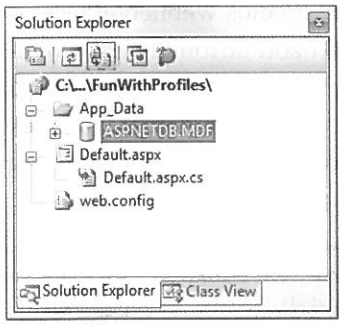
Miután az összes adatot az ASPNETDB.mdf fájlban rögzítettük, olvassuk ki az adatrészleteket az adatbázisból, majd formázzunk String típusként, amelyet a lblUserData címkén jelenítünk meg. Végül kezeljük az oldal Load eseményét, és jelenítsük meg ugyanezen adatokat a címkén. Így, amikor a felhasználó meglátogatja az oldalt, láthatja az aktuális beállításait. Íme, a teljes kódfájl:

```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
    GetUserAddress();
  }
  protected void btnSubmit_Click(object sender, EventArgs e)
  {
    // Itt történik az adatbázisírás!
    Profile.City = txtCity.Text;
    Profile.StreetAddress = txtStreetAddress.Text;
    Profile.State = txtState.Text;

    // Beállítások beolvasása az adatbázisból.
    GetUserAddress();
  }

  private void GetUserAddress()
  {
    // Itt történik az adatbázis-olvasás!
    lblUserData.Text = String.Format("You live here: {0}, {1}, {2}",
                Profile.StreetAddress, Profile.City, Profile.State);
  }
}
```

Ha lefuttatjuk a programot, azt tapasztaljuk, hogy az Default.aspx első letöltése előtt hosszú késleltetés van. Ennek az az oka, hogy a rendszer menetközben hozza létre az ASPNETDB.mdf fájlt, és az App_Data mappába helyezi. Ezt magunk is ellenőrizhetjük, ha frissítjük a Solution Explorer ablakot (lásd a 33.13. ábrát).



33.13. ábra: *Íme az ASPNETDB.mdf*

Továbbá azt tapasztaljuk, hogy az oldal első megnyitásakor az `lblUserData` címke nem jelenít meg semmilyen profiladatot, mivel még nem írtuk be az adatainkat az `ASPNETDB.mdf` adatbázis megfelelő táblájába. Miután beírjuk az adatokat a szövegdobozba, és visszaküldjük a kiszolgálónak, a 33.14. ábrán látható módon a címke a rögzített adatokat jeleníti meg.

33.14. ábra: *A rögzített felhasználói adataink*

A technológia legérdekesebb része csak most következik. Ha bezárjuk a böngészőt, és újra meglátogatjuk a webhelyet, azt látjuk, hogy a korábban megadott profiladatokat valóban rögzítettük, és a címke a megfelelő információkat mutatja. Felmerülhet a kérdés, hogy hogyan azonosított bennünket az alkalmazás.

Példánkban a profil-API a Windows hálózati azonosítót használta, amelyet az aktuális azonosítóadatokból nyert ki. Ha nyilvános webhelyet fejlesztünk (ahol a felhasználók nem tagjai egy meghatározott tartománynak), nyugodtak lehetünk, hogy a profil-API együttműködik az ASP.NET Forms-alapú hitelesítési modelljével, és támogatja a „névtelen profilok" fogalmát, amelyek használatával azon felhasználók profiladatait tárolhatjuk, akik pillanatnyilag nem rendelkeznek az oldalon érvényes azonosítóval.

Megjegyzés A könyv jelen kiadása nem foglalkozik az ASP.NET biztonsági megoldásaival (például a Forms-alapú hitelesítéssel vagy a névtelen profilokkal). További információkért lapozzuk fel a .NET Framework 3.5 SDK dokumentációját.

Profiladatok csoportosítása és egyedi objektumok tárolása

A fejezet végén szót kell ejtenünk arról, hogy a profiladatokat hogyan definiálhatjuk a `Web.config` fájlban. A jelenlegi profil egyszerűen négy adatot határozott meg, amelyeket közvetlenül a profiltípusból érhetünk el. Bonyolultabb profilok készítése során segítséget jelenthet, ha az összetartozó adatokat egyedi néven csoportosítjuk. Nézzük meg a következőt módosítást:

```
<profile>
  <properties>
    <group name ="Address">
      <add name="StreetAddress" type="String" />
      <add name="City" type="String" />
      <add name="State" type="String" />
    </group>
    <add name="TotalPost" type="Integer" />
  </properties>
</profile>
```

Ezúttal meghatároztuk az `Address` egyedi csoportot, amely a felhasználó címét (utca, város, ország) tartalmazza. Ahhoz, hogy az oldalainkon az adatokat elérjük, a forráskódot módosítanunk kell, és a részelemek beolvasásához meg kell adnunk a `Profile.Address` objektumot. Például a `GetUserAddress()` metódust a következőképp kell módosítani (a gomb kattintási eseménykezelőjét hasonló módon kell frissítenünk):

```
private void GetUserAddress()
{
  // Itt történik az adatbázis-olvasás!
  lblUserData.Text = String.Format("You live here: {0}, {1}, {2}",
             Profile.Address.StreetAddress,
             Profile.Address.City, Profile.Address.State);
}
```

Megjegyzés A profilon belül tetszőleges számú csoportot létrehozhatunk. Egyszerűen definiáljunk több `<group>` elemet a `<properties>` hatókörön belül.

Végül pedig érdemes megemlíteni, hogy a profil egyedi objektumokat tárolhat (és olvashat be) az `ASPNETDB.mdf` adatbázisban. A példa kedvéért tételezzük fel, hogy egyedi osztályt (vagy struktúrát) készítünk, amely a felhasználó címadatait képviseli. A profil-API egyetlen követelménye, hogy a típust megjelöljük a `[Serializable]` attribútummal, például így:

```
[Serializable]
public class UserAddress
{
  public string Street = string.Empty;
  public string City = string.Empty;
  public string State = string.Empty;
}
```

Az osztály elkészítése után a következőképpen módosíthatjuk a profil definícióját (bár nem kötelező, de a példából eltávolítottuk az egyedi csoportot):

```
<profile>
  <properties>
    <add name="AddressInfo"
         type="UserAddress"
         serializeAs ="Binary"/>
    <add name="TotalPost" type="Integer" />
  </properties>
</profile>
```

Vegyük észre, hogy amikor a `[Serializable]` típusokat adjuk a profilhoz, a típus attribútuma a rögzített típus teljesen meghatározott neve. Ha például `ArrayList` típusú objektumot szeretnénk a profilban tárolni, típusként a `System.Collections.ArrayList` értéket kell beállítanunk. A `serializeAs` attribútum segítségével szabályozhatjuk, hogy az állapotadatokat miként mentjük az `ASPNETDB.mdf` adatbázisban. A Visual Studio 2008 IntelliSense segítségével láthatjuk, hogy alapvetően bináris, XML- vagy sztringadatok közül választhatunk.

Mivel a címadat-információkat egyedi osztályként rögzítjük, a forráskódot (ismét) módosítanunk kell a következőképpen:

```
private void GetUserAddress()
{
  // Itt történik az adatbázis-olvasás!
  lblUserData.Text = String.Format("You live here: {0}, {1}, {2}",
              Profile.AddressInfo.Street, Profile.AddressInfo.City,
              Profile.AddressInfo.State);
}
```

A profil jóval többet tud annál, mint amit ebben a fejezetben bemutathatunk. A `Profile` tulajdonság például valójában a `ProfileCommon` típust foglalja magában. A típus segítségével programozottan lekérdezhetünk információt adott felhasználóról, törölhetünk (vagy felvehetünk új) profilokat az `ASPNETDB.mdf` adatbázisból, frissíthetjük a profilt, stb.

Ezen kívül a profil-API működését több módon bővíthetjük, és így optimalizálhatjuk, hogy a profilkezelő hogyan férjen hozzá az `ASPNETDB.mdf` adatbázis tábláihoz. Elvárásainknak megfelelően, többféleképp csökkenthetjük az adatbázis által visszaadott találatok számát. Ha szeretnénk többet megtudni a profil-API-ról, lapozzuk fel a .NET Framework 3.5 SDK dokumentációját.

Forráskód A FunWithProfiles kódfájlokat a forráskódkönyvtár 33. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Összefoglalás

Ebben a fejezetben kiegészítettük eddigi ASP.NET tudásunkat a `HttpApplication` típus használatának vizsgálatával. Láthattuk, hogy a típus több alapértelmezett eseménykezelővel rendelkezik, amelyekkel különböző alkalmazás- és munkamenetszintű eseményeket kezelhetünk. A fejezet nagyobb részében több állapotkezelő módszert vizsgáltunk meg. Emlékezzünk rá, hogy a nézetállapot segítségével a HTML-vezérlők automatikusan visszakapják eredeti értéküket a visszaküldések között. Később megvizsgáltuk az alkalmazás- és munkamenetszintű adatok közötti különbséget, a sütik kezelését és az ASP.NET alkalmazásgyorsítótárat.

A fejezet hátralévő részében az ASP.NET profil-API-ját ismertük meg. Láthattuk, hogy ez a technológia kész megoldást biztosít a felhasználói adatok munkamenetek közötti tárolására. A webhely `Web.config` fájljának segítségével tetszőleges számú profilelemet határozhatunk meg (többek között csoportosított elemeket és `[Serializable]` típusokat), amelyeket az ASP.NET automatikusan eltárol az `ASPNETDB.mdf` adatbázisban.

8. rész

Függelékek

A FÜGGELÉK

A COM és a .NET együttműködése

A könyv célja az, hogy a C# nyelvhez és a .NET platform által nyújtott alapvető szolgáltatásokhoz szilárd alapot biztosítson. Ha szembeállítjuk egymással a .NET által biztosított objektummodellt és a Microsoft korábbi komponensarchitektúráját (COM), nyilvánvalóvá válik, hogy két teljesen egyedi rendszerről van szó. Függetlenül attól, hogy a COM már hagyományos keretrendszernek tekinthető, lehetnek olyan COM-alapú rendszereink, amelyeket szeretnénk az új .NET-alkalmazásainkba integrálni.

A .NET platform szerencsére olyan különböző típusokat, eszközöket és névtereket biztosít, amelyek meglehetősen egyszerűvé teszik a COM és a .NET együttműködését. Az A függelék a .NET és a COM együttműködési folyamatának, valamint a kapcsolódó RCW-nek (Runtime Callable Wrapper – futási időben hívható burkoló) a vizsgálatával kezdődik. A második rész az ezzel ellentétes szituációt vizsgálja: COM-típus kommunikál .NET-típussal a CCW (COM Callable Wrapper – COM-ból hívható burkoló) segítségével.

Megjegyzés A .NET-együttműködési réteg teljes áttekintése külön könyvet igényelne. Ha több információra van szükségünk, mint amennyit a függelékben találhatunk, forduljunk a *COM and .NET Interoperability* (Apress, 2002) című munkához.

A .NET együttműködési képesség hatóköre

Amikor .NET-képes fordító használatával készítünk szerelvényeket, akkor olyan *felügyelt kódot* írunk, amelyet a közös nyelvi futtatórendszer (CLR) hosztolhat. A felügyelt kódnak több előnye van, ilyen például az automatikus memóriakezelés, az egységes típusrendszer (CTS), az önleíró szerelvények és így tovább. A .NET-szerelvények sajátságos belső felépítéssel rendelkeznek. A CIL-utasítások és a típusmetaadatok mellett a szerelvények olyan

manifesztumot tartalmaznak, amely teljes körűen dokumentálja a szükséges külső szerelvényeket, illetve a fájlokhoz kapcsolódó további részleteket (erős név, verziószám stb.)

A spektrum másik végén a korábbi COM-kiszolgálókat találjuk (amelyek természetesen *nem felügyelt kódok*). Ezek a bináris fájlok a közös fájlkiterjesztésen (`.dll` vagy `.exe`) kívül semmilyen más tekintetben nem kapcsolódnak a .NET-szerelvényekhez. Először is a COM-kiszolgálók platformfüggő gépi kódot tartalmaznak, nem pedig platformfelismerő CIL-utasításokat, valamint egyedi adattípusok készletével dolgoznak (amelyet gyakran *OLE-automatizmusnak* vagy *variánsengedékeny* adattípusoknak nevezünk), amelyek egyikét sem érti meg közvetlenül a CLR.

A COM bináris fájlok (mint a rendszerleíró adatbázis bejegyzései és az `IUnknownhoz` hasonló alapvető COM-interfészek támogatása) által igényelt COM-központú infrastruktúra mellett a COM-típusoknak *referenciaszámláltaknak* kell lenniük, hogy megfelelően szabályozhassák a COM-objektumok élettartamát. Természetesen ez éles kontrasztban áll azokkal a .NET-objektumokkal, amelyeket a felügyelt heapen foglalunk le, és a CLR szemétgyűjtője kezeli őket.

