```
    // 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 eltárolhatjuk az alkalmazáskonfigurációs fájlban, és/vagy használhatjuk az `SqlConnectionStringBuilder` típust az objektumorientáltabb kezelés érdekében.

Ezután megszerezzük az `Inventory` entitásosztályunk egy példányát a `DataContext` típus generikus `GetTable<T>()` metódusának meghívásával, ahol az entitásosztályt adjuk meg típusparaméterként. Végezetül létrehozunk egy LINQ-lekérdezéskifejezést, és végrehajtjuk 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 létrehoznunk egy olyan osztályt, amely kibővíti a kezelt táblák mindegyike számára tagváltozókat definiáló `DataContext` típust. Illesszünk be egy új osztályt `AutoLotDatabase` névvel, adjuk meg, hogy a `System.Core` és a `System.Data.Linq` névtereket 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:

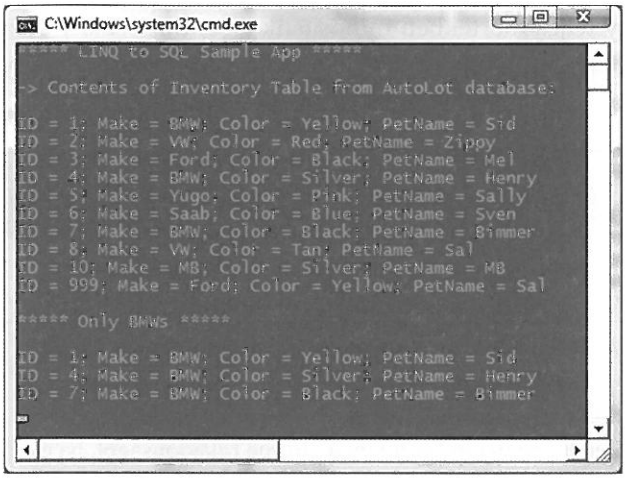
```
static void Main(string[] args)
{
  Console.WriteLine("***** LINQ to SQL Sample App *****\n");

  // Hozzunk létre egy AutoLotDatabase objektumot.
  AutoLotDatabase db = new AutoLotDatabase(cnStr);

  // 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());
  Console.ReadLine();
}
```

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

A LINQ-lekérdezések természetesen használhatók adott eredményhalmaz megszerzésére is. Tegyük fel, hogy létrehoztuk 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 SimpleLinqToSqlApp kódfájlokat 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 lekérdezését adatbázis SQL-lekérdezésekre. Csak a `[Table]` és a `[Column]` attribútumokat fogjuk használni; mindazonáltal további attribútumok is léteznek azoknak a metódusoknak a megjelölésére, amelyek végrehajtják az SQL beszúró, módosító és törlő parancsait. Ezenfelül a LINQ to SQL attribútumok mindegyike 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 `IsPrimaryKey` tulajdonságon túl a `ColumnAttribute` további olyan tagokat definiál, ame-

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

ColumnAttribute tulajdonság	Jelentés
`CanBeNull`	Ez a tulajdonság azt jelzi, hogy az oszlop tartalmazhat `null` értéket.
`DbType`	Az adatmezők deklarációja alapján a LINQ to SQL automatikusan kikövetkezteti, milyen tulajdonságokat kell az adatbázismotornak átadni. Emiatt általában csak akkor kell közvetlenül beállítani a `DbType` tulajdonságot, ha dinamikusan hozunk létre adatbázisokat a `DataContext` típus `CreateDatabase()` metódusának használatával.
`IsDbGenerated`	A tulajdonság azt állítja be, hogy az adatbázis automatikusan generálja egy mező értékét.
`IsVersion`	Ez a tulajdonság határozza meg, hogy az oszlop adatbázis-időbélyeg, esetleg verziószám-e, vagy sem. A verziószámok növekednek, az időbélyegoszlopok pedig frissülnek minden alkalommal, amikor a kapcsolódó sor módosul.
`UpdateCheck`	Ez a tulajdonság vezérli, hogy a LINQ to SQL hogyan kezelje az adatbázis-ütközéseket az optimista konkurenciaszabályozás 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 termékszintű LINQ to SQL alkalmazások több, egymással kapcsolatos adattáblával dolgoznak, s ezek mindegyike oszlopok tucatjait definiálhatja. Ilyen esetekben unalmas lenne manuálisan létrehozni minden egyes kívánt entitásosztályt. Szerencsére kétféle módon is megoldhatjuk ezeknek 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 entitásosztályok létrehozását, hogy generál egy megfelelő C#-osztálytípust az adatbázis metaadataiból. Bár az eszköz több parancssori paraméterrel rendelkezik, a 24.2. táblázat csak a lényegesebb kapcsolókat mutatja be.

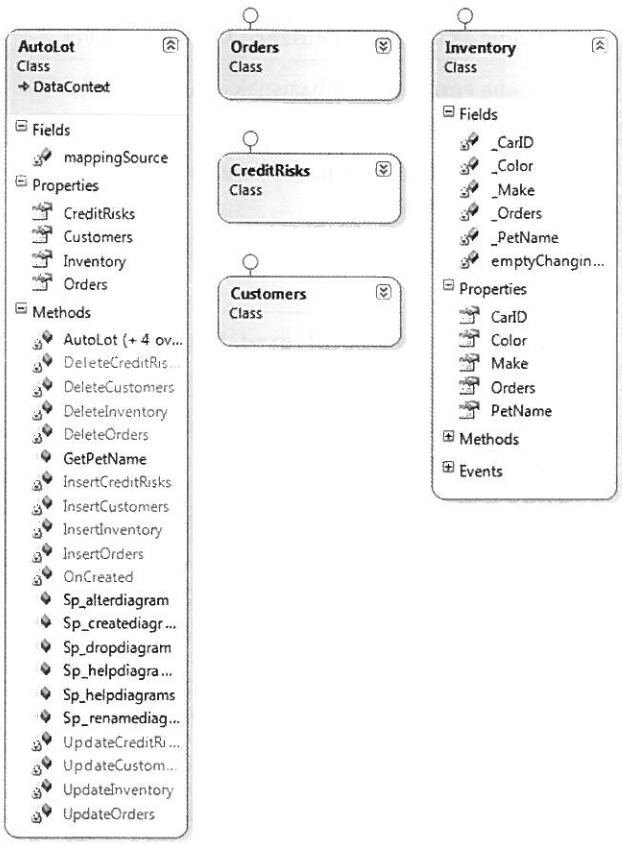
sqlmetal.exe parancssori paraméter	Jelentés
`/server`	Megadja az adatbázist hosztoló kiszolgáló nevét.
`/database`	Megadja annak az adatbázisnak a nevét, amelyből ki kell olvasni a metaadatokat.
`/user`	Megadja a felhasználónevet, amellyel be kell jelentkezni a kiszolgálóra.
`/password`	Megadja a jelszót, amellyel be kell jelentkezni a kiszolgálóra.
`/views`	Tájékoztatja az `sqlmetal.exe`-t, hogy létező adatbázisnézetek alapján generáljon kódot.
`/functions`	Tájékoztatja az `sqlmetal.exe`-t, hogy készítse el az adatbázisfüggvények megfelelőit.
`/sprocs`	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 paraméterei*

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

