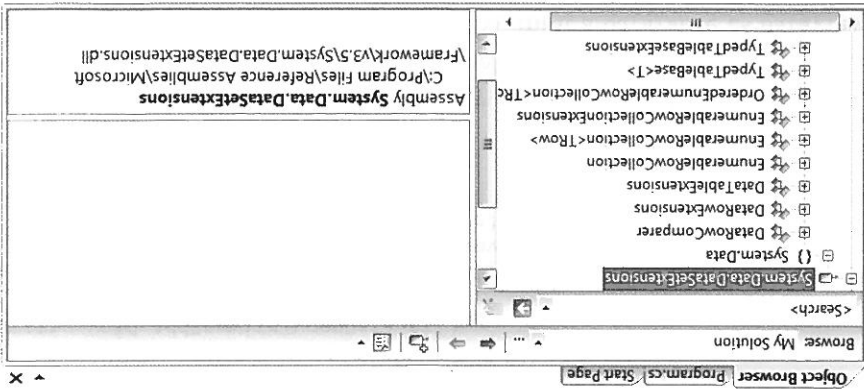
    // Az 5-nél nagyobb CarID-vel rendelkező típusok beolvasása?
    var moreData = from c in data where (int)c["CarID"] > 5 select c;
}

```

Ha lefordítanánk a `LinqOverDataTable()` metódust, akkor a fordító arról tájékoztatna, hogy a `DataTable` típus biztosít egy „lekérdezésiminta-implementációt”. Hasonlóan az `IEnumerable<T>` típust (pl. az `ArrayList`-et) nem implementáló objektumokra vonatkozó LINQ-lekérdezések alkalmazásához, az ADO.NET-objektumokat pedig kompatibilis típusúak kell alakítani. Hogy megértjük ennek mikéntjét, meg kell vizsgálnunk a `System.Data.DataTable`-típusait.

A DataSet bővítmények szerepe

A `System.Data.DataSetExtensions.dll` több új taggal bővíti a `System.Data` névteret (lásd a 24.1. ábrát).



24.1. ábra: A `System.Data.DataSetExtensions.dll` szerelvény

A két leghasznosabb tag vitathatatlanul a `DataTableExtensions` és a `DataRowExtensions`. Ahogy a nevük is mutatja, ezek a típusok a `DataTable` és a `DataRow` típusok működését bővítő metódusok készletével egészítik ki. Egy másik kulcsfontosságú típus a `TypedTableBase`-típus a `TypedTableBaseExtensions`, amely olyan bővítő metó-

duásokat definiál, amelyeket erősen típusos dataset objektumokra lehet alkalmazni azért, hogy a belső datatable objektumok LINQ-kompatibilitásuk legyenek. A `System.Data.DataSetExtensions`.`d11` szerelvény összes többi tagja pusztán az infrastruktúrát szolgálja ki, és nem közvetlen használatra szánják őket.

DataTable LINQ-kompatibilis használata

A dataset bővítmények használatának bemutatásához tételizzük fel, hogy van egy új C# parancssori konzolalkalmazásunk `LinqOverDataSet` névvel. Fontos megjegyezni, hogy amikor a .NET 3.5-öt célzó projekteket készítettünk, akkor automatikusan referenciát kapunk a `System.Core`.`d11` és `System.Data.DataSetExtensions`.`d11` szerelvényekre; ám ebben a példában adjunk hozzá még egy szerelvényreferenciát az `AutoloDAL`.`d11` szerelvényre (lásd a 23. fejezetet), és egészítsük ki a kezdeti kóddal a következő logikával:

```
using System.Data;
using AutoloDAL;

namespace LinqOverDataSet
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** LINQ over DataSet *****\n");
            // Beolvassuk az Autolo adatbázis aktuális Inventory
            // táblájának datatable típusát.
            InventoryDALDataLayer dal = new InventoryDALDataLayer(
                @"data source=(local)\SQLExpress;Initial
                Catalog=Autolo;"+ "Integrated Security=True");
            datatable data = dal.GetAllInventory();