Mivel a .NET- és a COM-típusokban nincs sok közös vonás, felmerülhet a kérdés: vajon a két architektúra hogyan használhatja egymás szolgáltatásait? Hacsak nem vagyunk olyan szerencsések, hogy „kizárólag" .NET-fejlesztéseket alkalmazó cégnél dolgozunk, akkor minden bizonnyal olyan .NET-megoldásokat kell majd fejlesztenünk, amelyek korábbi COM-típusokat használnak. Emellett szembesülhetünk azzal, hogy egy korábbi COM-kiszolgáló esetleg kommunikálni szeretne egy vadonatúj .NET-szerelvény típusaival.

A lényeg, hogy a COM-nak és a .NET-nek a közeljövőben meg kell tanulniuk együttműködni. A következőkben megvizsgáljuk a felügyelt és nem felügyelt típusok harmonikus együttélésének folyamatát a .NET együttműködési réteg használatával. Általánosságban, a .NET keretrendszer az együttműködés két alapvető fajtáját támogatja:

- COM-típusokat használó .NET-alkalmazásokat,
- .NET-típusokat használó COM-alkalmazásokat.

A .NET Framework 3.5 SDK és a Visual Studio 2008 több olyan eszközt biztosít, amelyek segítenek áthidalni a szakadékot a két egyedi architektúra között. A .NET-alaposztálykönyvtár emellett meghatároz egy olyan névteret (`System.Runtime.InteropServices`), amelyet egyedül az együttműködés támogatásának szenteltek. Nézzük meg először egy egyszerű példát arra, amikor egy .NET-osztály COM-kiszolgálóval kommunikál.

Megjegyzés A .NET platform jelentősen megkönnyíti, hogy egy .NET-szerelvény meghívhassa az operációs rendszer alapjául szolgáló API-ját (valamint bármilyen C-alapú, nem felügyelt `*.dll`-t) a *platformhívás* (vagy egyszerűen PInvoke) nevű technológiával. Ha a PInvoke-kal dolgozunk, a C#-tekintetében legalább a `[DllImport]` attribútumot alkalmaznunk kell a futtatni kívánt külső metódusra. A PInvoke technológiát nem vizsgáljuk meg a függelékben, további információkat a `[DllImport]` attribútumról a .NET Framework 3.5 SDK dokumentációjában találunk.

A .NET és a COM együttműködésének egyszerű példája

Ebben a részben egy Visual Basic 6.0 ActiveX `*.dll` kiszolgálót készítünk, amelyet aztán egy C#-alkalmazás fog felhasználni.

Megjegyzés Sok COM keretrendszer létezik a VB6-on kívül (pl. az Active Template Library [ATL] és a Microsoft Foundation Classes [MFC]). A függelékben a VB6-ot használjuk a COM-kiszolgáló létrehozására, ez ugyanis rendkívül felhasználóbarát szintaxist biztosít a COM-alkalmazások készítéséhez. Ám nyugodtan használjuk akár az ATL-t, akár az MFC-t.

Indítsuk el a VB6-ot, és hozzunk létre új ActiveX `*.dll` projektet SimpleComServer névvel, nevezzük át a kezdeti osztályfájlt `ComCalc.cls`-re, és az osztálynak adjuk a `ComCalc` nevet. A projekt nevét és a tárolt osztályokhoz rendelt neveket arra használjuk, hogy definiáljuk a COM-típusok (ebben az esetben `SimpleComServer.ComCalc`) program azonosítóját (`ProgID`). Végül definiáljuk a következő metódusokat a `ComCalc.cls` fájlban:

```
' A VB6 COM-objektum
Option Explicit

Public Function Add(ByVal x As Integer, ByVal y As Integer)
  As Integer
    Add = x + y
End Function

Public Function Subtract(ByVal x As Integer, ByVal y As Integer)
  As Integer
    Subtract = x - y
End Function
```

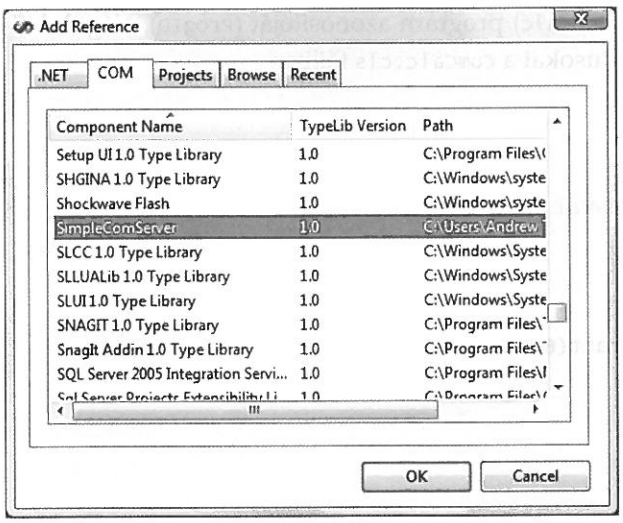
Fordítsuk le a `*.dll`-ünket (a File > Make menüpont segítségével), és állítsuk be a bináris kompatibilitást (a projekt Property oldalának Component lapján), mielőtt kilépnénk a VB6 integrált fejlesztői környezetből. Ez biztosítja, hogy ha majd újrafordítjuk az alkalmazást, a VB6 megőrzi a hozzárendelt globális egyedi azonosítókat (GUID).

Forráskód A SimpleComServer projektet a forráskódkönyvtár A Függelékének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A C#-ügyfélalkalmazás elkészítése

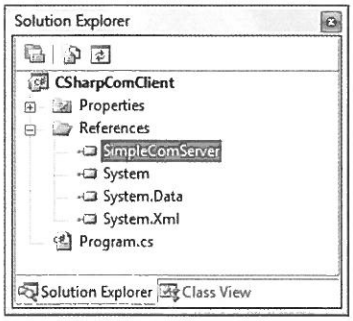
Nyissuk meg a Visual Studio 2008-at, és hozzunk létre egy új C# parancssori alkalmazást CSharpComClient névvel. Amikor olyan .NET-alkalmazást készítünk, amelynek korábbi COM-alkalmazással kell kommunikálnia, első lépésként hivatkoznunk kell a COM-kiszolgálóra a projektünkben (hasonlóan ahhoz, ahogyan egy .NET-szerelvényre hivatkozunk).

Ehhez használjuk a Project > Add Reference menüpontot, majd válasszuk a COM lapot az Add Reference párbeszédpanelen. Az ablak a COM-kiszolgálók nevét ábécésorrendben listázza, mivel a projekt fordítása során a VB6 fordító a szükséges listázásokkal frissítette a rendszerleíró adatbázist. Ahogy az A.1. ábrán látható, jelöljük ki a `SimpleComServer.dll`-t, majd zárjuk be a párbeszédablakot.

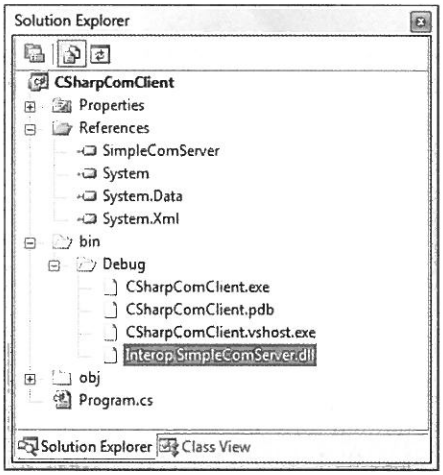


A.1. ábra: *Hivatkozás COM-kiszolgálóra a Visual Studio 2008 segítségével*

Ha megnézzük a Solution Explorer References mappáját, láthatunk valami olyasmit, ami a projekthez hozzáadott új .NET-szerelvényreferenciának tűnik (lásd az A.2. ábrát). Azokat a szerelvényeket, amelyeket a COM-kiszolgálóreferenciák létrehozása során generálunk, hivatalosan *együttműködési szerelvényeknek* hívjuk. Az együttműködési szerelvények a COM-típusok .NET-leírását tartalmazzák.



A.2. ábra: *A hivatkozott együttműködési szerelvény*



A.3. ábra. *Az automatikusan generált együttműködési szerelvény*

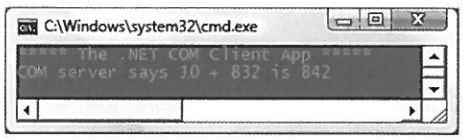
Noha még nem írtunk semmilyen kódot a kezdeti C#-osztálytípusunkba, ha most lefordítjuk az alkalmazást, és megvizsgáljuk a projekt bin\Debug könyvtárát, akkor láthatjuk, hogy a generált együttműködési szerelvény helyi másolata bekerült az alkalmazás könyvtárába (lásd az A.3. ábrát). A Visual Studio 2008 automatikusan hozzáadja az `Interop.` prefixumot az Add Reference párbeszédpanellel létrehozott együttműködési szerelvényekhez – ez azonban csak megegyezés; a CLR számára nem szükséges, hogy az együttműködési szerelvények ezt a sajátságos elnevezést kövessék.

Kezdeti példánk befejezéséhez módosítsuk a kezdő osztályunk `Main()` metódusát úgy, hogy meghívja az `Add()` metódust egy `ComCalc` objektumon, és megjelenítse az eredményt. Például:

```
using System;
using SimpleComServer;

namespace CSharpComClient
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** The .NET COM Client App *****");
      ComCalc comObj = new ComCalc();
      Console.WriteLine("COM server says 10 + 832 is {0}",
        comObj.Add(10, 832));
      Console.ReadLine();
    }
  }
}
```

Ahogy az előző példakódban láthattuk, a `ComCalc` COM-objektumot tartalmazó névtér neve megegyezik az eredeti VB6 projekt nevével (lásd a `using` utasítást). Az A.4. ábrán látható a kimenet.



A.4. ábra: *.NET és COM együttműködése*

Látható, hogy COM-típus felhasználása .NET-alkalmazásban valóban transzparens művelet lehet. Ám a háttérben sok minden történik, amely lehetővé teszi ezt a kommunikációt – ezek részleteit vizsgáljuk meg a következőkben, az együttműködési szerelvények részleteivel kezdve.

Egy .NET együttműködési szerelvény vizsgálata

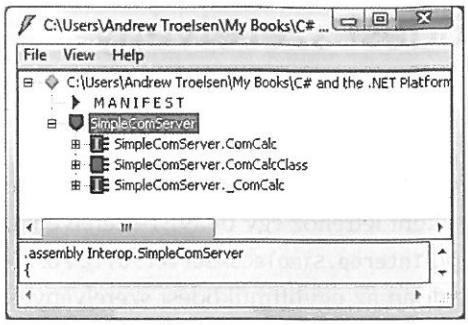
Ha egy COM-kiszolgálóra hivatkozunk a Visual Studio 2008 Add Reference párbeszédpaneljével, akkor az IDE válaszként létrehoz egy új .NET-szerelvényt, amelyet ellát az `Interop.` prefixummal (pl. `Interop.SimpleComServer.dll`). Az általunk létrehozott szerelvényekhez hasonlóan az együttműködési szerelvények is tartalmaznak típusmetaadatokat, szerelvénymanifesztumot, és – bizonyos körülmények között – *tartalmazhatnak* köztes nyelvi kódot. A „normális" szerelvényekhez hasonlóan az együttműködési szerelvényeket privát szerelvényként telepíthetjük (pl. az ügyfél szerelvénykönyvtárán belül), illetve rendelhetünk hozzájuk erős nevet, hogy telepíteni lehessen őket a globális szerelvénytárba (GAC).

Az együttműködési szerelvények alig többek, mint puszta tárolók az eredeti COM-típusok .NET-metaadat-leírásainak megőrzéséhez. Sokszor az együttműködési szerelvények nem foglalnak magukban CIL-utasításokat metódusaik implementálásához, hiszen az igazi munkát maga a COM-kiszolgáló végzi. Az együttműködési szerelvényekben csak akkor találunk végrehajtható CIL-utasításokat, ha a COM-kiszolgáló olyan COM-objektumokat tartalmaz, amelyek képesek eseményeket kiváltani az ügyfél számára. Ilyenkor az együttműködő szerelvényekben lévő köztes nyelvi kódot a CLR használja arra, hogy a COM kapcsolódási pontok eseménykezelő logikáját lefordítsa .NET-metódusreferenciákká.

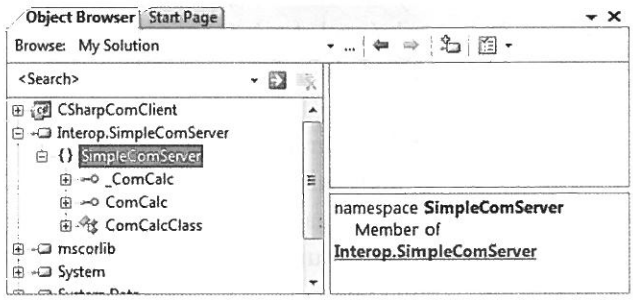
Első pillantásra az együttműködő szerelvények nem tűnnek túl hasznosak, hiszen nem tartalmaznak megvalósítási logikát. Az együttműködő szerelvényekben tárolt metaadat-leírások azonban rendkívül fontosak, ugyanis a CLR ezeket fogja felhasználni arra, hogy futási időben futásidejű proxyt készítsen (ez a *futási időben hívható burkoló* vagy röviden *RCW*), amely kommunikációs hídként szolgál a .NET-alkalmazás és a COM-objektum között.