```
sqlmetal /server:(local)\SQLEXPRESS /database:AutoLot
         /namespace:AutoLotDatabase /code:autoLotDB.cs /sprocs
```

Ha végrehajtottuk az utasítást, hozzunk létre egy új parancssorialkalmazás-projektet `LinqWithSqlMetalGenedCode` névvel, illetőleg adjunk referenciát a `System.Data.Linq.dll` szerelvényre, és adjuk hozzá az `autoLotDB.cs` fájlt a projekthez 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 entitásosztályok*

Van tehát egy új, a `DataContextet` kibővítő típusunk, amely tartalmazza a tulajdonságokat a megadott adatbázisban lévő összes adattábla számára (továbbá a `GetPetName()` tárolt eljárást a megegyező névvel rendelkező nyilvános metódus képviseli). Mielőtt elkezdenénk 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, CreditRisks) számára külön entitásosztályt definiált, és minden oszlopot beágyazott egy tulajdonságba. Továbbá az egyes entitásosztályok két interfészt implementálnak (`INotifyPropertyChanging` és `INotifyPropertyChanged`), amelyek mindegyike egy-egy eseményt határoz meg:

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

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 (`PropertyChanging` és `PropertyChanged`), amelyek a `System.ComponentModel` névtérben meghatározott `PropertyChangedEventHandler` metódusreferenciával működnek együtt. Ez a metódusreferencia meghívhat 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 PropertyChanging;
  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 váltsák ki az eseményeket. Ezeket a tagokat azonban *részleges metódusokként* definiáltuk, amelyek könnyű eseménykezelést hajtanak végre (lásd az előző kötet 13. fejezetét), lehetővé téve az érdekelt hívó számára, hogy biztosítsa a szükséges megvalósítást (ha nem, akkor a fordítás során a rendszer eltávolítja őket a típusdefinícióból):

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

Kapcsolatok definiálása entitásosztályok használatával

Azon túl, hogy tulajdonságokat és segédmezőket definiál az adattábla oszlopainak megjelenítésére, az `sqlmetal.exe` segédprogram az `EntitySet<T>` típussal modellezi az összefüggő táblázatok közti kapcsolatokat. Az AutoLot adatbázis definiált három, egymáshoz kapcsolódó belső táblát, amelyeket elsődleges és idegen kulcsok kapcsoltak össze (lásd a 22. fejezet). Ahelyett, hogy SQL-központú `join` szintaxis megírására kényszerülnénk a táblázatok közti navigáláshoz, a LINQ to SQL lehetővé teszi, hogy az objektumcentrikus C# pont operátorral navigáljunk.

Az ilyesfajta táblakapcsolat érdekében a szülő-entitásosztály tulajdonságként 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 :
                    INotifyPropertyChanging, INotifyPropertyChanged
{
  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 táblá*ból* az Orders táblá*ba* történő navigálást az OtherKey tulajdonság által definiált oszlop révén. Az EntitySet<T> tagváltozót arra használjuk, hogy jelképezze az adott kapcsolat „egy a többhöz" (one-to-many) természetét.

Az erősen típusos DataContext

Az sqlmetal.exe által generált kód utolsó figyelemre méltó eleme a DataContextből származtatott típus. Az előző példában létrehozott AutoLotDatabase 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 DataContextből származó osztály mutatja, hogyan dolgozhatunk 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 örökölt `ExecuteMethodCall()` metódust (és reflexiós szolgáltatást) használja, hogy biztosítsa a hívó számára a tárolteljárás-hívást, valamint az eredmény visszaadását.

A generált típusok használata

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

```
class Program
{
  const string cnStr =
        @"Data Source=(local)\SQLEXPRESS;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.Write("Enter ID: ");
    carID = int.Parse(Console.ReadLine());
```

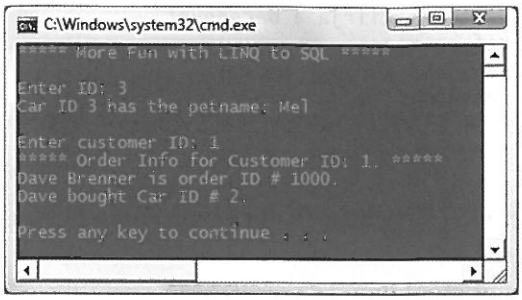
```
    // Meghívja a tárolt eljárást, és kiírja a becenevet.
    carsDB.GetPetName(carID, ref petName);
    Console.WriteLine("Car ID {0} has the petname: {1}",
                      carID, petName);
  }
}
```

A LINQ to SQL teljes mértékben elrejti előlünk az alapul szolgáló tárolt eljárás logikáját. Itt nem kell manuálisan `SqlCommand` objektumot létrehozni, feltölteni a paramétergyűjteményt vagy meghívni az `ExecuteNonQuery()` metódust. Ehelyett egyszerűen meghívjuk a `DataContext`ből származtatott típus `GetPetName()` metódusát. A kimeneti paraméterek referenciaparaméterként jelennek meg, ezért a C# `ref` kulcsszó használatával kell meghívni őket.

Tételezzük fel, hogy van egy másik segédfüggvényünk (amelyet szintén a `Main()` metódusból hívunk meg), ennek neve `PrintOrderForCustomer()`. Ez a metódus kiír néhány megrendelésinformációt a megadott ügyféllel kapcsolatban, valamint az ügyfél kereszt- és vezetéknevét:

```
static void PrintOrderForCustomer(AutoLot carsDB)
{
  int custID = 0;
  Console.Write("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 kérdezünk le.



24.5. ábra: *Egy adott ügyfél rendelési információinak kiírása*

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

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

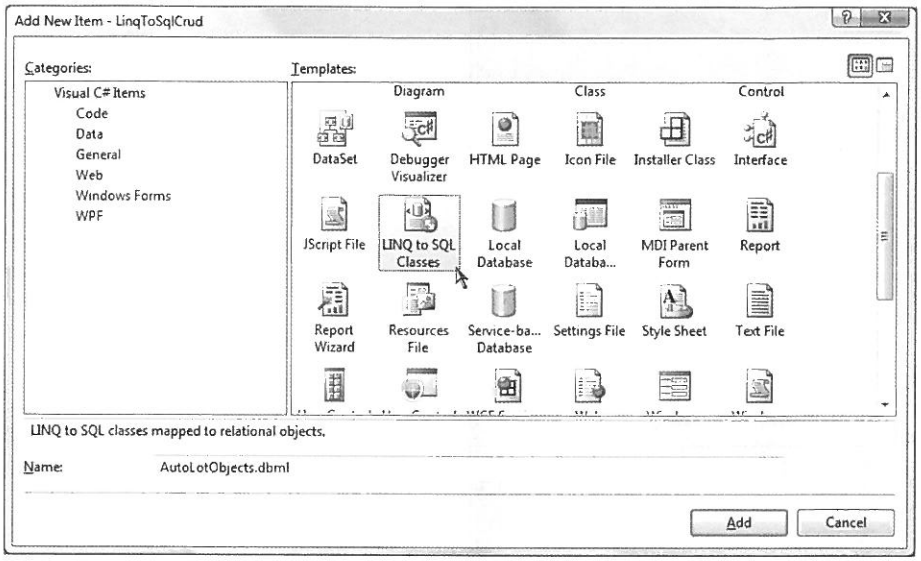
Ha újra lefuttatjuk a programot, azt tapasztalhatjuk, hogy a lekérdezéskifejezésünk sztringformátumú értéke az alapul szolgáló SQL-lekérdezést mutatja:

```
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])
```

