

```

static void queryStringWithOperators()
{
    Console.WriteLine("***** using Query Operators *****");
    string[] currentVideogames = {"Morrowind", "Bioshock",
    "Half Life 2: Episode 1", "The Darkness",
    "Daxter", "System Shock 2"};

    // Lekérdező kifejezés készítése lekérdezésoперátorok
    // használatával.
    var subset = from g in currentVideogames
    where g.Length > 6 orderby g select g;

    // Az eredmények kitírása.
    foreach (var s in subset)
        Console.WriteLine("Item: {0}", s);
}

```

A C#-lekérdezésoперátorok nyilvánvaló előnye a lekérdező kifejezések írásában, hogy a func<> metódusreferenciák és az Enumerable típus hívásai kívüli esnek a programozó látókörén, hiszen a fordításnak ezt a részét a C#-fordító-nak kell végrehajtania. A különböző lekérdezésoперátorok (from, in, where, orderby stb.) alkalmazása LINQ-kifejezések készítésére a lehető legáltalánosabban elterjedt és legkézenfekvőbb megközelítés.

Lekérdező kifejezések létrehozása az Enumerable

típus és a lambdák használataival

Az itt használt LINQ-lekérdezésoперátorok leegyszerűsített verziónak az enumerable típus különböző metódusainak hívásához. Nézzük meg a következő queryStringWithEnumerableAndLambda() metódust, amely a lokális-san tárolt sztringtömböt dolgozza fel az enumerable bővítő metódusainak közvetlen használatával:

```

static void queryStringWithEnumerableAndLambda()
{
    Console.WriteLine("***** using Enumerable /
    Lambda Expressions *****");
    string[] currentVideogames = {"Morrowind", "Bioshock",
    "Half Life 2: Episode 1", "The Darkness",
    "Daxter", "System Shock 2"};

    // Lekérdezőkifejező készítése bővítő metódusok használatával,
    // amelyek az Enumerable típus által érkező a tömböt.
    var subset = currentVideogames.Where(game => game.Length > 6)
    .OrderBy(game => game).Select(game => game);
}

```

```
// Az eredmények kitírása.
foreach (var game in subset)
    Console.WriteLine("{0}", game);
Console.WriteLine();
}
```

Ebben a programészben meghívjuk a sztringtömbobjektum generikus `where()` metódusát, amely az `Array` típushoz van rendelve úgy, mint az `Enumerable` által definiált bővítő metódus. Az `Enumerable.Where()` metódus a `System.Func<T0, TResult>`, `result` metódusreferencia-típust használja. A metódusreferencia el-ső paramétertípusa a feldolgozandó `IEnumerable<T>`-kompatibilis adatokat jelképezi (ebben az esetben ez sztringtömb), míg a második paramétertípus azt a metódust reprezentálja, amely feldolgozza az említett adatokat.

Mint ahogy a lambda kifejezések használata mellett döntöttünk (ahelyett, hogy közvetlenül létrehoznánk a `Func<T>` egy példányát vagy egy névtelen metódust), megadjuk, hogy a `game` paramétert a `game.Length > 6` utasítás dolgozza fel, amelynek boolean visszatérési típusa van.

A `where<T>()` metódus visszatérési értéke implicit típusú, de a felszín alatt egy `OrderedEnumerable` típuson dolgozunk. Az eredményobjektumból meghívjuk a generikus `orderBy<T, K>()` metódust, amely ugyancsak egy `Func<T, K>` metódusreferencia-paramétert igényel. Végezetül a megadott lambda kifejezés eredményéből kiválasztunk minden elemet, ismét a `Func<T, K>` használataval.

A bővítő metódusok egyedi, hiszen példányszintű tagként hívhatók meg azon a típuson, amelyet bővítenek (ebben az esetben a `System.Array`), vagy statikus tagként azt a típust használva, amelyben definiálva lettek. Így a következőképpen is megírhattuk volna a lekérdező kifejezésünket:

```
var subset = Enumerable.Where(currentvideogames, game =>
    game.Length > 6).OrderBy(game => game).Select(game => game);
```

Az `Enumerable` típus metódusainak direkt használataval sokkal „bőbeszédűbb” LINQ lekérdező kifejezést kell írni, mint a `C#` lekérdezésopektorával. Emellett, mint ahogy az `Enumerable` típus metódusai metódusreferenciákat várnak paraméterként, általában szükség van a lambda kifejezés létrehozására, hogy az alap metódusreferencia-célnak megengedjük a bemeneti adat feldolgozását.