Az RCW részleteit a későbbiekben még megvizsgáljuk; most nyissuk meg az `Interop.SimpleComServer.dll` szerelvényt az `ildasm.exe` segítségével (lásd az A.5. ábrát).

Noha az eredeti VB6 projekt egyetlen COM-osztályt (`ComCalc`) definiált, az együttműködési szerelvény *három* típust tartalmaz. Ezt ellenőrizhetjük a Visual Studio 2008 Objektum böngészőjével is (lásd az A.6. ábrát).



A.5. ábra: *Az Interop.SimpleComServer.dll együttműködési szerelvény lényege*



A.6. ábra: *Hogyan eredményezhet egyetlen COM-típus három .NET-típust?*

Minden COM-osztályt három különböző .NET-típus képvisel. Először is, van egy .NET-típusunk, amelynek a neve megegyezik az eredeti COM-típus nevével (ebben az esetben `ComCalc`). Másodszor, van egy .NET-típusunk, amely felveszi a `Class` utótagot (`ComCalcClass`). Ezek a típusok akkor nagyon hasznosak, ha van olyan COM-típusunk, amely több egyedi interfészt valósít meg, ugyanis a `Class` utótaggal rendelkező típusok tárják fel a COM-típus által támogatott *összes* interfész *összes* tagját. Így .NET-programozóként nem kell manuálisan létrehoznunk egy referenciát a speciális COM-interfészre, mielőtt annak funkcionalitását alkalmaznánk. Bár a `ComCalc` nem implementál több egyedi interfészt, az alábbiak szerint meghívhatjuk az `Add()` és a `Subtract()` metódusokat (`ComCalc` objektum helyett) egy `ComCalcClass` objektumon:

```
static void Main(string[] args)
{
  Console.WriteLine("***** The .NET COM Client App *****");

  // Most használjuk a Class utótaggal rendelkező típust.
  ComCalcClass comObj = new ComCalcClass();
  Console.WriteLine("COM server says 10 + 832 is {0}",
    comObj.Add(10, 832));
  Console.ReadLine();
}
```

Végül az együttműködő szerelvények meghatározzák a COM-kiszolgálón definiált eredeti COM-interfész .NET-ekvivalensét. Ebben az esetben egy `_ComCalc` nevű .NET-interfészt találunk. Hacsak nem vagyunk járatosak a VB6 COM szerkezetében, akkor ez bizonyára furcsának tűnhet, hiszen nem hoztunk közvetlenül létre interfészt a SimpleComServer projektünkben (most ne törődjünk a furcsa nevű `_ComCalc` interfésszel). Az aláhúzásjeles interfészek szerepe a későbbiekben világossá válik majd; egyelőre csak azt kell tudnunk, hogy használhatunk interfészalapú programozási módszereket az `Add()`, illetve a `Subtract()` metódusok meghívásához:

```
static void Main(string[] args)
{
  Console.WriteLine("***** The .NET COM Client App *****");

  // Most manuálisan szerezzük meg a rejtett interfészt.
  ComCalc itfComInterface = null;
  ComCalcClass comObj = new ComCalcClass();
  itfComInterface = (_ComCalc)comObj;

  Console.WriteLine("COM server says 10 + 832 is {0}",
    itfComInterface.Add(10, 832));
  Console.ReadLine();
}
```

Ritkán lesz szükségünk metódusok hívására a `Class` utótaggal rendelkező vagy az aláhúzásjeles interfészek használatával. Ha azonban olyan bonyolultabb .NET-alkalmazásokat készítünk, amelyeknek kifinomultabban kell együttműködniük a COM-típusokkal, akkor fontossá válik ezeknek a típusoknak az ismerete.

Forráskód A `CSharpComClient` projektet a forráskódkönyvtár A függelékének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

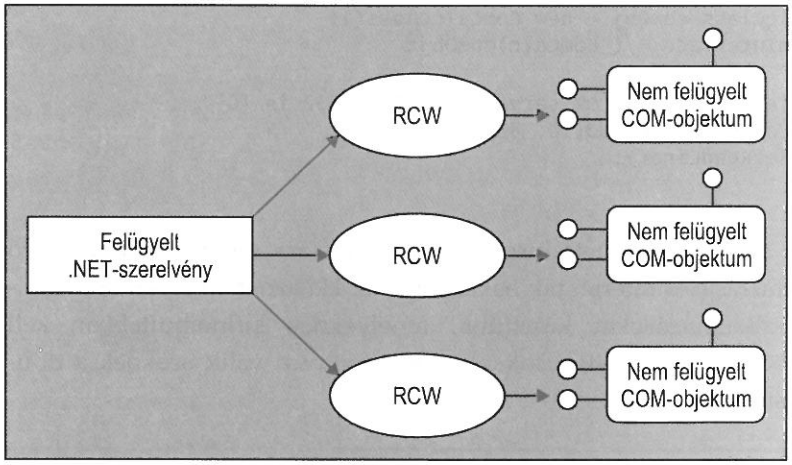
A futási időben hívható burkoló

A CLR futásidőben a .NET együttműködési szerelvény metaadatait használja fel olyan proxytípus létrehozásához, amely a .NET–COM-kommunikáció folyamatát kezeli. Ez a proxy a futási időben hívható burkoló, amelynek voltaképpen híd szerepe van a valódi COM-osztályhoz (gyakran a *coclass* elnevezéssel illetjük). Minden `coclass`, amelyhez egy .NET-kliens hozzáfér, megfe-

lelő RCW-t igényel. Ezért, ha .NET-alkalmazásunk három COM-coclasst használ, akkor végül három különböző RCW-nk lesz, amelyek a .NET-hívásokat a COM-kérésekre képezik le. Az A.7. ábra mutatja a teljes képet.

Megjegyzés COM-objektumonként mindig egyetlen RCW lesz, függetlenül attól, hogy a .NET-kliens hány interfészt kapott a COM-típustól (a többinterfészes VB6 COM-objektumokat a függelék későbbi részében vizsgáljuk). Ezzel a módszerrel az RCW karbantarthatja a COM-objektum megfelelő COM-azonosítóját (és referenciaszámlálóját).

Az RCW-t a rendszer automatikusan létrehozza akkor, amikor a CLR-nek szüksége van rá. A korábbi COM-kiszolgálókban pedig nincs szükség semmilyen módosításra ahhoz, hogy a .NET-alapú nyelvek feldolgozhassák őket. Az RCW közbenjár, és gondoskodik a belső munkáról. Ennek megismeréséhez formalizáljuk az RCW néhány alapvető feladatát.



A.7. ábra: *Az RCW-k a .NET-es hívó és a COM-objektum között helyezkednek el*

RCW: a COM-típusok mint .NET-típusok

Az RCW felelős a COM-adattípusok átalakításáért a .NET-ekvivalensükre (és fordítva). Tételezzük fel, hogy egy VB6 COM-szubrutinunk a következő definícióval rendelkezik:

```
' VB6 COM-metódus definíciója.
Public Sub DisplayThisString(ByVal s as String)
```

Az együttműködő szerelvény .NET `System.String`ként definiálja a metódusparamétert:

```
' A COM-metódus C#-beli leképezése.
public void DisplayThisString(string s)
```

Amikor a .NET-kódbázis meghívja a metódust, az RCW automatikusan felveszi a bejövő `System.String` típust, és átalakítja VB6 `String` adattípussá (amely valójában COM BSTR). Minden COM-adattípusnak megvan a maga .NET-ekvivalense. Az A.1. táblázat összefoglalja a COM IDL (interfészdefiníciós nyelv) adattípusok, a hozzájuk kapcsolódó .NET System adattípusok és a megfelelő C#-kulcsszavak (ha van) közötti leképezéseket.

COM IDL adattípus	Rendszertípusok	C#-kulcsszó
wchar_t, short	System.Int16	short
long,int	System.Int32	int
Hyper	System.Int64	long
unsigned char, byte	System.Byte	byte
single	System.Single	-
double	System.Double	double
VARIANT_BOOL	System.Boolean	bool
BSTR	System.String	string
VARIANT	System.Object	object
DECIMAL	System.Decimal	-
DATE	System.DateTime	-
GUID	System.Guid	-
CURRENCY	System.Decimal	-
IUnknown	System.Object	object
IDispatch	System.Object	object

A.1. táblázat: *Valódi COM-típusok leképezése .NET-típusokra*

RCW: coclassok referenciaszámlálójának kezelése

Az RCW másik fontos feladata a COM-objektum referenciaszámlálójának kezelése. A COM referenciaszámlálási sémája a közös pont a coclass és az ügyfélprogram között, középpontjában pedig az `AddRef()` és a `Release()` hívások megfelelő használata áll. A COM-objektumok megsemmisítik magukat, ha azt észlelik, hogy nincsenek külső hivatkozásaik.

A .NET-típusok azonban nem a COM referenciaszámlálási sémáját használják, ezért a .NET-kliensek nem tudják meghívni a `Release()` metódust a felhasznált COM-típusokon. Az RCW belsőleg gyorsítótárazza az összes interfészreferenciát, és aktiválja a felszabadításukat, ha a .NET-kliens már nem használja a típust. A lényeg az, hogy a VB6-hoz hasonlóan a .NET-kliensek soha nem hívják meg expliciten az `AddRef()`, a `Release()` és a `QueryInterface()` metódusokat.

Megjegyzés Ha közvetlenül a .NET-alkalmazásból szeretnénk hozzáférni a COM-objektum referenciaszámlálójához, akkor rendelkezésünkre áll a `Marshal` típus, amelyet a `System.Runtime.InteropServices` névtér biztosít. Ez az osztály több statikus metódust definiál, amelyek közül több használható arra, hogy manuálisan kezeljük a COM-objektumok élettartamát. Bár az alkalmazások többségében nem használjuk a `Marshal` típust, további részletekért forduljunk a .NET Framework 3.5 SDK dokumentációjához.

RCW: alacsony szintű COM-interfészek elrejtése

Az RCW által biztosított utolsó alapvető szolgáltatás az alacsony szintű COM-interfészek használata. Mivel az RCW mindent megtesz azért, hogy elhitesse a .NET-klienssel, hogy közvetlenül egy natív .NET-típussal kommunikál, ezért bizonyos alacsony szintű COM-interfészeket el kell rejtenie szem elől.

Amikor például olyan COM-osztályt készítünk, amely támogatja az `IConnectionPointContatiner` interfészt (és van egy vagy két alobjektuma, amely az `IConnectionPoint` interfészt támogatja), akkor a kérdéses coclass képes eseményeket kiváltani a COM-kliensnél. A VB6 az `Event` és a `RaiseEvent` kulcsszavakkal a teljes folyamatot elrejti szem elől. Ugyanígy az RCW a .NET-kliens elől elrejti a hasonló COM-„felesleget". Az A.2. táblázat felvázolja azoknak a rejtett COM-interfészeknek a szerepét, amelyeket az RCW használ.

Rejtett COM-interfész	Jelentés
`IConnectionPointContainer`	Lehetővé teszi, hogy a coclass eseményeket küldjön vissza az érdekelt kliensnek.
`IConnectionPoint`	A VB6 automatikusan biztosítja az interfészek alapértelmezett megvalósítását.
`IDispatch` `IProvideClassInfo`	Lehetővé teszi a „késői kötést" egy coclasshoz. Amikor VB6 COM-típusokat készítünk, ezeket az interfészeket automatikusan támogatja az adott COM-típus.

`IerrorInfo`
`IsupportErrorInfo`
`ICreateErrorInfo`
	Az interfészek lehetővé teszik, hogy a COM-kliensek és a COM-objektumok COM-hibákat küldjenek, és azokra válaszoljanak.

`IUnknown`	A COM nagyapja. Ez kezeli a COM-objektum referenciaszámlálóját, és lehetővé teszi, hogy a kliensek megkapják az interfészeket a coclasstól.