            // Hívjuk meg az itt következő metódusokat.
            Console.ReadLine();
        }
    }
}
```

Amikor LINQ-kompatibilis objektumná szeretnénk átalakítani egy `ADO.NET datatable` típust, akkor egyszerűen hívjuk meg az `AsEnumerable()` bővítő metódust, amelyet a `datatableextensions` típus definiál. Ez visszaad egy `enumerablecollection` objektumot, amely egy `DataRow` gyűjteményt tartalmaz. Az `EnumerableCollection` típus használatával az elvártak szerint dolgozhatunk az egyes sorokkal. Nézzünk egy egyszerű példát:

```

static void PrintAllCars(data)
{
    // Beolvassuk a datatáblát felosztható változatát.
    EnumerableViewCollection<T> enumData = data.AsEnumerable();

    // Kijelöljük az azonosító értékeit.
    foreach (DataRow r in enumData)
        Console.WriteLine("Car ID = {0}", r["CarID"]);
}

```

Mivel az `EnumerableViewCollection` megvalósítja az `IEnumerable<T>` típust, ezért az is megengedett, hogy az alábbi utasítások egyikével kapjuk meg a visszatérési értéket:

```

// A visszatérési érték tárolása IEnumerable<T> típusként.
IEnumerable<DataRow> enumData = data.AsEnumerable();

// A visszatérési értékek tárolása implicit módon.
var enumData = data.AsEnumerable();

```

Eddig ugyan még nem futtattuk le a LINQ-lekérdezést, ám az `enumData` objektumon immár végrehajthatunk LINQ-lekérdezéskifejezést. Az `EnumerableCollectionView` valóban `DataRow` objektumok gyűjteményét tartalmazza, ezért egy típusindexelőt alkalmazunk az összes alobjektumra, hogy kiirassuk a `CarID` oszlop értékét.

A legtöbb esetben nem kell `EnumerableCollectionView` típusú változót deklarálnunk azért, hogy tárolja az `AsEnumerable()` visszatérési értéket. Ehelyett hívjuk meg ezt a metódust magából a lekérdezéskifejezésből. Akad egy még érdekesebb metódus is, amely visszaadja a `CarID`/makes oszlopok leképezését a datatábla összes bejegyzéséből, ahol a `CarID` értéke nagyobb mint 5:

```

static void ApplyLINQquery(data)
{
    // Képezzünk le egy új eredményhalmazt, amely tartalmazza
    // az ID/color értékeket az 5-nél nagyobb CarID-vel rendelkező
    // sorok esetében.
    var cars = from car in data.AsEnumerable()
    where
        (int)car["CarID"] > 5
    select new
    {
        ID = (int)car["CarID"],
        color = (string)car["color"]
    };
}

```

```

Console.WriteLine("Cars with ID greater than 5:");
foreach (var item in cars)
{
    Console.WriteLine("<!--> CarID = {0} is {1}", item.ID, item.Color);
}
}

```

A DataRowExtensions.Field<T>() bővítő metódus szerepe

A jelenlegi LINQ-lekérdezéskifejezésben hátrányos lehet az, hogy számos kasztolási műveletet és datarow indexelést használunk az eredményhalmaz összeállításához, és ez futásidőjelel kivétel eredményezhet, ha nem kompatibilis adatátípusra próbálunk kasztolni. Ahhoz, hogy erős típizálást alkalmazzunk a lekérdezésünkben, használhatjuk a datarow típus `Field<T>()` bővítő metódusát. Így megnöveljük a lekérdezés típusbiztonságát, ugyanis az adat-típusok kompatibilitását a rendszer a fordítási időben ellenőrzi. Nézzük meg a következőképpen módosított kódot:

```

var cars = from car in data.AsEnumerable()
            where
                car.Field<int>("CarID") > 5
            select new
            {
                ID = car.Field<int>("CarID"),
                Color = car.Field<string>("Color")
            };

```

Látható, hogy ebben az esetben meghívhatjuk a `Field<T>()` metódust, és megadhatunk egy típusparamétert, amely az oszlop alapípusát határozza meg. A metódus argumentumaként átadjuk magát az oszlopnevet. A fordítási idejű ellenőrzés miatt tanácsos a `Field<T>()` metódust használni a datarow indexelője helyett `Enumerable.Cast<T>()` metódust használni. Azon túl, hogy meghívjuk az `AsEnumerable()` metódust, a LINQ-lekérdezés általános formátuma hasonló ahhoz, amelyet már láttunk az előző kötet 14. fejezetében. Ezért most nem térünk ki újra a különböző LINQ-operátorok részleteire. Ha további példákra vagyunk kíváncsiak, nézzük át a „LINQ to DataSet Examples” témakört a .NET Framework 3.5 SDK dokumentációjában.

Új DataTable objektumok feltöltése a LINQ-lekérdezésekből

Könnyedén fel lehet tölteni egy új DataTable adatait egy LINQ-lekérdezés alapján, feltéve, hogy *nem* használunk lekérdezéseket. Amikor van egy olyan eredményhalmazunk, amelyben az alapul szolgáló típus `IEnumerable<T>` típusként lehet ábrázolni, meghívhatjuk a `CopyToDataTable<T>()` bővítő metódust az eredményen. Például:

```
static void BuildDataTableFromQuery(DataTable data)
{
    var cars = from car in data.AsEnumerable()
               where
                   car.Field<int>("CarID") > 5
               select car;

    // Használjuk ezt az eredményhalmazt új DataTable létrehozásához.
    DataTable newtable = cars.CopyToDataTable();

    // Kírtja a DataTable-t.
    for (int currow = 0; currow < newtable.Rows.Count; currow++)
    {
        for (int curcol = 0; curcol < newtable.Columns.Count; curcol++)
        {
            Console.WriteLine(
                newtable.Rows[currow][curcol].ToString().Trim() + "\t");
        }
        Console.WriteLine();
    }
}
```

Megjegyzés Az `AsDataView<T>()` bővítő metódussal is átalakíthatunk egy LINQ-lekérdezést `DataView` típusúvá.

Ez a megközelítés akkor lehet nagyon hasznos, ha a LINQ-lekérdezés eredményét adatkötés során forrásként szeretnénk használni. A Windows Forms `DataGridView` (valamint az `ASP.NET GridView`) vezérlőelem például támogatja a `DataSource` tulajdonságot. LINQ-eredményt az alábbiak szerint köthetünk egy `DataGridView`-hez:

```
// Tegyük fel, hogy a mydatagrid GUI-alapú táblázatobjektum.
mydatagrid.DataSource = (from car in data.AsEnumerable()
                          where
                              car.Field<int>("CarID") > 5
                          select car).CopyToDataTable();
```

A következőkben térjünk át a LINQ to SQL-re.

Programozás a LINQ to SQL használatával

A LINQ to SQL olyan API, amely lehetővé teszi jól megformázott LINQ-lekérdezések kiírását a LINQ to SQL használatával. A LINQ to SQL több olyan típust kínál (a `system.data.linq.dll` szerezvényben), amelyek megkönnyítik a kommunikációt a kód és a fizikai adatbázis-motor között.

A LINQ to SQL legfőbb célja az, hogy konzisztenciát biztosítson a relációs adatbázisok és a velük folytatott munkára használt programozási logika számára. Ahelyett például, hogy az adatbázis-lekérdezéseket nagy sztringekkel jellepezzünk, használhatunk erősen típusos LINQ-lekérdezéseket. Továbbá ahelyett, hogy a relációs adatokat rekordok formájában kellene kezelnünk, dolgozhatunk az adatokkal szabványos objektumorientált programozási módszerekkel. Mivel a LINQ to SQL lehetővé teszi az adatelérés integrálását közvetlenül a C#-forráskódban, nagymértékben csökkenti annak a szükségességet, hogy manuálisan hozzuk létre egyedi osztályok tucait, valamint azokat az adatelérési könyvtárakat, amelyek elrejtik szem elől az ADO.NET csúnya kódjait.

Amikor LINQ to SQL felhasználásával programozunk, nyoma sincs az olyan megszokott ADO.NET-típusoknak, mint az `sqlconnection`, az `sqlcommand` vagy az `sqlDataAdapter`. A LINQ-lekérdezések kiírásai, a (harmarosan ismeretendő) entitáshoz tartozó típusok és a `DataContext` típus segítségével végrehajthatjuk a kívánt adatbázis-műveleteket (letréhozás, eltávolítás, módosítás és törlés), valamint definiálhatunk tranzakciós kontextusokat, létrehozhatunk új adatbázis-entitásokat (vagy egész adatbázisokat), meghatározhatunk tárolt eljárásokat, és elvégezhetünk egyéb adatbázis-centrikus tevékenységeket. Ezenfelül a LINQ to SQL típusok (példáként megint a `DataContext`-et említhetjük) feladata, hogy megvalósítsák a standard ADO.NET-adattípusok integrációját. A `DataContext` egyik túllerhelt konstruktora beemeltként például egy `IdbConnection` objektumot vár, amely az összes ADO.NET-kapcsolatobjektum által támogatott közös interfész. Ilyen módon a már meglévő ADO.NET adatelérési könyvtárak integrálhatók a C# 2008 LINQ-lekérdezésekbe (és fordítva). Valójában – ami a Microsoftot illeti – a LINQ to SQL egyszerűen az ADO.NET-család egyik új tagja.

Az entitásosztályok szerepe

Ha a LINQ to SQL-t szeretnénk használni az alkalmazásainkban, akkor az első lépés az *entitásosztályok* definiálása. Röviden: az entitásosztályok olyan típusok, amelyek azokat a relációs adatokat jelleképezik, amelyekkel dolgozni szeretnénk. Programozási szempontból az entitásosztályok olyan osztálydefiniciók, amelyeket különböző LINQ to SQL attribútumokkal (például [Table] és [Column]) jelölünk, amelyek leképezhetők fizikai táblákra egy adott adatbázisban. A LINQ to SQL attribútumok többségének definícióját a `system.data.Linq.Mapping` névtér foglalja magában (lásd a 24.2. ábrát).

Ahogy azt hamarosan látni fogjuk, a .NET Framework 3.5 SDK (valamint a Visual Studio 2008) olyan eszközöket biztosít, amelyek automatizálják az alkalmazás számára szükséges entitás típusok elkészítését. Ezt megelőzően az első LINQ to SQL példánk fogja bemutatni, hogy hogyan hozzunk létre entitásosztályokat.



24.2. ábra: A `System.Linq.Mapping` névtér számos LINQ to SQL attribútumot definiál

A DataContext típus szerepe

Ha meghatároztuk az entitásosztályokat, akkor a `DataContext` típusalattal tudjuk lekérdezési kifejezéseket a relációs adatbázisnak. Ez a LINQ to SQL-specifikus osztálytípus felelős a LINQ-lekérdezéskifejezések megfelelő SQL-lekérde-

zésekre történő fordításáért, valamint az adott adatbázissal történő kommunikációért. Bizonyos tekintetben a DataContext hasonló egy ADO.NET-kapcsolatobjektumra: szükség van egy kapcsolatsztringre. A típusos kapcsolatobjektummal ellentétben azonban a DataContext típus számos olyan taggal rendelkezik, amelyek a lekérdezéskifejezések eredményeit az általunk meghatározott entitásosztályokba képezik le.

Továbbá a DataContext típus egy factoryminit definiál a kódban használt entitásosztályok példányainak megszerzésére. Ha megszerezzünk egy entitáspéldányt, akkor a kívánt módon változtathatjuk meg annak állapotát (rektorok hozzáadása és frissítése stb.), és a módosított objektumot feldolgozásra visszaküldhetjük. Így módon a DataContext hasonló az ADO.NET adatillesztő típushoz.

Egy egyszerű LINQ to SQL példa

Először nézzük meg a LINQ to SQL használatának egy egyszerű példáját, amely a 22. fejezetben létrehozott AutoLot adatbázis Inventory táblájával dolgozik. Ebben a példában nem használjuk az AutoLotDAL.d11 könyvtárat, hanem minden kódot manuálisan készítünk el. Hozzunk létre egy új páranccsori konzolalkalmazást SimpleLINQtoSQLApp névvel, valamint adjunk egy referenciát a System.Data.Linq.d11 szerezvényre.

Ezután adjunk hozzá egy új C#-osztályfájlt (Inventory.cs). Ez a fájl fogja definiálni az entitásosztályunkat, amely megköveteli a típus kiegészítését különböző LINQ-központú attribútumokkal; ezért mindenképpen adjuk meg, hogy a System.Data.Linq.Mapping és System.Data.Linq névtérket használjuk. A következőkben nézzük meg az Inventory típus definícióját:

```
[Table]
public class Inventory
{
    [Column]
    public string Make;
    [Column]
    public string Color;
    [Column]
    public string PetName;
    // Az elsődleges kulcs azonosítása.
    [Column(IsPrimaryKey = true)]
    public int CarID;
```

```

    public override string ToString()
    {
        return string.Format("ID = {0}; Make = {1}; Color = {2};
                               PetName = {3}", CarID, Make.Trim(),
                               Color.Trim(), PetName.Trim());
    }
}

```

Az entitásosztályunk megkapta a [Table] attribútumot, amíg a nyilvános mezők mindegyikét megjelöltük a [Column] attribútummal. A nevek mindkét esetben közvetlen leképezései a fizikai adatbázis tábláinak. Ez azonban nem szigorú követelmény, hiszen a tableattribute és a columnattribute típusok támogatják a Name tulajdonságot, amely lehetővé teszi az adattábla programozott reprezentációjának elkülönítését magától a fizikai táblától. A carID mezőt tovább minősítettük a columnattribute típus isprimarykey tulajdonságának beállításával, a megnevezett tulajdonság szintaxis segítségével.

Az egyszerűség kedvéért minden mezőt publikusan deklaráltunk. Ha erősebb egyébe zárásra van szükségünk, akkor definiálhatunk nyilvános tulajdonságokba (vagy akár automatikus tulajdonságokba) csomagolt privát mezőket. Ekkor nem a mezők, hanem a *tulajdonságok* lesznek a [Column] attribútumnal megjelölve.

Egy entitásosztály bármennyi olyan tagot tartalmazhat, amelyek nem képezhetők le a reprezentált adattáblára. Ami a LINQ-futásidőt illeti, a rendszer csak a LINQ to SQL attribútumokkal megjelölt elemeket használja az adatcsere során. Ez az inventory osztálydefinició például a toString() egyedi megvalósítását biztosítja ahhoz, hogy lehetővé tegye az alkalmazás számára az állapot gyors megjelentését.

Ha már rendelkezünk entitásosztállyal, használhatjuk a DataContext típust, hogy elküldjük (és lefordítsuk) a LINQ-lekérdezéskifejezéseinket a megadott adatbázisnak. Tanulmányozzuk az alábbi main() metódust, amely megjeleníti az AutoLot adatbázis Inventory táblájában lévő összes elemet:

```

class Program
{
    const string cnstr =
        @"Data Source=(local)\SQLEXPRESS;Initial Catalog=AutoLot;" +
        "Integrated Security=True";
    static void Main(string[] args)
    {
        Console.WriteLine("***** LINQ to SQL Sample App *****\n");
        // DataContext objektum létrehozása.
        DataContext db = new DataContext(cnstr);
    }
}

```