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

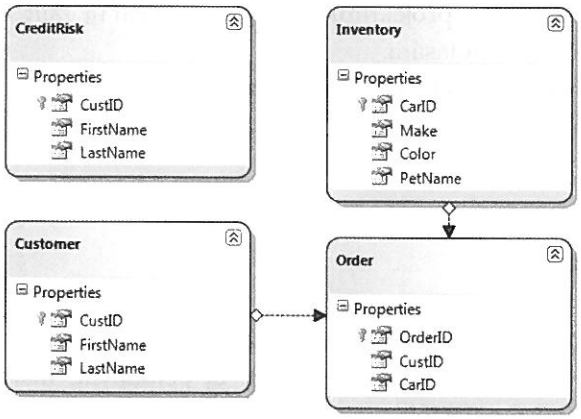
Entitásosztályok létrehozása a Visual Studio 2008 használatával

Végül hozzunk létre új parancssori konzolalkalmazást `LinqToSqlCrud` néven, valamint adjunk hozzá egy referenciát a `System.Data.Linq.dll` szerelvényre. Ezúttal ahelyett, hogy az `sqlmetal.exe` segítségével generálnánk az entitásosztályainkat, a Visual Studio 2008-ra bízzuk a „piszkos munkát". Ehhez válasszuk a Project > Add New Item menüpontot, majd adjunk hozzá egy új LINQ to SQL Classes elemet `AutoLotObjects` néven (lásd a 24.6. ábrát).



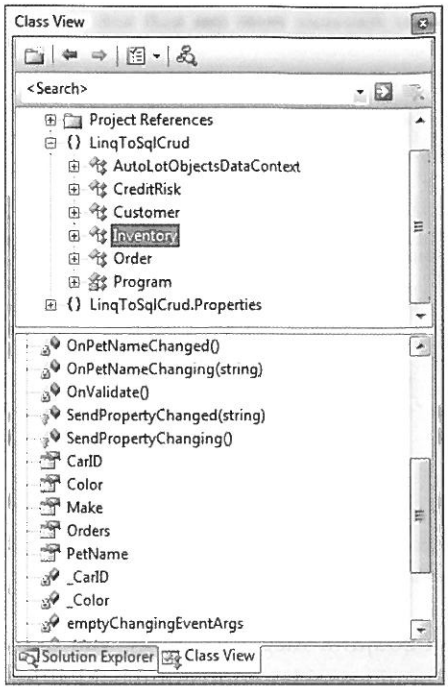
24.6. ábra: *A LINQ to SQL Classes elem ugyanazokat a feladatokat végzi el, mint az sqlmetal.exe*

Nyissuk meg a Server Explorert, és győződjünk meg róla, hogy van aktív kapcsolatunk az AutoLot adatbázissal (ha nincs, akkor jobb gombbal kattintsunk a Data Connections ikonra, és válasszuk az Add Connectiont). 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 névtér 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 projektünkhöz egy `app.config` fájlt a szükséges kapcsolatsztringadatok tárolására.

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

Új elemek beszúrása

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 objektuminicializáló szintaxist használja:

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

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

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

  // Új sor hozzáadása a "tömör" objektuminicializáló szintaxis
  // használatával.
  newCar = new Inventory { Make = "BMW", Color = "Silver",
                           PetName = "Henry", CarID = newCarID };

  ctx.Inventories.InsertOnSubmit(newCar);
  ctx.SubmitChanges();
}
```

Létező elemek módosítása

Egy elem módosítása meglehetősen egyértelmű. 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 rózsaszínre.
  var betty = (from c in ctx.Inventories
               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(AutoLotObjectsDataContext ctx)
{
  int carToDelete = 0;
  Console.Write("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";

  AutoLotObjectsDataContext ctx =
                    new AutoLotObjectsDataContext(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 belemerüljünk a részletekbe.

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

XML-dokumentumok kezelése a LINQ to XML használatával

Végezetül bemutatjuk a LINQ to XML szerepét, amely azt lehetővé teszi, hogy LINQ-lekérdezéskifejezéseket alkalmazzunk XML-dokumentumokon. Az XML-adatok reprezentációja mélyen beágyazódott a .NET keretrendszerébe.

Az alkalmazások és a webalapú konfigurációs fájlok XML-ként tárolják az adatokat. Az ADO.NET `DataSetek` könnyedén elmentik (vagy betöltik) XML-ként az adatokat. A Windows Presentation Foundation egy XML-alapú nyelvtant (XAML) használ az asztali felhasználói felületek ábrázolására, a Windows Communication Foundation (valamint a .NET-remoting API) szintén számos beállítást tárol olyan jól formázott sztringként, amelyet XML-nek hívunk.

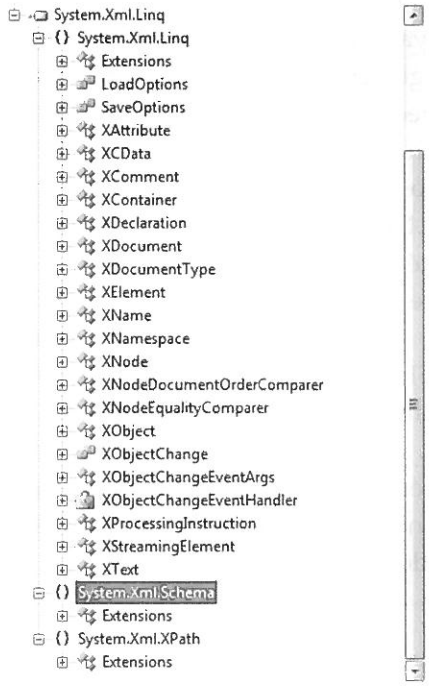
Noha az XML valóban mindenhol jelen van, programozása mindig is unalmas, terjengős és összetett volt, ha valaki nem volt tisztában számos XML-technológiával (XPath, XQuery, XSLT, DOM, SAX stb.). A .NET platform kezdetei óta a Microsoft egy olyan speciális szerelvényt biztosít az XML-dokumentumok programozásához, amelynek `System.Xml.dll` a neve. Ezen a bináris fájlon belül számos névtér és típus található a különböző 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 megszerezhetünk adatrészhalmazokat egy meglévő 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.XLinq névtér

Az alapvető LINQ to XML szerelvény (`System.Xml.Linq.dll`) 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.dll névterei*

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

A System.Xml.Linq tagja	Jelentés
`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.

A System.Xml.Linq tagja	Jelentés
`XElement`	Egy adott elemet jelképez egy XML-dokumentumban.
`XName/XNamespace`	Nagyon egyszerű módszert biztosít XML-névterek definálására és hivatkozására.

24.3. táblázat: *A System.Xml.Linq névtér néhány tagja*

Ennek (és más) típusoknak a vizsgálatához hozzunk létre egy parancssori konzolalkalmazást `LinqToXmlBasics` névvel, majd importáljuk a `System.Xml.Linq` névteret a kezdeti kódfájlba.

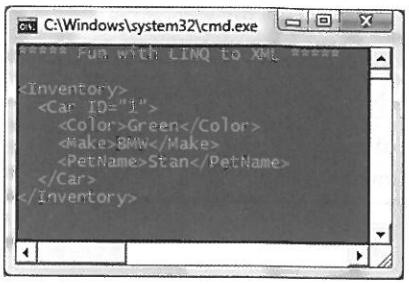
XML-dokumentumok létrehozása programozottan

Az eredeti .NET XML programozási modellel (`System.Xml.dll`) ellentétben XML-dokumentumot LINQ-val csak *funkcionális módon* lehet kezelni. Így ahelyett, hogy létrehoznánk egy dokumentumot a nagyon terjengős DOM API-val, a LINQ to XML lehetővé teszi, hogy ezt „DOM-mentesen" tegyük meg.

Ezzel nemcsak nagymértékben csökken a szükséges kód mennyisége, de a programozási modell leképezhető majdnem közvetlenül a jól formált XML-adatok formátumába. Ennek bemutatásához adjunk hozzá egy metódust a `Program` osztályunkhoz `CreateFunctionalXmlElement()` névvel, az alábbi megvalósítással:

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

Az `inventory` `XElement` objektumának konstruktora valójában további `XElement` és `XAttribute` 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.



24.10. ábra: *XML-dokumentum létrehozásának funkcionális megközelítése*

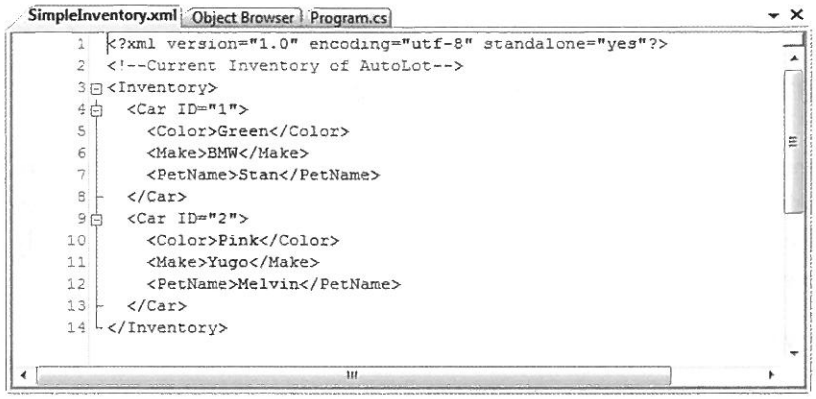
Ahhoz, hogy egy teljes XML-dokumentumot hozzunk létre a memóriában (megjegyzésekkel, feldolgozási utasításokkal, nyitó deklarációkkal stb.), megadhatjuk az objektumfát egy `XDocument` típus konstruktorába. Gondoljuk végig a következő `CreateFunctionalXmlDoc()` metódust, amely először létrehoz egy dokumentumot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