A.2. táblázat: *Rejtett COM- interfészek*

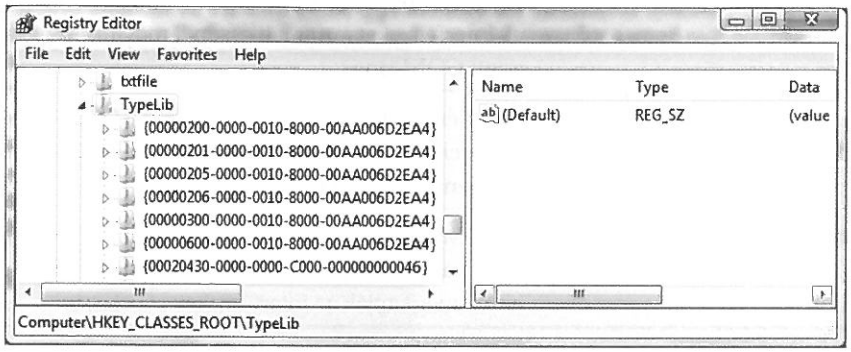
A COM IDL szerepe

A következőkben tekintsük át a COM IDL néhány apróbb részletét. A függeléknek *nem* célja, hogy teljes COM IDL gyakorlatot biztosítson, az együttműködési réteg jobb megértéséhez csak néhány IDL-szerkezetet kell ismernünk.

Minden .NET-szerelvény tartalmaz *metaadatot*. A metaadatot hivatalosan arra használjuk, hogy leírjuk egy .NET-szerelvény minden egyes jellemzőjét, beleértve a belső típusokat (a tagjaikat, ősosztályaikat és így tovább), a szerelvény verzióját, valamint az opcionális szerelvényszintű információkat (erős név, kultúra stb.).

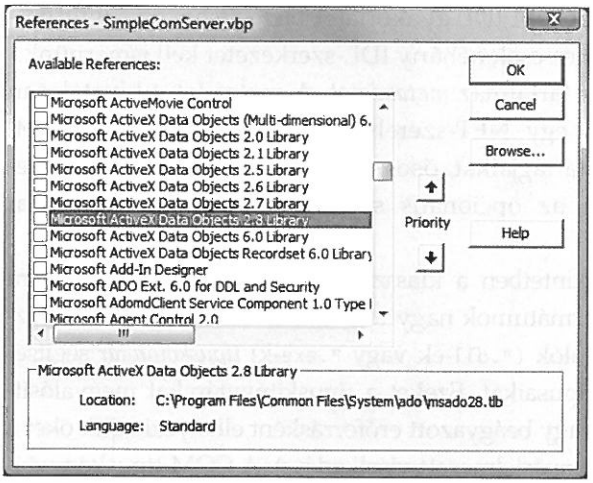
A .NET-metaadat sok tekintetben a klasszikus COM-kiszolgálók leírására szolgáló korábbi metaadat-formátumok nagy testvérének tekintethető. A klasszikus ActiveX COM-kiszolgálók (`*.dll`-ek vagy `*.exe`-k) *típuskönyvtár* segítségével dokumentálják belső típusaikat. Ezeket a típuskönyvtárakat megvalósíthatjuk önálló `*.tlb` fájlként, vagy beágyazott erőforrásként elhelyezhetjük őket a COM-kiszolgálón (ez a VB6 alapértelmezett viselkedése). A COM-típuskönyvtárak általában IDL (Interface Definition Language – interfészleíró nyelv) metaadatnyelven készülnek a speciális `midl.exe` fordítóban (Microsoft IDL-fordító).

A VB6 kiválóan elrejti szem elől a típuskönyvtárakat és az IDL-t. Igazából számos sikeres VB COM-programozó teljesen figyelmen kívül hagyja az IDL szintaxisát. Mindenesetre, ha ActiveX projekt `workspace` típusokat fordítunk, a VB automatikusan létrehozza és beágyazza a típuskönyvtárat a fizikai `*.dll` vagy `*.exe` COM-kiszolgálón. Ezenkívül a VB6 biztosítja, hogy a rendszer a típuskönyvtárat a rendszerleíró adatbázis speciális részében automatikusan regisztrálja: HKEY_CLASSES_ROOT\TypeLib (lásd az A.8. ábrát).



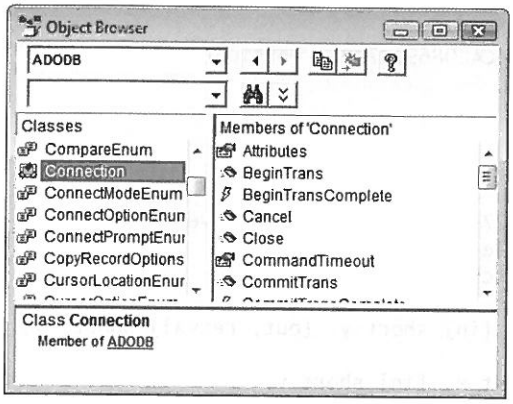
A.8. ábra: *A HKCR\TypeLib felsorolja az adott gépen található összes regisztrált típuskönyvtárat*

A típuskönyvtárakra több integrált fejlesztői környezet hivatkozik folyamatosan. Ha például a VB6-ban a Project > References menüpontot választjuk, az IDE a HKCR\TypeLib segítségével az A.9. ábrán látható módon meghatározza az összes regisztrált típuskönyvtárat.



A.9. ábra: *COM típusadat-információinak hivatkozása a VB6-ból*

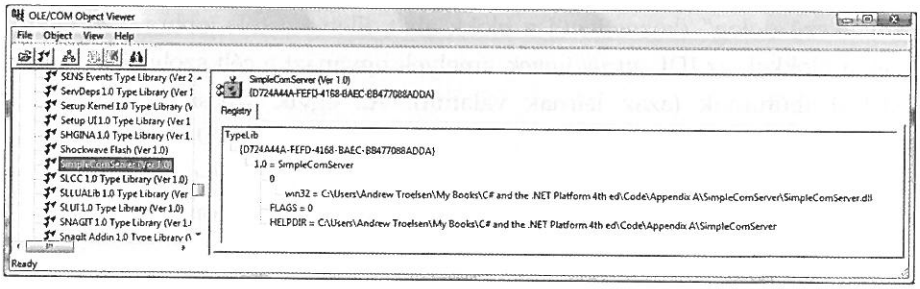
Ugyanígy, ha megnyitjuk a VB6 Objektum böngészőjét, a VB6 IDE beolvassa a típusadatot, és megjeleníti a COM-kiszolgáló tartalmát egy barátságos GUI segítségével (lásd az A.10. ábrát).



A.10. ábra: *Típuskönyvtárak megtekintése a VB6 objektum böngészőjével*

A VB COM-kiszolgálónkhoz generált IDL

Noha a VB6 objektumböngészője megmutatja egy típuskönyvtár összes típusát, az OLE View segédprogram (`oleview.exe`) lehetővé teszi a megfelelő típuskönyvtár létrehozásához használt, alapul szolgáló IDL-szintaxis megtekintését. Ha telepítettük a VB6-ot, akkor az OLE View eszközt a Start > All Programs > Microsoft Visual Studio 6.0 > Microsoft Visual Studio 6.0 Tools útvonalon nyithatjuk meg, és a SimpleComServer szervert a Treeview vezérlő Type Libraries csomópontja alatt találhatjuk (lásd az A.11. ábrát).



A.11. ábra: *A SimpleComServer keresése az OLE/COM objektummegjelenítő segítségével*

Ha kétszer kattintunk a típuskönyvtár ikonjára, megnyílik egy új ablak, amelyben látható a VB6-fordító által létrehozott típuskönyvtárt alkotó összes IDL-token. A releváns – és kissé formázott – IDL a következő (az [uuid] értékek különbözőek lesznek):

```
[uuid(8AED93CB-7832-4699-A2FC-CAE08693E720), version(1.0)]
library SimpleComServer
{
  importlib("stdole2.tlb");
  interface _ComCalc;

  [odl, uuid(5844CD28-2075-4E77-B619-9B65AA0761A3), version(1.0),
    hidden, dual, nonextensible, oleautomation]
  interface _ComCalc : IDispatch {
    [id(0x60030000)]
    HRESULT Add([in] short x, [in] short y, [out, retval] short* );
    [id(0x60030001)]
    HRESULT Subtract([in] short x, [in] short y,
      [out, retval] short* );
  };

  [uuid(012B1485-6834-47FF-8E53-3090FE85050C), version(1.0)]
  coclass ComCalc {
    [default] interface _ComCalc;
  };
};
```

Az IDL-attribútumok

Az IDL elemzéséhez először figyeljük meg, hogy az IDL-szintaxis szögletes zárójelek ([...]) közé foglalt kódblokkokat tartalmaz. A zárójeleken belül vesszővel elválasztott IDL-kulcsszavakat találunk, amelyek egyértelművé teszik, hogy mi a „következő dolog" (közvetlenül a blokk alatt, illetve a tőle jobbra lévő elem). Ezek a blokkok az IDL-*attribútumok*, amelyek ugyanazt a célt szolgálják, mint a .NET-attribútumok (azaz leírnak valamit). Az egyik kulcsfontosságú IDL-attribútum az [uuid], amellyel adott COM-típushoz GUID-t rendelhetünk. A COM-ban mindenhez tartozik GUID (interfészek, COM-osztályok, típuskönyvtárak és így tovább), ennek célja az adott elem egyedi azonosítása.

Az IDL-könyvtár-utasítás

A legfelső szint a COM-*könyvtár-utasítás*, amelyet az IDL `library` kulcsszavával jelölünk. A könyvtárutasítás magában foglal minden egyes interfészt és COM-osztályt, valamint az összes felsorolást és felhasználó által definiált típust. A `SimpleComServer` esetében a típuskönyvtár pontosan egy COM-osztályt listáz, a `ComCalcot`, amelyet a `coclass` (azaz COM-osztály) kulcsszóval jelöltünk meg.

A [default] interfész szerepe

A COM szabályai szerint a COM-kliens COM-osztállyal folytatott kommunikációjának az egyetlen lehetséges útja az interfészreferencia (nem objektumreferencia) használata. Ha már hoztunk létre C++-alapú COM-ügyfélprogramokat, akkor tisztában vagyunk egy adott interfész lekérdezésének folyamatával, a szükségtelenné váló interfész felszabadításával stb. Amikor azonban VB6-ban készítünk COM-klienseket, a COM-osztályhoz automatikusan egy *alapértelmezett interfészt* kapunk.

VB6 COM-kiszolgálók esetében egy `*.cls` fájl nyilvános tagjait (például az `Add()` függvényt) a rendszer a COM-osztály „alapértelmezett interfészére" helyezi. A `ComCalc` osztálydefinícióját megvizsgálva kiderül, hogy az alapértelmezett interfésze neve `_ComCalc`:

```
[uuid(012B1485-6834-47FF-8E53-3090FE85050C), version(1.0)]
coclass ComCalc {
  [default] interface _ComCalc;
};
```

A VB6 által a háttérben létrehozott alapértelmezett interfész neve mindig `_AzOsztályNeve` (az aláhúzás névadási konvenció a rejtett interfészek jelölésére). Így, ha van egy `Car` osztályunk, az alapértelmezett interfész `_Car`, a `DataConnector` osztály alapértelmezett interfésze `_DataConnector` és így tovább.

A VB6 alatt az alapértelmezett interfész egyáltalán nem látható. Ha azonban megírjuk a következő VB6 kódot:

```
' VB 6.0 COM-kliens kód.
Dim c As ComCalc
Set c = New ComCalc ' A [default] _ComCalc interfész automatikusan
                    ' vissza adódik!
```

a VB automatikusan lekérdezi az alapértelmezett interfészt az objektumtól (ahogy a típuskönyvtár meghatározza), és elküldi a kliensnek. Mivel a VB mindig visszaadja a COM-osztály alapértelmezett interfészét, úgy tűnik, mintha igazi objektumreferenciánk lenne. Ez azonban csak egy része a VB6 által biztosított szintaktikai egyszerűsítésnek. A COM-ban nem létezik közvetlen objektumreferencia. Interfészreferencia viszont mindig van (még akkor is, ha ez a referencia az alapértelmezett).

Az IDispatch szerepe

Az alapértelmezett `_ComCalc` IDL-leírását megvizsgálva, láthatjuk, hogy az interfész az `IDispatch` COM-interfészből származik. Ez az interfész teszi lehetővé, hogy klasszikus ASP-ből kommunikálhassunk a COM-objektumokkal akár a weben, akár bárhol másutt, ahol késői kötésre van szükség.