Lekérdező kifejezések létrehozása az Enumerable típus és a névtelen metódusok használatával

Mint ahogy a C# 2008 lambda kifejezései egy szűrőn a névtelen metódusok egyszerűsített változatai, nézzük meg a harmadik lekérdező kifejezést, amelyet a `queryStringwithAnonymousMethods()` segédfüggvényben hozunk létre:

```
static void queryStringwithAnonymousMethods()
{
    Console.WriteLine("***** using Anonymous Methods *****");

    string[] currentVideogames = {"Morrowind", "BioShock",
        "Half Life 2: Episode 1", "The Darkness",
        "Daxter", "System Shock 2"};

    // Elkészítjük a szükséges Func< metódusreferenciákat névtelen
    // metódusok használatával.
    Func<string, bool> searchFilter =
        delegate(string game) { return game.Length > 6; };
    Func<string, string> itemToProcess = delegate(string s)
    { return s; };

    // Adjuk át a metódusreferenciákat az Enumerable metódusainak.
    var subset = currentVideogames.Where(searchFilter)
        .OrderBy(itemToProcess).Select(itemToProcess);

    // Az eredmények kitírása.
    foreach (var game in subset)
        Console.WriteLine("{0}", game);
    Console.WriteLine();
}
```

A lekérdező kifejezés iterációja még hosszabb, ugyanis manuálisan készítjük el azokat a `Func< metódusreferenciákat, amelyek az Enumerable típus where(), orderby() és select() metódusai használnak.` Pozitívum az, hogy a névtelenmetódus-szintaxis megtartja az összes fel- dolgozást, amelyet egy egyszerű metódusdefinió tartalmaz. Ez a metódus funkcionálisan megegyezik az előzőekben létrehozott `queryStringwithAnonymousLambda() és queryStringwithOperators()` metódusokkal.

Lekérdező kifejezések létrehozása az Enumerable típus és a nyers metódusreferenciák használatával

Ha végül egy *igazán többször* lekérdező kifejezést szeretnénk írni, elkérülhetünk a lambda-k és a névtelen metódusok szintaxisát, és közvetlenül készíthetünk metódusreferencia-célt minden egyes `Func<>` típus számára. A lekérdező kifejezésünk végső iterációja, amelyet egy új osztályban, a `VeryComplexQueryExpression`-ben modellezünk, a következő:

```
class VeryComplexQueryExpression
{
    public static void QueryStringsWithRawDelegates()
    {
        Console.WriteLine("***** Using Raw Delegates *****");

        string[] currentVideogames = {"Morrowind", "Bioshock",
            "Half Life 2: Episode 1", "The Darkness",
            "Daxter", "System Shock 2"};

        // Elkészítjük a szükséges Func<> metódusreferenciákat névtelen
        Func<string, bool> searchFilter = new Func<string,
            bool>(Filter);
        Func<string, string> itemToProcess =
            new Func<string, string>(ProcessItem);

        // Adjuk át a metódusreferenciákat az Enumerable metódusainak.
        var subset = currentVideogames
            .Where(searchFilter).OrderBy(itemToProcess).
            Select(itemToProcess);

        // Az eredmények kitírása.
        foreach (var game in subset)
            Console.WriteLine("Item: {0}", game);
        Console.WriteLine();
    }

    // A metódusreferenciák célpontjai.
    public static bool Filter(string s) { return s.Length > 6; }
    public static string ProcessItem(string s) { return s; }
}
```

Kiprobállhatjuk stringfeldolgozó logikánk ezen iterációját, ha meghívjuk a `Program` osztály `Main()` metódusából a következők szerint:

```
VeryComplexQueryExpression.QueryStringsWithRawDelegates();
```


Ha most lefuttatjuk az alkalmazásunkat azért, hogy minden megoldást le-
teszteljünk, bármelyik utat is választottuk, azt tapasztaljuk, hogy az ered-
mény azonos lesz. A LINQ lekérdező kifejezések belső reprezentálásáról tart-
suk szem előtt a következőket:

- A lekérdező kifejezések különböző C#-lekérdezésoперátorokkal ké-
szülnek.