```
static void CreateFunctionalXmlDoc()
{
  XDocument inventoryDoc =
    new XDocument(
        new XDeclaration("1.0", "utf-8", "yes"),
        new XComment("Current Inventory of AutoLot"),
            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` fájlt.

Az `XElement` és az `XDocument` típusok meghatároznak egy olyan konstruktort, amely `XName` típust vesz 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-

jektumok paramétertömbje tartalmazhat bármennyi további LINQ to XML típust (`XComment`, `XProcessingInstruction`, `XElement`, `XAttribute` stb.), illetve egyszerű sztringeket (elemtartalomhoz) vagy az `IEnumerable` típust implementáló objektumot.



24.11. ábra: *A rögzített XML-dokumentum*

Dokumentumok létrehozása LINQ-lekérdezésekből

Ami az utolsó pontot illeti, tételezzük fel, hogy van egy névtelen típusokból 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 XAttribute("ID", c.ID),
                       new XElement("PetName", c.PetName)
                )
            );
  Console.WriteLine(vehicles);
}
```

XML-tartalom betöltése és elemzése

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

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

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

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

Navigálás egy memóriában lévő dokumentumban

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

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 hozzunk létre új parancssori konzolalkalmazást `NavigationWithLinqToXml` néven, és importáljuk a `System.Xml.Linq` névteret. Ezután adjunk hozzá egy új XML-dokumentumot aktuális projektünkhöz `Inventory.xml` névvel, ez bejegyzések kis gyűjteményét tárolja a gyökér `<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 Explorerben, és használjuk a Properties ablakot, hogy átállítsuk 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 betöltse 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();
}
```

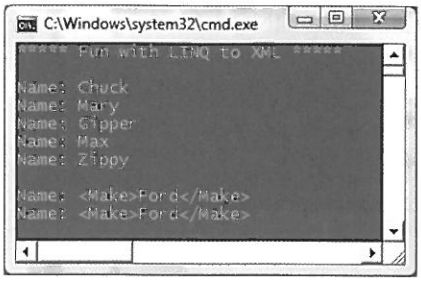
A `PrintAllPetNames()` metódus bemutatja az `XElement` használatát. A `Descendants()` metódus lehetővé teszi egy olyan adott alelem meghatározását, amelyhez el akarunk navigálni annak érdekében, hogy LINQ-lekérdezéskifejezést alkalmazzunk. Ekkor kiválasztunk minden `PetName` értéket, és kiíratjuk a tartalmukat 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("Name: {0}", name);
}
```

A `GetAllFords()` metódus nagyon hasonló. A bejövő `XElement` alapján definiálunk egy `where` operátort, és kiválasztjuk az összes `XElementet`, 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.



24.12. ábra: *A LINQ to XML működés közben*

Adatok módosítása egy XML-dokumentumban

A LINQ to XML több módszert is biztosít XML-tartalom beszúrására, törlésére, másolására és módosítására. Új `XElement` hozzáadása egy meglévő `XElement` (vagy `XDocument`) típushoz csak annyiból áll, hogy az `Add()` metódust meghívjuk, amely hozzáadja az adatot az elem/dokumentum végéhez. Ennek alternatívájaként meghívhatjuk az `AddFirst()` metódust, hogy az adatot az elem/dokumentum elejéhez adja, illetve az `AddAfterThis()/AddBeforeThis()` metódust, hogy az adatot egy meghatározott helyre szúrja be.

A tartalom módosítása vagy törlése elég egyértelmű. Miután létrehoztunk egy LINQ-lekérdezéskifejezést a manipulálni kívánt elem (vagy elemek) azonosítására, egyszerűen hívjuk meg a `ReplaceContent()` metódust (módosításhoz) vagy a `Remove()/RemoveContent()` metódust (adatok törléséhez). Egyszerű példaként vizsgáljuk meg a következő kódot, amely hozzáad néhány új `<Car>` elemet a bejövő `XElement` paraméterhez:

```
static void AddNewElements(XElement doc)
{
  // Hozzáad öt új lila Fordot a bejövő dokumentumhoz.
  for (int i = 0; i < 5; i++)
  {
    // Létrehoz egy új XElement objektumot.
    XElement newCar =
             new XElement("Car", new XAttribute("ID", i + 1000),
             new XElement("Color", "Green"),
             new XElement("Make", "Ford"),
             new XElement("PetName", "")
    );
    // Hozzáadás a dokumentumhoz.
    doc.Add(newCar);
  }
  // A módosítások megjelenítése.
  Console.WriteLine(doc);
}
```

A könyv további részében láthatunk még különböző LINQ-lekérdezéseket, ám az itt vizsgált API-k mindegyikét (LINQ to DataSet, LINQ to SQL és LINQ to XML) részletesen ismerteti a .NET Framework 3.5 SDK dokumentációja.

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

Összefoglalás

Ez a fejezet három LINQ-centrikus API bemutatásával bővítette a LINQ-kal kapcsolatos ismereteinket. Vizsgálódásunkat azzal kezdtük, hogy hogyan alakíthatunk át egy ADO.NET `DataSetet` LINQ-kompatibilis tárolóvá az `AsEnumerable()` bővítő metódus használatával. Az `IEnumerable<T>` típus elérése után képessé váltunk tetszőleges LINQ-lekérdezések alkalmazására.

A LINQ to DataSethez szorosan kapcsolódik a LINQ to SQL API, amely nemcsak azt teszi lehetővé, hogy lekérdezéskifejezéseket alkalmazzunk relációs adatbázisokban tárolt adatokra, hanem olyan infrastruktúrát is biztosít, amely lényegében beágyazza az alapul szolgáló ADO.NET-adattípusok minden formáját. Ahogyan láthattuk, a `DataContext` típus használható az összes megszokott adatbázissal kapcsolatos művelet végrehajtására, beleértve a tárolt eljárások meghívását is.

Végül a LINQ to XML API-t vizsgáltuk meg. A LINQ to SQL-hez hasonlóan ez is használható LINQ-lekérdezések megadására egy XML-dokumentum adatain, illetve XML-dokumentumok létrehozására funkcionális módon. A végeredmény látványos egyszerűsítése annak, hogy a .NET platform hogyan támogatja az XML-dokumentumok kezelését.

HUSZONÖTÖDIK FEJEZET

A WCF

A .NET 3.0-hoz, kifejezetten elosztott rendszerek készítéséhez fejlesztettek ki egy API-t, a Windows Communication Foundationt (WCF). A többi -korábban használt - elosztott API-val ellentétben (DCOM, .NET-remoting, XML-webszolgáltatások stb.) a WCF olyan egyszerű, egységesített és kiterjeszthető programozási objektummodellt kínál, amely sok, korábban szerteágazó elosztott technológiával működik együtt.

Először megnézzük, miért van szükség a WCF-re, valamint a korábbi elosztott API-k rövid áttekintésével megvizsgáljuk azokat a problémákat, amelyekre a WCF megoldást kínál. Ezután áttérünk a programozási modellt képező .NET-szerelvényekre (és alapvető típusokra), végül pedig WCF-szolgáltatásokat, -hosztokat és -klienseket készítünk különböző WCF fejlesztési eszközök segítségével.

Megjegyzés Jóllehet ez a fejezet kellő alapot nyújt a WCF-fejlesztéshez, a téma átfogó leírását azonban lásd Chris Peiris és Dennis Mulder *Pro WCF: Practical Microsoft SOA Implementation* című könyvében (Apress 2007).

Néhány elosztott API

A Windows operációs rendszer az idők folyamán több API-t is biztosított elosztott rendszerek építéséhez. Igaz, hogy az „elosztott rendszer" többnyire legalább két, hálózatba kötött számítógépet jelent, ám tágabb értelemben a kifejezés vonatkozhat egyszerűen két végrehajtható fájlra, amelyek adatokat cserélnek, még akkor is, ha történetesen mindkettő ugyanazon a gépen fut. Ha ezt a definíciót használjuk, a következő fontos kérdést kell feltenni, ha elosztott API-t választunk ki az aktuális programozási feladathoz:

A rendszert csak „házon belül" használják majd, vagy külső felhasználóknak is hozzáférést kell biztosítanunk az alkalmazás funkcionalitásához?

Ha házon belüli elosztott rendszert készítünk, akkor sokkal nagyobb az esély arra, hogy minden összekapcsolt számítógép ugyanazzal az operációs rendszerrel üzemel, valamint ugyanazt a programozási keretrendszert (.NET, COM, J2EE stb.) használja, így a meglevő biztonsági rendszert használhatjuk hitelesítésre, engedélyezésre és így tovább. Ebben az esetben a teljesítmény miatt hajlamosak lehetünk egy bizonyos API-t választani, olyat, amely egy bizonyos operációs rendszerhez vagy programozási keretrendszerhez köt bennünket.

Ám a kívülről is elérhető rendszer készítésekor jó néhány egyéb kérdést is figyelembe kell venni. Először is, nagy valószínűséggel a külső felhasználóknak *nem* lehet meghatározni, milyen operációs rendszert használjanak, milyen programozási keretrendszerben készítsék programjaikat, vagy hogyan konfigurálják a biztonsági beállításokat.

Továbbá, ha történetesen nagyobb vállalatnál vagy a felsőoktatásban dolgozunk, ahol többféle operációs rendszert és programozási technológiát is alkalmaznak, a házon belüli alkalmazás is ugyanazokkal a nehézségekkel fog szembenézni, mint a kívülről elérhető program. Mindkét esetben rugalmasabb elosztott API-t kell használni, hogy biztosítsuk az alkalmazás legjobb elérhetőségét.

Ha megvan a válasz erre a kulcskérdésre, a következő feladat a használni kívánt API (vagy API-k) kiválasztása. A továbbiakban röviden áttekintünk néhány olyan fontosabb elosztott API-t, amelyeket korábban a Windows-szoftverfejlesztők használtak. Ez az áttekintés meg fogja mutatni a Windows Communication Foundation hasznosságát.