IDL-paraméterattribútumok

Az IDL-lel kapcsolatosan még tudni kell, hogy a VB6 paraméterei hogyan jelennek meg a háttérben. A VB6 alatt az összes paramétert referenciaként adjuk át, hacsak nem használjuk explicit módon a `ByVal` kulcsszót, amelyet az IDL `[in]` attribútumával ábrázolunk. Egy függvény visszatérési értékét az `[out, retval]` attribútumokkal jelöljük. Ezért az alábbi VB6 függvény:

```
' VB függvény
Public Function Add(ByVal x as Integer, ByVal y as Integer) as
  Integer
    Add = x + y
End Function
```

IDL-ben a következőképpen nézne ki:

```
HRESULT Add([in] short* x, [in] short* y, [out, retval] short*);
```

Másrészről, ha nem jelölünk meg egy paramétert a `ByVal` kulcsszóval, akkor a VB6 automatikusan a `ByRef` kulcsszót feltételezi:

```
' Ezek a paraméterek átadják a ByRef-et VB6 alatt!
Public Function Subtract(x As Integer, y As Integer) As Integer
  Subtract = x - y
End Function
```

Az IDL-ben a ByRef paramétereket az [in, out] attribútumok jelölik:

```
HRESULT Subtract([in, out] short x, [in, out] short y,
  [out, retval] short*);
```

Típuskönyvtár használata együttműködési szerelvény készítéséhez

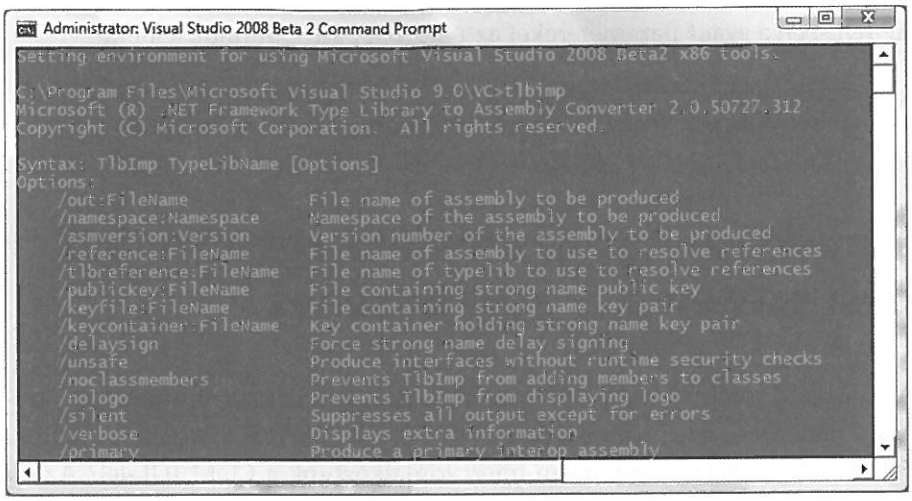
A VB6 fordító a biztonság kedvéért számos egyéb IDL-attribútumot is létrehoz a háttérben, amelyekből a megfelelő helyeken további részleteket látunk. Felmerülhet a kérdés: pontosan miért foglalkoztunk a COM IDL-lel? Az ok egyszerű: ha a Visual Studio 2008 segítségével adunk referenciát a COM-kiszolgálóhoz, az IDE beolvassa a típuskönyvtárat, hogy felépítse a megfelelő együttműködési szerelvényt. Bár a VS 2008 tökéletesen megfelelő az együttműködő szerelvények generálásához, az Add Reference párbeszédpanel az alapértelmezett szabálykészlethez igazodik az együttműködő szerelvény létrehozásakor, és nincs lehetőségünk a felépítés finomhangolására.

Ha nagyobb rugalmasságra van szükségünk, akkor a parancssorban is generálhatunk együttműködési szerelvényt a `tlbimp.exe` nevű (típuskönyvtár-importáló eszköz) .NET-segédprogram segítségével. Egyebek között a `tlbimp.exe` lehetővé teszi a típusokat és a kimeneti fájl nevét tartalmazó .NET-névtér nevének a kezelését. Továbbá, ha erős nevet szeretnénk hozzárendelni az együttműködő szerelvényhez, hogy telepíteni tudjuk a globális szerelvénytárba (GAC), a `tlbimp.exe` biztosítja számunkra a `/keyfile` kapcsolót az `*.snk` fájl meghatározásához (az erős nevekkel kapcsolatban lásd az előző kötet 15. fejezetét). Az összes opció megtekintéséhez írjuk be a *tlbimp* szót a Visual Studio 2008 parancssorába, majd nyomjuk meg az Entert (lásd az A.12. ábrát).

Bár az eszköz számos opciót felkínál, a következő paranccsal lehet erős névvel rendelkező együttműködési szerelvényt készíteni `CalcInteropAsm.dll` névvel (feltételezzük, hogy van egy `myKeyPair.snk` nevű `*.snk` fájlunk):

```
tlbimp SimpleComServer.dll /keyfile:myKeyPair.snk
  /out:CalcInteropAsm.dll
```

Ha a Visual Studio 2008 által létrehozott együttműködési szerelvény megfelelő, akkor nem kell közvetlenül használnunk a `tlbimp.exe` segédprogramot.



A.12. ábra: *A tlbimp.exe opciói*

Késői kötés a CoCalc coclasshoz

Ha létrehoztuk az együttműködési szerelvényt, a .NET-alkalmazásunk már használhatja a típusait a korai kötési vagy késői kötési technikák valamelyikével. Azt már láttuk, hogy hogyan lehet COM-típust létrehozni a korai kötés alkalmazásával (a C# `new` kulcsszava révén), most nézzük meg a COM-objektumok aktiválását a késői kötés által.

A `System.Reflection` névtér lehetőséget biztosít arra, hogy programozottan futásidőben vizsgáljuk meg az adott szerelvény típusait (lásd az előző kötet 16. fejezetét). A COM-ban ugyanezt a működést a szabványos interfészek (pl. `ITypeLib`, `ITypeInfo` és így tovább) alkalmazása támogatja. Ha a kliens futási időben (és nem fordítási időben) egy taghoz kötést hoz létre, akkor ezt a „késői" kötés létrehozásának nevezzük.

Voltaképpen mindig célszerű a C# `new` kulcsszavát és ezzel a korai kötési technikát alkalmazni. Természetesen vannak olyan esetek, amikor a késői kötést kell használnunk. Néhány korábbi COM-kiszolgálót például úgy is felépíthettünk, hogy azok semmilyen típusinformációt nem biztosítottak. Ilyen helyzetben nyilvánvaló, hogy nem futtathatjuk azonnal a `tlbimp.exe` segédprogramot. Mivel ez ritkán fordul elő, a klasszikus COM-típusokhoz a .NET reflexiós szolgáltatásának segítségével is hozzáférhetünk.

A késői kötés folyamata azzal kezdődik, hogy a kliens megszerzi az IDispatch interfészt egy adott coclasstól. Ez a COM-interfész összesen négy metódust definiál, ezek közül jelenleg kettővel kell foglalkoznunk. Az egyik a GetIDsOfNames(). Ez a metódus lehetővé teszi, hogy a hívó késői kötést alkalmazzon: megszerzi a hívni kívánt metódus azonosítására szolgáló numerikus értéket (ennek neve dispatch ID vagy DISPID).

A COM IDL-ben egy taghoz az [id] attribútummal rendelhetjük hozzá az DISPID-ját. Ha megvizsgáljuk a VB6 által generált IDL-kódot (az OLE View eszközzel), látható, hogy az Add() metódus DISPID-ja a következőképpen kapott értéket:

```
[id(0x60030000)]
  HRESULT Add( [in] short x, [in] short y, [out, retval] short* );
```

Ez az az érték, amelyet a GetIDsOfNAmes() metódus visszaadott a késői kötésű kliensnek. Amint a kliens megszerzi ezt az értéket, meghívja a következő, érdeklődésre számot tartó metódust, az Invoke()-ot. Az IDispatch ezen metódusa számos argumentumot felvesz, közülük az egyik a GetIDsOfNames() használatával megszerzett DISPID. Továbbá az Invoke() felveszi COM VARIANT típusok egy tömbjét is, azt, amelyik a függvénynek átadott paramétereket ábrázolja. Az Add() metódus esetén ez tömb két (valamilyen értékű) shortot tartalmaz. Az Invoke() utolsó argumentuma egy másik VARIANT, amely a metódushívás visszaadott értékét tárolja (ismét egy shortot).

Noha a késői kötést alkalmazó .NET-kliensek közvetlenül nem használják az IDispatch interfészt, a System.Reflection névtérrel ugyanez az általános működés valósítható meg. Ennek bemutatásához készítsünk egy másik C#-klienst, amely késői kötést használ az Add() logika kiváltásához. Ez az alkalmazás *nem* készít semmilyen referenciát a szerelvényhez, ezért nincs szükség a tlbimp.exe segédprogram használatára.

```
// Mindenképpen használjuk a System.Reflection névteret.
static void Main(string[] args)
{
  Console.WriteLine("***** The Late Bound .NET Client *****");

  // Először megszerezzük az IDispatch referenciát a coclasstól.
  Type calcObj =
    Type.GetTypeFromProgID("SimpleCOMServer.ComCalc");
  object calcDisp = Activator.CreateInstance(calcObj);

  // Készítsük el az argumentumok tömbjét.
 object[] addArgs = { 100, 24 };
```

```
    // Hívjuk az Add() metódust, majd kérjünk összegzést.
    object sum = null;
    sum = calcObj.InvokeMember("Add", BindingFlags.InvokeMethod,
    null, calcDisp, addArgs);

    // Az eredmények megjelenítése.
    Console.WriteLine("Late bound adding: 100 + 24 is: {0}", sum);
    Console.ReadLine();
  }
```

Forráskód A CSharpComLateBinding kódfájlokat a forráskódkönyvtár A Függelékének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Bonyolultabb COM-kiszolgáló készítése

A következőkben olyan VB6 ActiveX szervert készítünk, amely bonyolultabb COM programozási módszereket használ. Hozzunk létre egy új ActiveX *.dll workspace-t Vb6ComCarServer névvel. Kezdeti osztályunkat nevezzük át CoCarra, ez a következő megvalósítással rendelkezik:

```
Option Explicit

' Egy COM-felsorolás.
Enum CarType
  Viper
  Colt
  BMW
End Enum

' Egy COM-esemény.
Public Event BlewUp()

' Tagváltozók.
Private currSp As Integer
Private maxSp As Integer
Private Make As CarType

' Ne feledjük! Minden nyilvános tagot
' az alapértelmezett interfész biztosít.
Public Property Get CurrentSpeed() As Integer
  CurrentSpeed = currSp
End Property

Public Property Get CarMake() As CarType
  CarMake = Make
End Property
```

```
Public Sub SpeedUp()
  currSp = currSp + 10
  If currSp >= maxSp Then
    RaiseEvent BlewUp
' Esemény kiváltása, ha túlerőltetjük a motort.
  End If
End Sub

Private Sub Class_Initialize()
  MsgBox "Init COM car"
End Sub

Public Sub Create(ByVal max As Integer, _
  ByVal cur As Integer, ByVal t As CarType)
  maxSp = max
  currSp = cur
  Make = t
End Sub
```

Ez egy egyszerű COM-osztály, amely a könyvben használt C# `Car` osztályának működését imitálja. Az egyetlen érdekes pont a `Create()` szubrutin, amely lehetővé teszi, hogy hívó átadja a `Car` objektumot leíró állapotadatokat.

Még egy COM-interfész támogatása

Mindezek után szúrjunk be egy új `*.cls` fájlt, amely a következő `IDriverInfo` interfészt definiálja:

```
Option Explicit

' A sofőrnek van neve.
Public Property Let DriverName(ByVal s As String)
End Property
Public Property Get DriverName() As String
End Property
```

Ha már készítettünk több interfészt támogató COM-objektumokat, tudjuk, hogy a VB6 biztosítja számunkra az `Implements` kulcsszót. Ha megadjuk az adott COM-osztály által megvalósított interfészeket, akkor használhatjuk a VB6-kódablakot a metódus csonkjainak elkészítéséhez. Tételezzük fel, hogy hozzáadtunk egy privát sztringváltozót (`driverName`) a `CoCar` osztálytípushoz, és az alábbiak szerint megvalósítottuk az `IDriverInfo` interfészt:

```
' Megvalósított interfészek
' [General][Declarations]
Implements IDriverInfo
...
' ***** IDriverInfo implementációja ***** '
Private Property Let IDriverInfo_DriverName(ByVal RHS As String)
  driverName = RHS
End Property

Private Property Get IDriverInfo_DriverName() As String
  IDriverInfo_driverName = driverName
End Property
```

Az interfészmegvalósítás vizsgálatának befejezéséhez állítsuk az `IDriverInfo` `Instancing` tulajdonságát `PublicNotCreatable` értékre (annak érdekében, hogy kívülről ne lehessen lefoglalni interfésztípusokat).

Belső objektumok feltárása

A VB6 (és maga a COM) alatt nem áll rendelkezésünkre a kényelmes klasszikus megvalósítási származtatás. Ehelyett kénytelenek vagyunk a tartalmazás/delegálás modellt (a „van egy" kapcsolatot) használni. Tesztelési célból adjunk egy utolsó `*.cls` fájlt az aktuális Engine nevű VB6-projektünkhöz, és állítsuk az `Instancing` tulajdonságát `PublicNotCreatable` értékre (mivel szeretnénk azt megakadályozni, hogy a felhasználó közvetlenül létrehozhasson egy `Engine` objektumot).