- A lekérdezésoперátorok egyszerűsítések a System.Linq.Enumerable tí-
pus által definiált bővítő metódusok hívására.
- Az Enumerable számos metódusa metódusreferenciákat (konkrétan
Func<>) vár paraméterként.

- A C# 2008-ban minden metódusreferencia-paramétert igénylő metó-
dus helyett átadhatunk egy lambda kifejezést.
- A lambda kifejezések valójában leplezett névtelen metódusok (ez
nagyban javítja az olvashatóságot).

- A névtelen metódusok egyszerűsítések a nyers metódusreferencia al-
lokálásra és a metódusreferencia által mutatott metódusának manuá-
lis készítésére.

Mind Ezek után térjünk át az operátorok vizsgálatára.

Forráskód A LinqOverArrayUsingEnumerable projekt a 14. fejezet alkönyvtárában található.

A LINQ-lekérdezésoперátorok

A C# számos lekérdezésoперátort definiál. A 14.3. táblázat bemutatja a leg-
gyakrabban használtakat.

Megjegyzés A .NET keretrendszer 3.5 SDK dokumentációja teljes körű leírást ad minden egyes
C# LINQ-operátorról. A „LINQ általános programozói útmutatóban” (LINQ General Programming
Guide) további információk találhatók.

Lekérdezésoperátorok Valós jelentés

from, in	Minden LINQ-kifejezés gerincének definiálására szolgál, amelyek segítségével az adatok egy részhalmazát lehet kinyerni egy illeszkeddő tárolóból.
where	Olyan megkötés definiálására szolgál, amely alapján ki kell választani a tárolóból az elemeket.
select	Adatsorozatot kiválasztására szolgál a tárolóból.
join, on, equals, into	Összekapcsolást hajt végre egy adott kulcs alapján. Nem szükséges, hogy ezeknek az „összekapcsolásoknak” bár milyen közülük legyen a relációs adatbázis adataihoz.
orderby, ascending, descending	Az eredményrészhalmazt növekvő vagy csökkenő irányba lehet rendezni a segítségével.
group, by	Egy meghatározott értéket szerint csoportosítja a részhalmaz adatait.

14.3. táblázat: Különböző LINQ-lekérdezésoperátorok

A 14.3. táblázatban bemutatott operátorok részletes listáján kívül az Enumerablenek köztvetlen C# lekérdezésoperátor-párjaik, hanem bővítő módszereknek mince- rable típus a módszerek egy olyan halmazát is felkínálja, amelyeknek nincs- nek köztvetlen C# lekérdezésoperátor-párjaik, hanem bővítő módszereknek mince- érhetők el. Ezeket a generikus módszereket lehet meghívni ízt, hogy az eredményhalmazt különböző módokon átalakítsák (Reverse(), ToArray(), ToList() stb.). Néhányuk arra szolgál, hogy egyedi elemeket válasszon ki az eredményhalmazból, mások különböző műveleteket hajtanak végre (Dis- tinct(), Union(), Intersect() stb.), megint mások pedig az eredménye- ket csoportosítják (Count(), Sum(), Min(), Max() stb.).

Elemek számának meghatározása az Enumerable segítségével

Ezeknek a lekérdezésoperátoroknak (és a System.Linq.Enumerable típus ki- egészítő tagjainak) használatával erősen tipizáltan nagyon kifejező lekérd- zéskifejezéseket tudunk írni. Ahhoz, hogy egy Enumerable bővítő módszert hívassunk, a LINQ-kifejezést először IEnumerablen<T>-kompatibilis obje- kttumra kell kasztolnunk, ezt a LINQ-kifejezés zárójelezésével érhetjük el.