Megjegyzés A WCF-nek (és az azt körülvevő technológiáknak) semmi köze a HTML-alapú webes alkalmazások fejlesztéséhez. Igaz, hogy a webes alkalmazásokat is „elosztottnak" tekinthetjük olyan értelemben, hogy az adatforgalomban tipikusan két számítógép vesz részt, de a WCF célja az, hogy számítógépekkel létesítsen kapcsolatot a távoli komponensek funkcionalitásának megosztásához – nem pedig HTML megjelenítése a böngészőben. (A 31. fejezet foglalkozik a .NET platformon a webhelyek fejlesztésének vizsgálatával.)

A DCOM szerepe

A .NET platform kiadása előtt a Distributed Component Object Model (DCOM - elosztott komponens-objektummodell) volt a választott elosztott API a Microsoft-központú fejlesztésekhez. A DCOM révén elosztott rendszereket lehetett építeni COM-objektumokkal, a rendszerleíró adatbázissal és persze fáradságos munkával. A DCOM egyik előnye az volt, hogy lehetővé tette a komponensek *elhelyezkedésének átlátszóságát*. Azaz ez lehetővé tette,

hogy a kliensprogramot úgy készítsük el, hogy a távoli objektumok elhelyezkedését nem „égettük" a kódba. Függetlenül attól, hogy a távoli objektumok ugyanazon a gépen vagy egy másik, hálózatban levő gépen voltak, az alapkód semleges maradhatott, mivel a tényleges elhelyezkedést a külső rendszerleíró adatbázis rögzítette.

Bár a DCOM valamennyire sikeres lett, de csak egy Windows-centrikus API maradt. Annak ellenére, hogy néhány más operációs rendszeren is rendelkezésünkre állt, önmagában nem volt elegendő olyan átfogó megoldások elkészítéséhez, amelyek többféle operációs rendszert használnak (Windows, Unix, MAC), vagy különböző architektúrák (COM, J2EE, CORBA stb.) között támogatják az adatmegosztást.

Megjegyzés Voltak próbálkozások, hogy a DCOM-ot átszabják többféle Unix/Linux verzióhoz, de a végeredmény középszerűre sikerült, és végül csak egy lábjegyzet lett a technológia történetében.

Összességében a DCOM leginkább házon belüli alkalmazások fejlesztéséhez volt a legmegfelelőbb, ugyanis a COM-típusok közzététele a cég falain túl további komplikációk sorát vonta maga után (tűzfalak és egyebek). A .NET platform megjelenésével a DCOM gyorsan túlhaladott programozási modell lett, és ha csak nem egy korábbi DCOM rendszert tartunk karban, valójában tekinthetjük elavult technológiának is.

A COM+/Enterprise Services szerepe

A DCOM önmagában nem tett sokkal többet, mint kommunikációs csatorna létesítésére definiált módszert két COM-alapú szoftver között. Sokoldalú, elosztott megoldások fejlesztéséhez, hogy a hiányzó részeket pótolja, a Microsoft végül kiadta a Microsoft Transaction Servert (MTS - Microsoft tranzakció szerver), amely egy későbbi kiadásban a COM+ nevet kapta.

Az elnevezés ellenére a COM+-t nemcsak COM-programozók használják - .NET-fejlesztők számára is teljes mértékben hozzáférhető. A .NET platform első kiadása óta biztosítják az alaposztálykönyvtárak a `System.EnterpriseServices` névteret. Itt a .NET-programozók felügyelt osztályokat készíthettek, amelyeket a COM+-futtatókörnyezetbe telepíthettek, hogy ugyanazokat a szolgáltatásokat érhessék el, mint a hagyományos COM+-alapú COM-kiszolgáló. Mindkét esetben a COM+-környezetbe telepített COM+-alapú könyvtárt *szolgáltatást használó komponensnek* neveztük.

A COM+ sok olyan funkciót kínál, amelyeket a szolgáltatást használó komponensek alkalmazhatnak, többek között a tranzakciókezelést, az objektumok élettartamának kezelését, szolgáltatáskészletek kialakítását (pooling), a szerepalapú biztonsági rendszert, lazán kapcsolt eseménymodellt és így tovább. Akkor ez jelentős előny volt, hiszen a legtöbb elosztott rendszernek ugyanazokra a szolgáltatásokra van szüksége. Ahelyett, hogy a fejlesztőket manuális kódolására kényszerítette volna, a COM+ kész megoldást biztosított.

A COM+ egyik pozitívuma az volt, hogy minden ilyen beállítást deklaratív módon adminisztrációs eszközök segítségével lehetett konfigurálni. Így, ha biztosítani akartuk, hogy egy objektumot tranzakció közben meg lehessen figyelni, vagy egy bizonyos biztonsági szerephez tartozzon, egyszerűen csak be kellett kapcsolni a megfelelő jelölőnégyzeteket.

Bár a COM+/Enterprise Services technológiát ma is használjuk, ez is csak a házon belüli fejlesztéshez megfelelő Windows-megoldás, vagy olyan háttérszolgáltatásként használható, amelyet a harciasabb front end közvetetetten kezel (pl. egy publikus webhely, amely szolgáltatást használó komponenseket [más néven COM+-objektumokat] hív meg a háttérben).

Megjegyzés A WCF-ben jelenleg nincs mód szolgáltatást használó komponensek készítésére, arra azonban van lehetőség, hogy a WCF-szolgáltatások létező COM+-objektumokkal kommunikáljanak. Ha C# segítségével szeretnénk szolgáltatást használó komponenseket létrehozni, közvetlenül a `System.EnterpriseServices` névteret kell használnunk. További részleteket a .NET Framework 3.5 SDK dokumentációjában találunk.

Az MSMQ szerepe

A Microsoft Message Queuing (MSMQ) API olyan elosztott rendszerek fejlesztésére szolgál, amelyeknek biztosítaniuk kell az üzenetadatok megbízható kézbesítését a hálózaton. Minden elosztott rendszerben fennáll a kockázat, hogy a kiszolgáló nem működik, az adatbázis éppen offline, vagy valamilyen rejtélyes okból megszakadnak a kapcsolatok. Továbbá sok alkalmazásnak szüksége van az üzenetadatok tárolására a későbbi kézbesítéshez (más néven az *adat-sorbaállításra*).

Kezdetben az MSMQ csak alacsonyszintű C-alapú API-k és COM-objektumok gyűjteménye volt. A `System.Messaging` névtér segítségével a .NET-programozók is felhasználhatták az MSMQ-t, és így időnként csatlakoztatott alkalmazásokkal megbízhatóan kommunikáló programokat írhattak.