Az `Engine` alapértelmezett nyilvános interfésze rövid és egyszerű. Definiáljunk egy függvényt, amely sztringtömböt ad vissza, és ez képviseli a motor minden egyes hengerének becenevét (igaz, nem szokás barátságos nevet adni a hengereknek, de hát...):

```
Option Explicit

Public Function GetCylinders() As String()
  Dim c(3) As String
  c(0) = "Grimey"
  c(1) = "Thumper"
  c(2) = "Oily"
  c(3) = "Crusher"
  GetCylinders = c
End Function
```

Végül adjuk a `GetEngine()` nevű metódust a `CoCar` alapértelmezett interfészéhez, amely visszaadja a tárolt `Engine` egy példányát (ehhez létre kell hoznunk az `Engine` típusú, `eng` névvel rendelkező privát tagváltozót):

```
' Visszaadjuk az Engine-t a külvilágnak.
Public Function GetEngine() As Engine
  Set GetEngine = eng
End Function
```

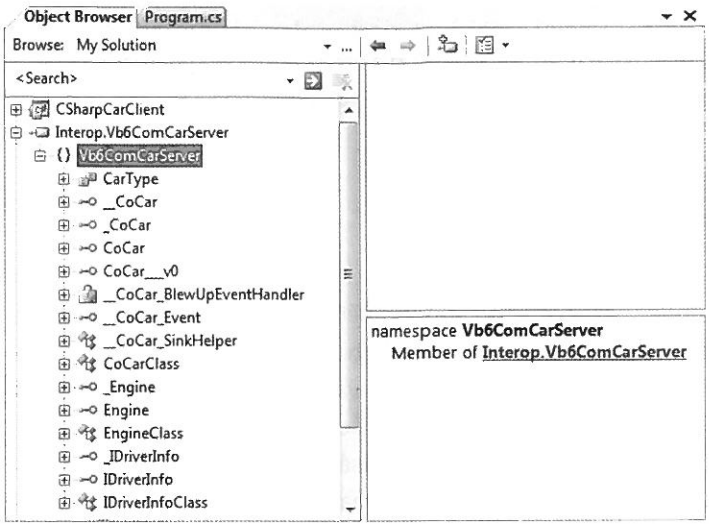
Van tehát egy ActiveX szerverünk, amely két interfészt támogató COM-osztályt foglal magában. Emellett visszaadhatunk egy belső COM-típust a `CoCar [default]` interfészének használatával, valamint együttműködhetünk néhány alapvető programozási szerkezettel (felsorolt típusokkal és COM-tömbökkel). Folytassuk a kiszolgáló lefordításával (és ha ezzel készen vagyunk, állítsuk be a bináris kompatibilitást), majd zárjuk le az aktuális VB6-munkaterületet.

Forráskód A Vb6ComCarServer kódfájlokat a forráskódkönyvtár A Függelék alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Az együttműködési szerelvény

Ahelyett, hogy a `tlbimp.exe` segédprogramot használnánk az együttműködési szerelvény generálásához, hozzunk létre új konzolalkalmazást (CSharpCarClient névvel) a Visual Studio 2008 használatával, és az Add Reference párbeszédpanel COM-lapján állítsunk be referenciát a `Vb6ComCarServer.dll` szerelvényre. Vizsgáljuk meg az együttműködési szerelvényt a VS 2008 Objektum böngészőjében (lásd az A.13. ábrát).

Megint lett néhány `Class` utótaggal rendelkező és aláhúzásjeles interfésztípusunk, valamint néhány új elemünk, amelyeknek a neve azt sugallja, hogy minden bizonnyal a COM–.NET-eseményértesítések (nevezetesen a `CoCar_Event`, `CoCar_SinkHelper` és `CoCarBlewUpEventHandler`) kezelésére szolgálnak. Amikor egy COM-objektum COM-eseményeket tár fel, az együttműködési szerelvény további CIL-kódot tartalmaz, amelyet a CLR arra használ, hogy leképezze a COM-eseményeket .NET-es eseményekre (lásd ezt működés közben később).



A.13. ábra: *Az Interop.VbComCarServer.dll szerelvény*

Saját C#-kliensalkalmazás készítése

Mivel a CLR futási időben automatikusan létrehozza a szükséges RCW-t, a C#-alkalmazásunk `CoCar`, `CarType`, `Engine` és `IDriveInfo` típusai közvetlenül programozhatók, mintha mindegyiküket felügyelt kód segítségével implementáltuk volna. A teljes megvalósítás az elemzéssel együtt a következő:

```
// Mindenképpen importáljuk a Vb6ComCarServer névteret.
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** CoCar Client App *****");

    // Korai kötés használatával létrehozzuk a COM-osztályt.
    CoCar myCar = new CoCar();

    // A BlewUp esemény kezelése.
    myCar.BlewUp += new __CoCar_BlewUpEventHandler(myCar_BlewUp);

    // A Create() metódus meghívása.
    myCar.Create(50, 10, CarType.BMW);

    // Sofőr nevének beállítása.
    IDriverInfo itf = (IDriverInfo)myCar;
    itf.DriverName = "Fred";
    Console.WriteLine("Drive is named: {0}", itf.DriverName);
```

```
      // Autó típusának kiírása.
      Console.WriteLine("Your car is a {0}.", myCar.CarMake);
      Console.WriteLine();

      // Az Engine beolvasása és a hengerek nevének kiírása.
      Engine eng = myCar.GetEngine();
      Console.WriteLine("Your Cylinders are named:");
      string[] names = (string[])eng.GetCylinders();
      foreach (string s in names)
      {
        Console.WriteLine(s);
      }
      Console.WriteLine();

      // Az autó gyorsítása az esemény kiváltásához.
      for (int i = 0; i < 5; i++)
      {
        myCar.SpeedUp();
      }
    }

    // A BlewUp esemény kezelője.
    static void myCar_BlewUp()
    {
      Console.WriteLine("Your car is toast!");
    }
}
```

A `GetCylinders()` hívásakor a visszatérési értéket sztringtömbbé kasztoltuk. Ennek oka az, hogy a COM-tömböket (általában) SAFEARRAY COM-típus képviseli (mindig ez a helyzet, ha a VB6 használatával készítünk COM-alkalmazásokat). Az RCW `System.Array` objektumokra képezi le a SAFEARRAY típusokat ahelyett, hogy automatikusan egy C#-szintaxissal ábrázolt tömbbe képezné le őket. Ezáltal, ha az `Array` objektumot egy `string[]` típusba kasztoljuk, jóval könnyebben tudjuk feldolgozni a tömböt.

Együttműködés a CoCar típussal

Amikor létrehoztuk a `CoCar` típust a VB6-ban, a VB6-fordító által automatikusan generált alapértelmezett interfészen (`_CoCar`) kívül definiáltuk és implementáltuk az `IDriverInfo` egyedi COM-interfészt is. Amikor a `Main()` metódusunk létrehozza a `CoCar` egy példányát, közvetlen hozzáféréssel csak `_CoCar` interfész tagjaihoz rendelkezünk, ez pedig a COM-osztály nyilvános tagjaiból áll:

```
// Most valóban a [default] interfésszel dolgozunk.
myCar.Create(50, 10, CarType.BMW);
```

Így ahhoz, hogy meghívhassuk az `IDriverInfo` interfész `DriverName` tulajdonságát, az alábbiak szerint a `CoCar` objektumot explicit módon kasztolnunk kell az `IDriverInfo` interfészre:

```
// Sofőr nevének beállítása.
IDriverInfo itf = (IDriverInfo)myCar;
itf.DriverName = "Fred";
Console.WriteLine("Drive is named: {0}", itf.DriverName);
```

Amikor egy típuskönyvtárat együttműködő szerelvénnyé konvertálunk, az olyan `Class` utótaggal rendelkező típusokat fog tartalmazni, amelyek feltárják az összes interfész összes tagját. Így egyszerűsíthetjük a programozást, ha `CoCar` objektum helyett `CoCarClass` objektumot hozunk létre és használunk. Nézzük meg például a következő metódust, amely a `CoCar` alapértelmezett interfészének tagjait, valamint az `IDriverInfo` tagjait alkalmazza:

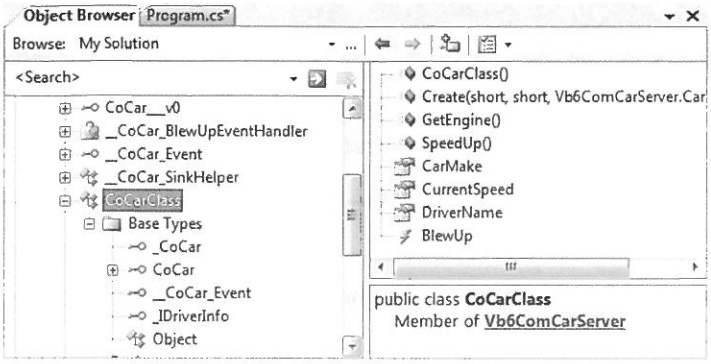
```
static void UseCar()
{
  // A Class utótaggal rendelkező típusok megmutatják az összes
  // interfész összes tagját.
  CoCarClass c = new CoCarClass();

  // Ez a tulajdonság az IDriverInfo tagja.
  c.DriverName = "Mary";

  // Ez a metódus a _CoCar tagja.
  c.SpeedUp();
}
```

Ahhoz, hogy választ kapjunk arra a kérdésre, hogy ez az egy típus pontosan hogyan mutatja meg az összes megvalósított interfész tagjait, a Visual Studio 2008 Objektum böngészőjében nézzük át a `CoCarClass` implementált interfészeinek és ősosztályainak listáját (lásd az A.14. ábrát).

Ez a típus megvalósítja a `_CoCar` és `_IDriverInfo` interfészeket, és „normális" nyilvános tagként teszi hozzáférhetővé őket.



A.14. ábra: *A CoCarClass összetétele*

COM-események elfogása

Az előző kötet 11. fejezetében megismerkedhettünk a .NET eseménymodelljével. Ez az architektúra azon alapul, hogy a programlogika folyását az alkalmazás egyik részéből a másikba delegáljuk. A kérések továbbításáért felelős entitás a `System.MulticastDelegate` egyik leszármazott típusa, amelyet közvetetten a `delegate` kulcsszó használatával hozunk létre a C#-ban.

Amikor a `tlbimp.exe` segédprogram eseménydefiníciókat talál a COM-kiszolgálók típuskönyvtárában, válaszként olyan felügyelt típusokat hoz létre, amelyek beburkolják az alacsony szintű COM kapcsolódási pontot. Ezeknek a típusoknak a használatával azt a látszatot kelthetjük, mintha hozzáadtunk volna egy tagot a `System.MulticastDelegate` belső metóduslistájához. A háttérben természetesen a proxy a felügyelt megfelelőjére képezi le a bejövő COM-eseményt. Az A.3. táblázat röviden ismerteti ezeket a típusokat.

Generált típus (a _CarEvents [source] interfész alapján)	Jelentés
`_CoCar_Event`	Ez a felügyelt interfész definiálja az `add` és a `remove` tagokat egy metódus hozzáadásához (vagy eltávolításához) a `System.MulticastDelegate` láncolt listáján.
`_CoCar_BlewUpEventHandler`	Ez egy felügyelt metódusreferencia (amely a `System.MulticastDelegate` osztályból származik).

Generált típus (a _CarEvents [source] interfész alapján)	Jelentés
_CoCar_SinkHelper	Ez a generált osztály valósítja meg a kimenő interfészt egy .NET-alapú nyelőobjektumban.