Ugyanezt tettük, amikor az azonnali futtatást vizsgáltuk; de nézzünk egy másik példát, amely lehetővé teszi, hogy meghatározzuk a LINQ-lekérdezés által visszaadott elemek számát:

```
static void GetCount()
{
    string[] currentVideogames = {"Morrowind", "Bioshock",
    "Half Life 2: Episode 1", "The Darkness",
    "Daxter", "System Shock 2"};

    // Határozzuk meg a darabszámot a lekérdezésből.
    int numb = (from g in currentVideogames
    where g.Length > 6
    orderby g
    select g).Count<string>();

    // A numb értéke 5.
    Console.WriteLine("{0} items honor the LINQ query.", numb);
}
```

Egyszerű tesztprojektek létrehozása

A bonyolultabb LINQ-lekérdezések készítésének megértéséhez hozzunk létre egy új parancssori alkalmazást FunwithLINQexpressions névvel. Ezután definiáljunk egy egyszerű car típust, ezáltal egy egyedi ToString() implementációval, hogy gyorsan megnezhessük az objektum állapotát:

```
class Car
{
    public string PetName = string.Empty;
    public string Color = string.Empty;
    public int Speed;
    public string Make = string.Empty;

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

Töltsünk fel egy tömböt a következő a car objektumokkal a main() metódu-
son belül:

```

static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Query Expressions *****\n");

    //Ez a tömb lesz a teszteleseink alapja...
    Car[] myCars = new [] {
        new Car { PetName = "Henry", Color = "Silver", Speed = 100,
            Make = "BMW"},
        new Car { PetName = "Daisy", Color = "Tan", Speed = 90,
            Make = "BMW"},
        new Car { PetName = "Mary", Color = "Black", Speed = 55,
            Make = "VW"},
        new Car { PetName = "Clunker", Color = "Rust", Speed = 5,
            Make = "VW"},
        new Car { PetName = "Hank", Color = "Tan", Speed = 0,
            Make = "Ford"},
        new Car { PetName = "Sven", Color = "White", Speed = 90,
            Make = "Ford"},
        new Car { PetName = "Mary", Color = "Black", Speed = 55,
            Make = "VW"},
        new Car { PetName = "Zippy", Color = "Yellow", Speed = 55,
            Make = "VW"},
        new Car { PetName = "Melvin", Color = "White", Speed = 43,
            Make = "Ford"}
    };

    //Itt fogunk különböző metódusokat meghívni!
    Console.ReadLine();
}

```

A kiválasztó szintaxis alapjai

Mivel a LINQ lekérdező kifejezések fordítási időben validálódnak, ezért az operátorok sorrendje nagyon fontos. Minden LINQ-kifejezés a `from`, az `in` és a `select` operátorból épül fel:

```
var result = from item in container select item;
```

Ekkor a lekérdezésünk nem csinál mást, mint kiválasztja a tároló összes elemét (hasznolva a `select * SQL` utasításhoz). Nézzük meg a következőket:

```

static void BasicSelection(Car[] myCars)
{
    //Kérünk mindent.
    Console.WriteLine("All cars:");
    var allCars = from c in myCars select c;
    foreach (var c in allCars)
    {
        Console.WriteLine(c.ToString());
    }
}

```

Ez a lekérdezés nem igazán hasznos, hiszen a részhalmozunk megegyezik a bemeneti paraméter adataival. Használhatjuk a bemeneti paramétert úgy, hogy minden egyes autónak csak a PetName értékét kapjuk vissza a következő lekérdezés szintaxisával:

```
// kérdezzük le csak az autók neveit.
Console.WriteLine("Only PetNames:");
var names = from c in myCars select c.PetName;

foreach (var n in names)
{
    Console.WriteLine("Name: {0}", n);
}
```

Ekkor a names valóban egy belső típus, amely implementálja az IEnumerable<string> típust, hiszen a car objektumoknak csak a PetName értékét válasszúk ki. A var kulcsszó révén az implicit típusálás segítségével leegyszerűsödött a kódolási feladat.

Nézzük meg a következő teendőt. Mi lenne, ha a járművek típusát szeretnénk visszakapni és megjeleltetni? Ha a következő lekérdezést írjuk meg:

```
var makes = from c in myCars select c.Make;
```

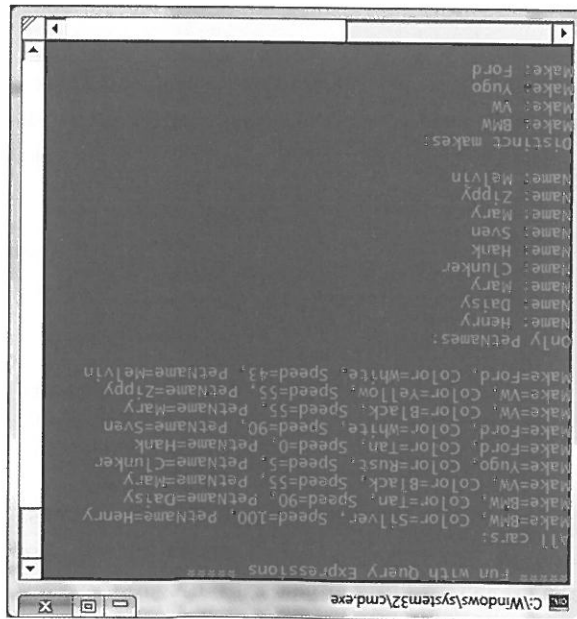
akkor jó néhány redundáns listát kapunk eredményül, ugyanis a BMW, a Ford és a VW márkák többször is szerepelnek. Az Enumerable.Distinct<T>() metódus segítségével kiszűrhetjük a duplikátumokat:

```
var makes = (from c in myCars select c.Make).Distinct<string>();
```

Amikor meghívunk bármely, az Enumerabale által definiált bővítő metódust, ezt megtehetjük a lekérdező kifejezés készítésekor (ahogy azt az előző példában láthattuk) vagy bővítő metóduson keresztül egy kompatibilis tömb-típuson. Ezért a következő kód azonos eredményt fog adni:

```
var makes = from c in myCars select c.Make;
Console.WriteLine("Distinct makes:");
foreach (var m in makes.Distinct<string>())
{
    Console.WriteLine("Make: {0}", m);
}
```

A 14.7. ábra mutatja a BasicSelections() metódus hívásának az eredményét.



14.7. ábra: Alapvető adatok lekérdezése a Car [] paraméterből

Adatok egy részhalmozának kinyerése

A tárolók által tartalmazott adatok egy részhalmozát a `where` operátor segítségével kérhetjük le. Ekkor az általános mintánk a következőképpen fog kinézni:

```
var result = from item in container where Boolean expression
select item;
```

A `where` operátor olyan kifejezést vár, amely `Boolean` ad vissza. Ha például a `Car[]` paraméterből csak azokat az elemeket szeretnénk kiválasztani, amelyeknek `make` mezője „BMW” értékű, a következő kódot írhatjuk meg egy új metóduson belül, amelyet nevezzünk `getSubsets()`-nek:

```
static void getSubsets(Car[] myCars)
{
    // kérdezzük le csak a BMW-ket.
    var onlyBmw = from c in myCars where c.make == "BMW" select c;
    foreach (Car c in onlyBmw)
    {
        Console.WriteLine(c.ToString());
    }
}
```

Ha egy `where` kifejezést hozunk létre, bármely érvényes `C#`-operátor használata megengedhető komplex kifejezések készítése céljából. Nézzük meg például a következő lekérdezést, amely csak azokra a BMW-kre kíváncsi, amelyeknek a sebessége legalább 100 mph:

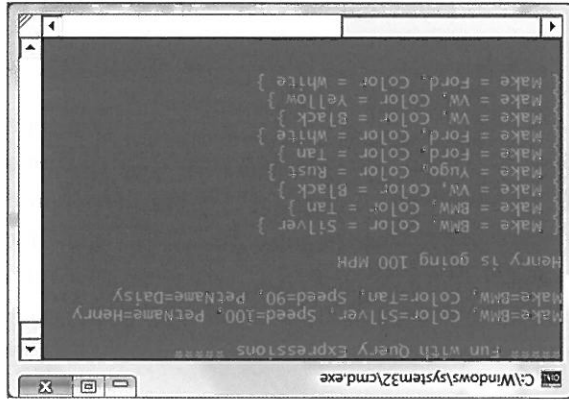
```
//Kérdezzük le azokat a BMW-ket, amelyek sebessége legalább 100 mph.
var onlyFastBmw = from c in myCars
where c.Make == "BMW" && c.Speed >= 100
select c;
foreach (Car c in onlyFastBmw)
{
    Console.WriteLine("{0} is going {1} MPH", c.PetName, c.Speed);
}
```

Új adattípusok leképázése

Lehetőségünk van arra is, hogy létező adattorrásból új adatformákat képezzünk le. Tegyük fel, hogy fogjuk a `bemeneti car[]` paramétert, és egy olyan eredményhalmazt szeretnénk visszakapni, amely a járművek típusát és a színt tartalmazza. Ehhez definiálhatunk egy `select` utasítást, amely a `# 2008 új típusú használatával` dinamikusan hoz létre új típusokat. A fordító minden megadott névhez egy `trásvédett tulajdonságot` és egy `trásvédett segédmezőt` definiál, valamint felülbírálja a `ToString()`, a `GetHashCode()` és az `Equals()` metódusokat (lásd 13. fejezet):