Végül, de nem utolsósorban a COM+-réteg beépítette az MSMQ funkcionalitását a futtatókörnyezetbe (egyszerűsített formában) a Queued Components (QC - sorba állított komponensek) technológia segítségével.

Függetlenül attól, hogy melyik programozási modellt használtuk az MSMQ-futtatókörnyezettel történő kommunikációra, végül az alkalmazások mindig megbízhatóan és pontosan kézbesítették az üzeneteket. A Windows operációs rendszerben a COM+-hoz hasonlóan az MSMQ is feltétlenül része az elosztott szoftverek készítéséhez szükséges eszközöknek.

A .NET-remoting szerepe

A .NET platform megjelenésével a DCOM gyorsan elavult. Helyette a .NET-remoting-réteget biztosították a `System.Runtime.Remoting` névteret alkotó .NET-alaposztálykönyvtárak. Ezzel az API-val több számítógép megoszthatja az objektumait, feltéve, ha mindegyikük a .NET platform alatt futtatja az alkalmazásait.

A .NET-remoting-API számos nagyon hasznos funkciót biztosított. Ezek közül az egyik az XML-alapú konfigurációs fájlok voltak, amelyek deklaratív módon definiálták a kliens- és a kiszolgálószoftver közötti alapvető összeköttetéseket. A `*.config` fájlokkal nagyon egyszerű radikálisan megváltoztatni az elosztott rendszer funkcionalitását: egyszerűen a konfigurációs fájlok módosításával és az alkalmazás újraindításával.

Emellett, mivel ezt az API-t csak .NET-alkalmazások tudják használni, többféleképpen tehetünk szert teljesítménynövekedésre, mert az adatokat kompakt bináris formában lehet kódolni, valamint paraméterek és visszatérési értékek definiálásakor a közös típusrendszer (CTS - Common Type System) előnyeit is kihasználhatjuk. De amíg a .NET-remoting lehetővé tette különböző operációs rendszereket használó elosztott rendszerek készítését (a MONO révén, lásd röviden az előző kötet 1. fejezetében, részletesen a B mellékletben), a más programozási architektúrákkal (pl. a J2EE) való együttműködés még mindig nem volt közvetlenül lehetséges.

Megjegyzés A könyv előző kiadásaiban egy teljes fejezetet szenteltünk a .NET-remoting-API-k témájának, de a WCF megjelenésével ez a fejezetet ebből a kiadásból kimarad. A .NET-remoting-API-kal kapcsolatos kihagyott fejezetet (címe: A .NET remoting réteg) térítésmentesen letölthetik az Apress webhelyről ennek a kiadásnak a vásárlói (http://www.apress.com).

Az XML-webszolgáltatás szerepe

A korábban említett elosztott API-k egyike sem nyújtott sok támogatást (ha egyáltalán nyújtott) ahhoz, hogy külső hívók *agnosztikus módon* hozzáférjenek a program funkcionalitásához. Ha arra van szükségünk, hogy a távoli objektumok szolgáltatásait felajánljuk *bármilyen* operációs rendszernek vagy bármilyen programozási modellnek, az XML-webszolgáltatások biztosítják ehhez a legegyszerűbb utat.

A hagyományos böngészőalapú webes alkalmazásokkal ellentétben a webszolgáltatásokkal egyszerűen felajánlhatjuk távoli komponensek funkcionalitását szabványos webprotokollokon keresztül. A .NET kezdeti kiadása óta a programozók kiváló támogatást kapnak XML-webszolgáltatások készítéséhez és felhasználásához a `System.Web.Services` névtéren keresztül. Tulajdonképpen a teljes funkciójú webszolgáltatás készítése sokszor nem több, mint a `[WebMethod]` attribútum hozzárendelése minden olyan nyilvános metódushoz, amelyhez hozzáférést szeretnénk biztosítani. Továbbá a Visual Studio 2008 azt is lehetővé teszi, hogy egy (vagy két) gombnyomással távoli webszolgáltatáshoz kapcsolódjunk.

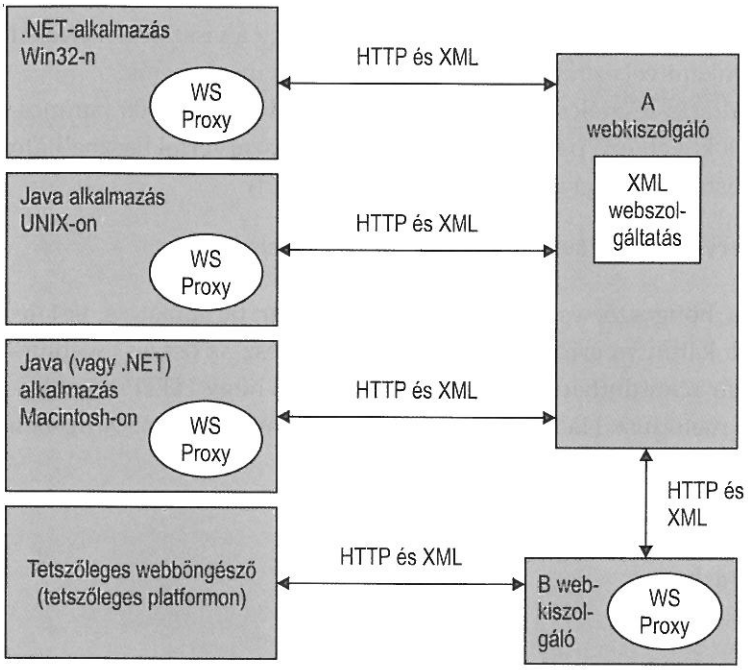
A webszolgáltatások segítségével olyan .NET-szerelvényeket lehet fejleszteni, amelyek egyszerű HTTP-n keresztül elérhető típusokat tartalmaznak, ráadásul a webszolgáltatás egyszerű XML-ként kódolja az adatait. Mivel a webszolgáltatások nyílt ipari szabványokon alapulnak (HTTP, XML, SOAP stb.), és nem védjeggyel rendelkező típusrendszereken vagy szabadalmaztatott műveletformátumokon (mint a DCOM vagy a .NET-remoting esetében), nagymértékű együttműködést és adatforgalmat tesznek lehetővé. A 25.1. ábra mutatja az XML-webszolgáltatások agnosztikus természetét.

Persze nincs tökéletes elosztott API. A webszolgáltatások egyik lehetséges hátránya teljesítménybeli hiányosságuk (mivel HTTP-t és XML-adatábrázolást használnak), így nem igazán ideálisak házon belüli alkalmazásokhoz, ahol a TCP-alapú protokoll és a bináris kódolás probléma nélkül használható lenne.

Példa .NET-webszolgáltatásra

A .NET-programozók évek óta a Visual Studio ASP.NET Web Service projektsablonjával készítenek webszolgáltatásokat, amelyet a File > New > Web Site párbeszédablakkal érhetünk el. Ez a projektsablon létrehozza az általában használt könyvtárstruktúrát, valamint néhány kiindulási fájlt, amelyek magát a webszolgáltatást képviselik.

Bár ezzel a sablonnal könnyen elindulhatunk, egyszerű szövegszerkesztővel is lehet .NET-es XML-webszolgáltatást készíteni, valamint az ASP.NET fejlesztői webszerverrel, a `WebDev.WebServer.exe`-vel pedig azonnal tesztelni is tudunk (a 31. fejezet bővebben ismerteti ezt az segédprogramot).



25.1. ábra: *Az XML-webszolgáltatások nagyfokú együttműködést tesznek lehetővé*

Egy rövid példa kedvéért tételezzük fel, hogy a következő programrészt készítettük el a `HelloWebService.asmx` fájlban (`*.asmx` a .NET-es XML webszolgáltatás-fájlok alapértelmezett kiterjesztése). Ha készen vagyunk, mentsük a C:\HelloWebService könyvtárba.