A.3. táblázat: *COM-esemény segédtípusok*

A bejövő COM-eseményeket ugyanúgy kezelhetjük, mint ahogyan a metódusreferenciám alapuló .NET-eseményeket:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** CoCar Client App *****");
    CoCar myCar = new CoCar();

    // A BlewUp esemény kezelése.
    myCar.BlewUp += new __CoCar_BlewUpEventHandler(myCar_BlewUp);
    ...
  }

  // A BlewUp esemény kezelője.
  static void myCar_BlewUp()
  {
    Console.WriteLine("Your car is toast!");
  }
}
```

A C#-kódunk az összes eseményközpontú jelölést (névtelen metódusok, metóduscsoport átalakítás, lambda kifejezések stb.) alkalmazhatja a COM-objektumok eseményeinek elfogása során.

Forráskód A CSharpCarClient kódfájlokat a forráskódkönyvtár A Függelék alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezető XIV. oldalát.

A fent megvizsgált módszerek *bármilyen* COM-kiszolgáló esetében működnek. Ezt azért fontos megjegyezni, mert előfordulhat, hogy több COM-kiszolgálót soha senki nem fog natív .NET-alkalmazásként újraírni. A Microsoft Outlook objektummodellje például jelenleg COM-könyvtárként jelenik meg. Ezért, ha olyan .NET-programot kell készítenünk, amely ezzel a termékkel dolgozik együtt, akkor az együttműködési réteg alkalmazása (pillanatnyilag) kötelező.

A COM és a .NET együttműködési képessége

A továbbiakban azt a folyamatot vizsgáljuk meg, amelyben a COM-alkalmazások .NET-típussal kommunikálnak. Az együttműködésnek ez az „iránya" lehetővé teszi, hogy a korábbi COM-kódalapok (például egy már létező VB6-projekt) kihasználják az újabb .NET-szerelvényekben rejlő funkcionalitást. Ez a helyzet ritkábban fordul elő, mint a .NET–COM-együttműködés, ám érdemes erre is figyelni.

Ahhoz, hogy a COM-alkalmazás .NET-típust használhasson, valahogy át kell vernünk a COM-programot, hogy azt higgye, a felügyelt .NET-típus valójában *nem felügyelt* típus. Lényegében lehetővé kell tenni a COM-alkalmazás számára, hogy a COM-architektúra által megkövetelt funkcionalitás segítségével működhessen együtt a .NET-típussal. A COM-típusnak például meg kell szereznie az új interfészeket a `QueryInterface()` hívásokon keresztül, szimulálnia kell a nem felügyelt memóriakezelést az `AddRef()` és `Release()` metódusok segítségével, használnia kellene a COM kapcsolódásipont-protokollját, és így tovább.

A COM-kliens rászedésén túl a COM és a .NET együttműködési képesség magába foglalja a COM-futtatómotor átverését is. A COM-kiszolgálót a CLR helyett a COM-futtatórendszerrel aktiváljuk. Ennek érdekében a COM-futtatórendszernek több információdarabot meg kell keresnie a rendszerleíró adatbázisban (programazonosítókat, CLS-azonosítókat, interfészazonosítókat és így tovább). A probléma természetesen az, hogy a .NET-szerelvényeket először is nem a rendszerleíró adatbázisban regisztráljuk.

Ennek köszönhetően a következő lépéseket kell végrehajtanunk, ha a .NET-szerelvényeinket szeretnénk elérhetővé tenni a COM-kliensek számára:

1. Regisztráljuk a .NET-szerelvényünket a rendszerleíró adatbázisban, hogy a COM-futtatórendszer megtalálhassa.
2. Hozzunk létre COM-típuskönyvtárfájlt (`*.tlb`) (a .NET-metaadat alapján), és tegyük lehetővé a COM-kliens számára, hogy együttműködhessen a nyilvános típusokkal.
3. Telepítsük a szerelvényt ugyanabba a könyvtárba, mint ahol a COM-ügyfélprogram található, vagy (inkább) telepítsük a globális szerelvénytárba.

Ezeket a lépéseket végrehajthatjuk a Visual Studio 2008-ban vagy a parancssorban azokkal a különböző eszközökkel, amelyeket a .NET Framework 3.5 SDK foglal magában.

A System.Runtime.InteropServices attribútumai

A fenti lépések végrehajtása mellett általában a C#-típusainkat különböző olyan .NET-attribútumokkal kell ellátnunk, amelyeket a `System.Runtime.InteropServices` névtér definiál. Végeredményben ezek az attribútumok szabályozzák azt, hogy a COM-típuskönyvtár hogyan készüljön el, egyúttal meghatározzák azt is, hogy a COM-alkalmazás hogyan legyen képes együtt dolgozni a felügyelt típusainkkal. Az A.4. táblázat bemutat néhány (de nem minden) attribútumot, amelyeket a létrehozott COM-típuskönyvtár vezérlésére használhatunk.

.NET együttműködési attribútum	Jelentés
`[ClassInterface]`	Ezzel az attribútummal hozhatjuk létre egy .NET-osztálytípus alapértelmezett COM-interfészét.
`[ComClass]`	Ez az attribútum a `[ClassicInterface]` attribútumhoz hasonlít, azt leszámítva, hogy ezzel előállíthatjuk a típuskönyvtárban lévő COM-típusok osztályazonosítójához (CLSID) és interfészazonosítóihoz használt GUID-azonosítókat.
`[DispId]`	Ezt az attribútumot egy taghoz rendelt DISPID-értékek bedrótozására alkalmazhatjuk késői kötés céljából.
`[Guid ]`	Ez az attribútum egy GUID-érték bedrótozására szolgál a COM-típuskönyvtárban.
`[In]`	Ez az attribútum a COM IDL-ben bemeneti paraméterként biztosít egy tagparamétert.
`[InterfaceType]`	Ez az attribútum szabályozza azt, hogy egy .NET-interfész hogyan jelenjen meg a COM számára (csak `IDispatch`, kettős vagy csak `IUnknown` legyen).
`[Out]`	Ez az attribútum a COM IDL-ben kimeneti paraméterként biztosít egy tagparamétert.

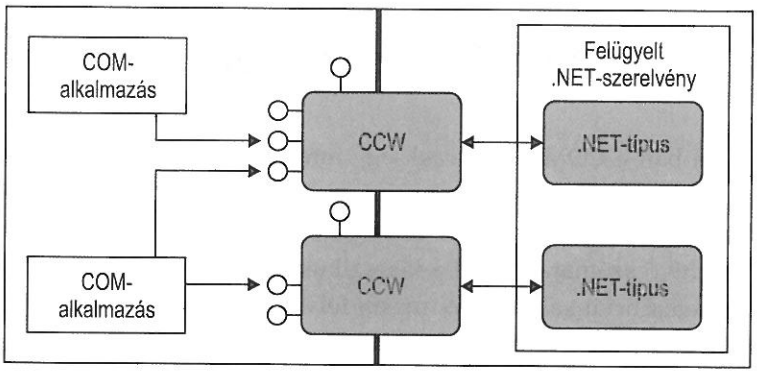
A.4. táblázat: *A System.Runtime.InteropServices legfontosabb tagjai*

Egyszerű COM–.NET együttműködési helyzetekben nem szükséges tucatnyi attribútummal ellátni a .NET-kódot annak érdekében, hogy szabályozhassuk az alapul szolgáló COM-típuskönyvtár definiálását. Ha azonban nagyon specifikusnak kell lennünk abban, hogy a .NET-típusok hogyan jelenjenek meg a COM számára, akkor célszerű minél többet tudni a COM IDL-attribútumokról, hiszen a `System.Runtime.InteropServices` névtérben definiált attribútumok nagyjából ezeknek az IDL-kulcsszavaknak a felügyelt definíciói.

A CCW szerepe

Nézzük meg röviden azt, hogy a COM-programok pontosan hogyan működnek együtt a .NET-típusokkal a CCW (COM Callable Wrapper - COM-ból hívható burkoló) segítségével. Amikor .NET-program COM-típussal kommunikál, a CLR létrehoz egy futási időben hívható burkolót. Ehhez hasonlóan, amikor a COM-kliens hozzáfér egy .NET-típushoz, a CLR felhasznál egy proxyt, amely a COM-ból hívható burkoló névre hallgat, hogy gondoskodjon a COM–.NET-átalakításról (lásd az A.15. ábrát).

Mint minden COM-objektum, a CCW is referenciaszámlált entitás, ugyanis a COM-kliens feltételezi, hogy a CCW valódi COM-típus, és ezért tartania kell magát az `AddRef()` és a `Release()` szabályaihoz. Amikor a COM-kliens véglegesen eldobta, a CCW elengedi a referenciáját, amely a valódi .NET-típusra hivatkozik, és az ettől a pillanattól szemétgyűjtésre lesz jelölve.



A.15. ábra: *A COM-típusok a CCW segítségével kommunikálnak a .NET-típusokkal*

A CCW több COM-interfészt automatikusan megvalósít, és így még inkább az a látszat, hogy a proxy eredeti coclass. A .NET-típus által definiált egyedi interfészek készlete mellett (beleértve az *osztályinterfész* nevű entitást, ezt lásd később), a CCW támogatja az A.5. táblázatban leírt szabványos COM-viselkedéseket.

CCW által megvalósított interfész	Jelentés
`IconnectionPoint` `IConnectionPointContainer`	Ha a .NET-típus eseményeket támogat, akkor ezeket COM kapcsolódási pontok képviselik.
`IEnumVariant`	Ha a .NET-típus támogatja az `IEnumerable` interfészt, akkor a COM-kliens számára szabványos COM-felsorolásként jelenik meg.
`IerrorInfo` `ISupportErrorInfo`	Ezek az interfészek teszik lehetővé, hogy a coclassok COM-hibaobjektumokat küldjenek.
`ITypeInfo` `IProvideClassInfo`	Ezek az interfészek teszik lehetővé a COM-kliens számára, hogy szimulálja egy szerelvény COM-típusinformációinak kezelését. Valójában azonban, a COM-kliens a .NET-metaadatokkal működik együtt.

`Iunknown`
`Idispatch`
`IDispatchEx`
	Ezek a COM-interfészek a .NET-típusok késői és korai kötését támogatják.

A.5. táblázat: *A CCW támogat számos belső COM-interfészt*

A .NET-osztályinterfész szerepe

A klasszikus COM-ban a COM-kliens csak egy interfészreferencia használatával kommunikálhat a COM-objektumokkal. Ezzel szemben a .NET-típusoknak nem kell támogatniuk semmilyen interfészt, ez pedig nyilvánvalóan probléma a COM-hívó számára. Mivel a klasszikus COM-kliensek nem dolgoznak objektumreferenciákkal, a CCW másik feladata olyan *osztályinterfész* biztosítása, amely a típus nyilvános szektora által definiált tagokat képviseli. A CCW ugyanazt a megközelítést használja, mint a Visual Basic 6.0.

Osztályinterfész definiálása

Ahhoz, hogy meghatározhassunk egy osztályinterfészt a .NET-típusaink számára, alkalmaznunk kell a `[ClassInterface]` attribútumot minden olyan nyilvános osztályon, amelyet biztosítani szeretnénk a COM-nak. Ezzel azt érjük el, hogy az osztály összes nyilvános tagja megjelenik egy alapértelmezett, automatikusan generált interfészen, és ez ugyanazt az elnevezési hagyományt használja, mint a VB6 (`_AzOsztályNeve`). Az attribútum alkalmazása alapvetően tetszőleges, ám meglehetősen gyakran használjuk majd. Ha mégsem, akkor az egyetlen mód, amellyel a COM-hívó kommunikálni tud a típussal, a késői kötés használata (ez jóval kevésbé típusbiztos, és általában kisebb teljesítményt eredményez).

A `[ClassInterface]` attribútum támogat egy megnevezett tulajdonságot (`ClassInterfaceType`), amely szabályozza, hogy az alapértelmezett interfész pontosan hogyan jelenjen meg a COM-típuskönyvtárban. Az A.6. táblázat ismerteti a lehetséges beállításokat.

ClassInterfaceType tag neve	Jelentés
`AutoDispatch`	Jelzi, hogy az automatikusan generált alapértelmezett interfész csak a késői kötést támogatja, és megfelel a `[ClassInterface]` attribútum elhagyásának.
`AutoDual`	Jelzi, hogy az automatikusan generált alapértelmezett interfész „kettős interfész", így korai és késői kötés használatával is együttműködhetünk vele. Ez ugyanolyan viselkedés, mint amelyet a VB6 követ, amikor definiálja az alapértelmezett COM-interfészt.
`None`	Azt jelzi, hogy a rendszer nem készít interfészt az osztály számára. Ez akkor lehet hasznos, amikor definiáltuk a saját erősen típusos .NET-interfészeinket, amelyeket elérhetővé teszünk a COM számára, és nem szeretnénk, hogy „potyautas" interfészünk legyen.

A.6. táblázat. *A ClassInterfaceType felsorolás értékei*

A következő példában a `ClassInterfaceType.AutoDual` tulajdonsággal jelöljük meg az osztályinterfészt. Így a késői kötéssel működő kliensek, mint a VBScript, az IDispatch használatával hozzáférhetnek az `Add()` és a `Subtract()` metódusokhoz, míg a korai kötéssel működő kliensek (mint a VB6 vagy a C++) az osztályinterfészt használhatják (amelynek `VbDotNetCalc` a neve).

Saját .NET-típusok készítése

Annak bemutatásához, hogy a COM-típus hogyan kommunikálhat felügyelt kóddal, tételezzük fel, hogy létrehoztunk egy egyszerű C#-osztálykönyvtár-projektet `ComCallableDotNetServer` névvel, amely definiálja a `DotNetCalc` osztályt. Ez az osztály két egyszerű metódust definiál: az `Add()` és a `Subtract()` metódusokat. A megvalósítás logikája nagyon egyszerű, figyeljük meg azonban a `[ClassInterface]` attribútum használatát:

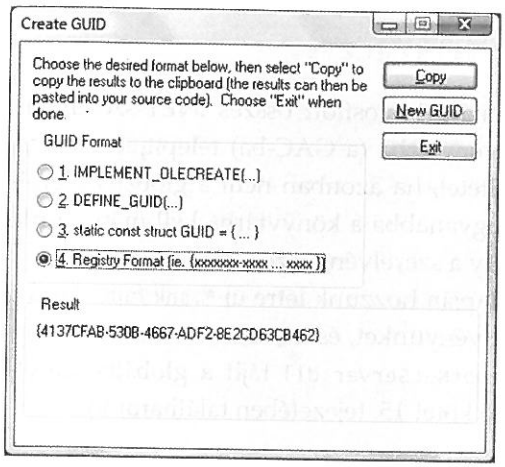
```
// Szükségünk van rá, hogy megszerezzük a szükséges
// együttműködési attribútumokat.
using System.Runtime.InteropServices;