```
<%@ WebService Language="C#" Class="HelloWebService" %>
using System;
using System.Web.Services;

public class HelloWebService
{
  [WebMethod]
  public string HelloWorld()
  {
    return "Hello World";
  }
}
```

Bár ez az egyszerű szolgáltatás nem végez semmilyen különös feladatot, a fájl a `WebService` direktívával kezdődik, amellyel megadjuk a szolgáltatást jelképező osztálytípus nevét és a fájlban használt .NET programozási nyelvet. A `HelloWorld()` metódust a `[WebMethod]` attribútummal ruháztuk fel. Sokszor ennyi elég is ahhoz, hogy a metódust HTTP-n keresztül külső hívók számára elérhetővé tegyük. Végül semmi különös nem kell ahhoz, hogy a visszatérési értéket XML-ben kódoljuk, mivel ez futásidőben automatikusan megtörténik.

A webszolgáltatás tesztelésére nyissunk egy Visual Studio 2008 parancssort, és írjuk be a következő parancsot (a fejlesztői webszerverrel használható összes parancs listázásához csak írjuk be a `-?` paramétert):

```
webdev.webserver /port:8080 /path:"C:\HelloWebService"
```

Ekkor elindul a böngésző, és megmutatja a könyvtár tartalmát. A `HelloWorld.asmx` fájlra kattintva erről az oldalról is látható lesz az összes „webmetódus". Most már rákattinthatunk a `HelloWorld` linkre, hogy HTTP-n keresztül meghívjuk a metódust. Ha megtettük, a böngésző megjeleníti az XML-ben kódolt visszatérési értéket:

```
<?xml version="1.0" encoding="utf-8" ?>
<string xmlns="http://tempuri.org/">
  Hello World
</string>
```

Még ennél is könnyebb dolgunk van, hogy ha a webmetódusokat hívó ügyféloldali proxyfájlt szeretnénk generálni, ekkor ugyanis használhatjuk a `wsdl.exe` parancssori eszközt vagy a Visual Studio Add Service Reference pontját a Project menüben. Ugyanezekkel az eszközökkel generálhatunk ügyféloldali `*.config` fájlt, amely deklaratív formában tartalmazza a proxy különböző konfigurációs beállításait (pl. a webszolgáltatás végpontját).

A proxykód elrejti a kézi HTTP-beállítások részleteit és az XML-adatok visszafordítását .NET-adattípusokra. Tételezzük fel, hogy generáltunk egy proxyt ehhez az egyszerű webszolgáltatáshoz, ilyenkor a kliensalkalmazás nagyon egyszerűen képes meghívni a webmetódust, például:

```
static void Main(string[] args)
{
  // A proxytípus tartalmazza a kódot a *.config fájl olvasásához,
  // hogy megtalálja a webszolgáltatás helyét .
  HelloWebServiceProxy proxy = new HelloWebServiceProxy();
  Console.WriteLine(proxy.HelloWorld());
}
```

Megjegyzés A könyv előző kiadásaiban egy teljes fejezetet szenteltünk az XML-webszolgáltatások témájának, de a WCF megjelenésével ez a fejezet kimaradt ebből a kiadásból. A .NET-es XML-webszolgáltatásokról szóló kihagyott fejezetet (címe: XML-webszolgáltatások megértése) térítésmentesen letölthetik az Apress webhelyről ennek a kiadásnak a vásárlói (http://www.apress.com).

Bár még mindig abszolút tökéletes az ilyen „hagyományos" jellegű XML-webszolgáltatások készítése a .NET 3.5 alatt, a legtöbb új szolgáltatásprojektnek hasznára válik, ha ehelyett WCF-sablonokat használunk. Sok HTTP-alapú kötés közül választhatunk, és lényegében mindegyikkel készíthetünk webszolgáltatást anélkül, hogy kifejezetten valamelyik „webszolgáltatás"-projektsablont választanánk.

Forráskód A HelloWorldWebService kódfájlokat a forráskódkönyvtár 25. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Webszolgáltatási szabványok

A webszolgáltatások egyik fő problémája már a kezdetektől az volt, hogy minden nagy „játékos" a piacon (Microsoft, IBM és a Sun Microsystems) olyan webszolgáltatásokat implementált, amelyek nem voltak 100 százalékban kompatibilisek a többi webszolgáltatás-implementációval. Ez nyilvánvalóan problémát okozott, hiszen a webszolgáltatások lényege pont az, hogy elérjék: a különböző platformok és operációs rendszerek nagymértékben együtt tudjanak működni.

Hogy biztosítsák a webszolgáltatások együttműködését, olyan csoportok, mint a World Wide Web Consortium (W3C; http://www.w3.org) és a Web Services Interoperability Organization (WS-I; http://www.ws-i.org) elkezdtek különböző specifikációkat írni, amelyek részletesen megadják, hogy a szoftvergyártó cégek (pl. az IBM, a Microsoft vagy a Sun Microsystems) hogyan készítsék el webszolgáltatás-központú szoftverkönyvtáraikat a kompatibilitás megőrzésének érdekében.

Ezeknek a specifikációknak az együttes gyűjtőneve WS-*, és ezek tartalmazzák a biztonsági előírásokat, a mellékletek és a webszolgáltatások leírását (Web Service Description Language nyelven, azaz WSDL-ben), rendszabályokat, SOAP-formátumokat és rengeteg egyéb fontos részletet.

A Microsoft a legtöbb ilyen szabványának (felügyelt és nem felügyelt kód) az implementációját a Web Services Enhancements (WSE) 3.0 eszközrendszer tartalmazza, amely a támogatási webhelyről ingyenesen letölthető: http://msdn2.microsoft.com/en-us/webservices.

WCF-szolgáltatásalkalmazás készítésekor nem lesz közvetlenül szükség a WSE 3.0 eszközrendszer szerelvényeire. Ehelyett, a HTTP-alapú kötést használó WCF-szolgáltatás készítésekor ugyanezeket a WS-* specifikációkat készen megkapjuk (pontosan azokat, amelyek az általunk választott kötésen alapulnak).

Named pipe-ok, socketek és P2P

Ha a DCOM, .NET-remoting, webszolgáltatások, COM+ és az MSMQ közötti választás nem lett volna elég nehéz, tovább bővíthetjük az elosztott API-k listáját. A programozók használhatnak más folyamatok közötti kommunikációs API-kat, mint például a named pipe-okat, a socketeket és a peer-to-peer (P2P) szolgáltatásokat. Ezek az alacsonyabb szintű API-k általában jobb teljesítményt nyújtanak (különösen ugyanazon LAN hálózatban elhelyezkedő gépeken), de használatuk bonyolultabb (ha nem lehetetlen) kifelé kommunikáló alkalmazásoknál.

Ha több alkalmazást magában foglaló, de egyetlen gépen futó elosztott rendszert készítünk, célszerű lehet a named pipes API-t használni a `System.IO.Pipes` névtéren keresztül (újdonság a .NET 3.5-ben). Ez a módszer lehet a legesleggyorsabb lehetőség az ugyanazon a gépen belüli alkalmazások közötti adatcserére.

Ha olyan alkalmazást készítünk, amelynek teljes felügyelettel kell rendelkeznie a hálózati hozzáférés biztosításának és fenntartásának módjáról, a `System.Net.Sockets` és a `System.Net.PeerToPeer` névterekkel hasznosíthatjuk a socketek és a P2P funkcionalitását a .NET platform alatt.

A WCF szerepe

Az elosztott technológiák széles skálája nagyon megnehezíti a megfelelő eszköz kiválasztását. Ezt még tovább bonyolítja, hogy ezek közül több technológia szolgáltatásai átfedik egymást (legfőképpen a tranzakciók és a biztonság területén).

Még ha a .NET-fejlesztő kiválasztotta is a feladathoz megfelelőnek tűnő technológiát, az ilyen alkalmazás elkészítése, karbantartása és konfigurálása a legenyhébb kifejezéssel is összetett. Minden API-nak megvan a maga programozási modellje, egyéni konfigurációs segédprogramjai és így tovább.