namespace ComCallableDotNetServer
{
  [ClassInterface(ClassInterfaceType.AutoDual)]
  public class DotNetCalc
  {
    public int Add(int x, int y)
    { return x + y; }

    public int Subtract(int x, int y)
    { return x - y; }
  }
}
```

A COM világában majdnem mindent egy 128 bites számmal, az úgynevezett GUID-dal azonosítunk. Ezeket az értékeket a rendszerleíró adatbázis rögzíti azért, hogy definiálja egy COM-típus azonosítóját. Jelenleg nem határoztunk meg GUID-értékeket a `DotNetCalc` osztályunk számára, ezért a típuskönyvtár-exportáló eszköz (`tlbexp.exe`) menet közben hozza létre a GUID azonosítókat. Ezzel a megközelítéssel az a probléma, hogy minden alkalommal, amikor a típuskönyvtárat generáljuk (és ezt hamarosan megtesszük), egyedi GUID-értékeket kapunk, amelyek lehetetlenné tehetik a meglévő COM-kliensek működését.

Saját GUID-értékek definiálásához használhatjuk a `guidgen.exe` segédprogramot, amely a Visual Studio 2008 Tools > Create Guid menüpontjából érhető el. Bár az eszköz négy GUID-formátumot biztosít a számunkra, a `[Guid]` attribútum megköveteli, hogy az A.16. ábrán látható módon a GUID-értéket a rendszerleíró formátumopció használatával adjuk meg.



A.16. ábra. *GUID-érték előállítása*

Ha ezt az értéket vágólapra másoljuk (a Copy GUID gomb segítségével), argumentumként beilleszthetjük a `[Guid]` attribútumba. A kapcsos zárójeleket el kell távolítani a GUID-értékből. A módosított `DotNetCalc` osztálytípus (a GUID-érték különböző lehet) a következő:

```
[ClassInterface(ClassInterfaceType.AutoDual)]
[Guid("4137CFAB-530B-4667-ADF2-8E2CD63CB462")]
public class DotNetCalc
{
  public int Add(int x, int y)
  { return x + y; }

  public int Subtract(int x, int y)
  { return x - y; }
}
```

A Solution Explorerben kattintsunk a Show All Files gombra, majd nyissuk meg a Properties ikon alatt található `AssemblyInfo.cs` fájlt. Az alapértelmezés szerint az összes Visual Studio 2008 projekt munkaterülete rendelkezik szerelvényszintű `[Guid]` attribútummal, amely a .NET-kiszolgáló alapján generált típuskönyvtár GUID-jának azonosítására szolgál (ha a COM hozzáférhet).

```
// A következő GUID a típuskönyvtár azonosítója, ha ez a projekt
// hozzáférhető a COM számára.
[Assembly: Guid("EB268C4F-EB36-464C-8A25-93212C00DC89")]
```

Erős név definiálása

Hasznos gyakorlat, ha a COM számára biztosított összes .NET-szerelvény erős nevet kap, és a globális szerelvénytárba (a GAC-ba) telepítjük őket. A normál működéshez ez nem alapfeltétel, ha azonban nem a globális szerelvénytárba telepítjük a szerelvényt, ugyanabba a könyvtárba kell másolnunk, ahol az a COM-alkalmazás van, amely a szerelvényt szeretné használni.

A Properties szerkesztő Signing lapján hozzunk létre új `*.snk` fájlt aláírási célokra. Ekkor lefordíthatjuk a szerelvényünket, és a `gacutil.exe` segédprogrammal telepíthetjük a `ComCallableDotNetServer.dll` fájlt a globális szerelvénytárba (ehhez részleteket az előző kötet 15. fejezetében találhatunk).

```
gacutil -i ComCallableDotNetServer.dll
```

A típuskönyvtár létrehozása és a .NET-típusok regisztrálása

Már elkészíthetjük a szükséges COM-típuskönyvtárat, és regisztrálhatjuk a .NET-szerelvényünket a rendszerleíró adatbázisban ahhoz, hogy a COM használhassa őket. Ennek végrehajtásához két lehetséges megközelítés áll a rendelkezésünkre. Az első lehetőség, hogy a .NET Framework 3.5 SDK `regasm.exe` parancssori eszközét alkalmazzuk. Ezzel az eszközzel több listázást hozzáadhatunk a rendszerleíró adatbázishoz, és ha megadjuk a `/tlb` kapcsolót, akkor létrehozhatjuk a szükséges típuskönyvtárat is:

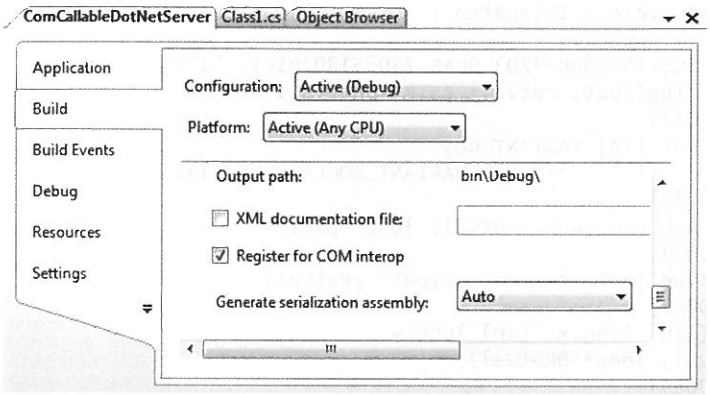
```
regasm ComCallableDotNetServer.dll /tlb
```

Megjegyzés A .NET Framework 3.5 SDK a `tlbexp.exe` nevű eszköz biztosítja. A `regasm.exe` segédprogramhoz hasonlóan ez az eszköz .NET-szerelvényből hoz létre típuskönyvtárakat, viszont nem adja hozzá a szükséges bejegyzéseket a rendszerleíró adatbázishoz. Emiatt általában a `regasm.exe`-t használjuk az összes szükséges lépés végrehajtásához.

Bár a `regasm.exe` a lehető legrugalmasabb eszköz, ha a COM–típuskönyvtár készítését szeretnénk szabályozni, a Visual Studio 2008 egy kezesebb alternatíva. A Properties szerkesztőben az A.17. ábrán látható módon kapcsoljuk be a Register for COM interop jelölőnégyzetet a Compile lapon, majd fordítsuk le újra a szerelvényünket.

Amint futtattuk a `regasm.exe` segédprogramot, vagy engedélyeztük a Register for COM interop lehetőséget, láthatjuk, hogy a bin\Debug mappa (`*.tlb` kiterjesztéssel) egy COM-típuskönyvtárfájlt tartalmaz.

Forráskód A ComCallableDotNetServer kódfájlokat a forráskódkönyvtár A Függelékének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.



A.17. ábra: *Szerelvény regisztrálása COM-mal való együttműködésre a Visual Studio 2008 használatával*

Az exportált típus adatainak a vizsgálata

A megfelelő COM-típuskönyvtár tartalmát az OLE View segédprogrammal nézhetjük meg, ha betöltjük a `*.tlb` fájlt. Ha betöltjük a `ComCallableDotNetServer.tlb` fájlt (a File > View Type Library menüpont segítségével), láthatjuk az összes .NET-osztálytípusunk COM-típusleírását. A `DotNetCalc` osztály definíciója például előírja, hogy az osztály a `[ClassInterface]` attribútumnak köszönhetően támogassa az alapértelmezett `_DotNetClass` interfészt, valamint az `_Object` interfészt. Ez a `System.Object` által definiált funkcionalitás nem felügyelt definíciója:

```
[uuid(88737214-2E55-4D1B-A354-7A538BD9AB2D),
  version(1.0), custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
  "ComCallableDotNetServer.DotNetCalc")]
coclass DotNetCalc {
  [default] interface _DotNetCalc;
  interface _Object;
};
```

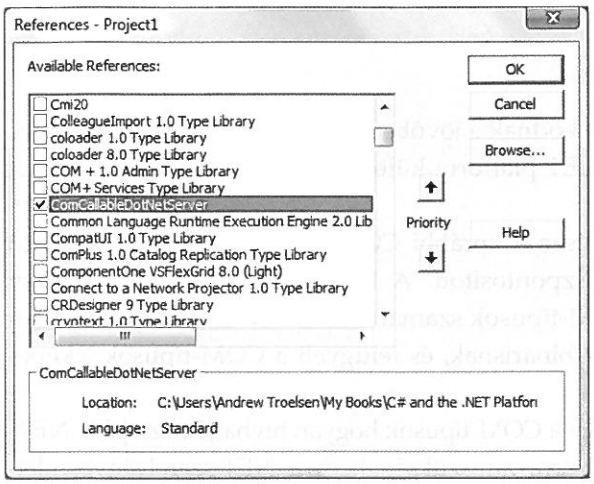
Ahogy a [ClassInterface] attribútummal megadtuk, az alapértelmezett interfészt kettős interfészként állítottuk be, ezért korai és késői kötés segítségével is hozzáférhetünk:

```
[odl, uuid(AC807681-8C59-39A2-AD49-3072994C1EB1), hidden,
  dual, nonextensible, oleautomation,
  custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
  "ComCallableDotNetServer.DotNetCalc")]
interface _DotNetCalc : IDispatch {
  [id(00000000), propget,
    custom({54FC8F55-38DE-4703-9C4E-250351302B1C}, "1")]
  HRESULT ToString([out, retval] BSTR* pRetVal);
  [id(0x60020001)]
  HRESULT Equals( [in] VARIANT obj,
                  [out, retval] VARIANT_BOOL* pRetVal);
  [id(0x60020002)]
  HRESULT GetHashCode([out, retval] long* pRetVal);
  [id(0x60020003)]
  HRESULT GetType([out, retval] _Type** pRetVal);
  [id(0x60020004)]
  HRESULT Add([in] long x, [in] long y,
    [out, retval] long* pRetVal);
  [id(0x60020005)]
  HRESULT Subtract( [in] long x, [in] long y,
                    [out, retval] long* pRetVal);
};
```

A _DotNetCalc interfész nemcsak az Add() és a Subtract() metódusokat írja le, hanem hozzáférhetővé teszi a System.Object által örökölt tagokat. Szabálynak tekinthetjük, hogy amikor egy .NET-osztálytípust a COM számára hozzáférhetővé teszünk, a származási láncban található összes nyilvános metódus megjelenik az automatikusan generált osztályinterfészen keresztül.

Visual Basic 6.0 tesztkliens készítése

Mivel a .NET-szerelvényt megfelelően konfiguráltuk, így együttműködhet a COM-futtatórendszerrel, készíthetünk egy COM-klienst. Hozzunk létre egy egyszerű VB6 Standard *.exe projekttípust (VB6DotNetClient néven), majd állítsunk be referenciát az újonnan generált típuskönyvtárra (lásd az A.18. ábrát).



A.18. ábra: *Hivatkozás a .NET-kiszolgálóra a VB6-ból*

Ami a GUI front endet illeti, legyen teljesen egyszerű. Egyszerűen egy gombbal kezeljük a `DotNetCalc` .NET-típust. A kód a következő (figyeljük meg, hogy meghívjuk az `_Object` interfész által definiált `ToString()` metódust):

```
Private Sub btnUseDotNetObject_Click()
  ' A .NET-objektum létrehozása.
  Dim c As New DotNetCalc
  MsgBox c.Add(10, 10), , "Adding with .NET"

  ' A System.Object néhány tagjának meghívása.
  MsgBox c.ToString, , "ToString value"
End Sub
```

Forráskód A VB6DotNetClient kódfájlokat a forráskódkönyvtár A Függelékének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Láttuk olyan .NET-alkalmazások készítésének folyamatát, amelyek a COM-típusokkal kommunikálnak, és olyan COM-alkalmazásokét, amelyek a .NET-típusokkal kommunikálnak. Ezzel már biztos alappal rendelkezünk a további kutatáshoz.

Összefoglalás

A felügyelt és nem felügyelt kódnak a jövőben meg kell tanulnia időről időre együtt dolgozni. Emiatt a .NET platform különböző módszereket kínál a két világ összeolvasztására.

A függelék nagyobbik része a korábbi COM-komponenseket alkalmazó .NET-típusok részleteire összpontosított. A folyamat egy szerelvényproxy készítésével kezdődik a COM-típusok számára. Az RCW továbbítja a hívásokat az alapul szolgáló COM-binárisnak, és felügyeli a COM-típusok leképezését .NET-ekvivalensükre.

Ezután azt vizsgáltuk, hogy a COM-típusok hogyan hívhatják az újabb .NET-típusok szolgáltatásait. Ehhez arra van szükség, hogy a .NET-szerelvényben létrehozható típusokat regisztráljuk a COM számára, és a .NET-típusokat leírja egy COM-típuskönyvtár.