```
var makeColors = from c in myCars select new {c.Make, c.Color};
{
    //Használhatunk a Make és Color tulajdonságot közvetlenül is.
    Console.WriteLine(c.ToString());
}
```

A 14.8. ábra megmutatja az új lekérdezések kimenetét.



14.8. ábra: Részhalmozatok felsorolása

Az eredményhalmaz-elemek sorrendjének megfordítása

Az eredményhalmaz elemeinek sorrendjét elég egyszerűen meg lehet fordítani az Enumerabile típus generikus `Reverse<T>()` módszerével. A következő módszer például fordított sorrendben adja vissza a bemeneti `car[]` paraméter elemeit:

```
static void ReverseSelection(car[] mycars)
{
    // kérünk mindent fordított sorrendben.
    Console.WriteLine("All cars in reverse:");
    var subset = (from c in mycars select c).Reverse<Car>();
    foreach (car c in subset)
    {
        Console.WriteLine("{0} is going {1} MPH", c.Petname, c.Speed);
    }
}
```

Ebben a példában a `Reverse<T>()` módszert a lekérdezés létrehozásakor hívtuk meg. Alternatívaként megtehetjük azt is, hogy a `mycars` tömbön hívjuk meg ezt a módszert:

```
static void ReverseSelection(car[] mycars)
{
    // kérünk mindent fordított sorrendben.
    Console.WriteLine("All cars in reverse:");
    var subset = from c in mycars select c;
    foreach (car c in subset.Reverse<Car>())
    {
        Console.WriteLine(c.ToString());
    }
}
```

Kifejezések rendezése

Ahogy már láthattuk, a lekérdező kifejezés tartalmazhat egy `orderby` operátort, amellyel a részhalmaz elemeit adott érték szerint lehet rendezni. Az alapértelmezés szerint a sorrend növekvő, ezért ha sztring szerint rendez-nénk, az abcéssorrendben lenne, ha numerikus adatok szerint, akkor az alacsonyabb értéktől haladnánk a magasabb felé és így tovább. Ha az eredményt csökkenő sorrendben szeretnénk látni, egyszerűen be kell szűrni a `descending` operátort. Nézzük meg a következő módszert:

```
static void OrderedResults(car[] mycars)
{
    //Rendezzük az autókat Petname tulajdonság szerint.
    var subset = from c in mycars orderby c.Petname select c;
```



```

Console.WriteLine("Ordered by PetName:");
foreach (Car c in subset)
{
    Console.WriteLine(c.ToString());
}

//Most keressük ki azokat az autókat, amelyek kevesebbet mennek,
// mint 55 mph, és rendezzük a PetName érték alapján csökkenő
// sorrendbe.
subset = from c in myCars
where c.Speed > 55 orderby c.PetName descending select c;
Console.WriteLine("\nCars going faster than 55, ordered by
PetName:");
foreach (Car c in subset)
{
    Console.WriteLine(c.ToString());
}
}

```

Noha a növekvő sorrend az alapbeállítás, lehetőségünk van a kihatagsúlyozására az ascending operátor használatával:

```

var subset = from c in myCars
orderby c.PetName ascending select c;

```

Az alap rendezett lekérdezés kifejezés formátuma a következő:

```

var result = from item in container orderby value
ascending/descending select item;

```

Különbségek keresése

Végül vegyünk görcső alá egy olyan LINQ-lekérdezést, amely eredmény-halmazként két IEnumerable<T>-kompatibilis tároló különbségét adja vissza. Nézzük meg a következő metódust, amely az Enumerable.Except() metódus használatával példánkban egy „Yugó”-t ad vissza:

```

static void GetDiff()
{
    List<string> myCars = new List<string>
{ "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<string>
{ "BMW", "Saab", "Aztec" };
    var cardiff = (from c in myCars select c)
.Except(from c2 in yourCars select c2);
}

```

```

Console.WriteLine("Here is what you don't have, but I do:");
foreach (string s in card1.ff)
    Console.WriteLine(s); // Prints yugo.
}

```

Végezetül vizsgáljuk meg a LINQ-lekérdezések mint módszer-visszatérési értékek témáját.

Forráskód A `FunWithLINQExpressions` projekt a 14. fejezet alkönyvtárában található.

LINQ-lekérdezések: szigetek a szigeteken?

A fejezetben bemutatott összes LINQ-lekérdezést egy lokális módszer hatókörén belül definiáltuk. Sőt, hogy tovább egyszerűsítsük a programozást, az eredményt tároló változó egy implicit típusú lokális változóba került (leképezések esetén ez tulajdonképpen kötelező is). Az implicit típusú lokális változók *new* használható paraméterek, visszatérési értékek vagy egy osztály-típus mezőinek definiálására (lásd 13. fejezet).

Ennek tudatában feltehető a kérdés: pontosan hogyan lehet egy lekérdezés eredményét visszaadni egy külső hívónak. A válasz az, hogy attól függ. Ha az eredményhalmaz erősen tipizált adatokat tartalmaz (pl. sztringtömböt, a car objektumok listát vagy bármit), akkor elhagyhatjuk a var kulcsszót, és helyette használhatunk egy megfelelő `IEnumerable<T>` vagy `IEnumerable` típust (az `IEnumerable<T>` kibővíti az `IEnumerable`-t). Nézzük meg a következő mintapéldát egy új .NET 3.5 parancssori alkalmazás számára, amelynek `LinqRetVal`-es a neve:

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** LINQ Transformations *****\n");
        IEnumerable<string> subset = GetStringSubset();
        foreach (string item in subset)
        {
            Console.WriteLine(item);
        }
        Console.ReadLine();
    }
}

```

```

static IEnumerable<string> GetStringSubsets()
{
    string[] currentVideogames = {"Morrowind", "Bioshock",
    "Half Life 2: Episode 1", "The Darkness",
    "Daxter", "System Shock 2"};

    // Jegyezzük meg, hogy a részhalmoz egy IEnumerable<string>-
    // kompatibilis objektum.
    IEnumerable<string> subset = from g in currentVideogames
    where g.Length > 6
    orderby g
    select g;
    return subset;
}

```

A példa csak azért működik a várakozásnak megfelelően, mert a GetStringSubsets() visszatérési értéke és ezen a metóduson belül a LINQ-lekérdezés erősen tipizáltak. Ha használnánk a var kulcsszót a subset változó definíciájához, *csak* akkor adhatnánk vissza az értéket, ha a metódust úgy írtuk volna meg, hogy egy IEnumerable<string> típusal térjen vissza (és ha az implicit típusú lokális változó kompatibilis a megadott visszatérési érték típusával). Ha a LINQ-lekérdezés használja a lekérdezést, nem tudhatjuk az alapadat-típust, azt ugyanis csak fordítási időben határozzuk meg. Ekkor a var kulcsszó használata kötelező, így a következő metódus nem fordulna le:

```

// Hiba! Nem térhetünk vissza var adattípussal!
static var GetProjectedSubsets()
{
    Car[] myCars = new Car[] {
        new Car { PetName = "Henry", Color = "Silver", Speed = 100,
        Make = "BMW" },
        new Car { PetName = "Daisy", Color = "Tan", Speed = 90,
        Make = "BMW" },
        new Car { PetName = "Mary", Color = "Black", Speed = 55,
        Make = "VW" },
        new Car { PetName = "Clunker", Color = "Rust", Speed = 5,
        Make = "Yugo" },
        new Car { PetName = "Melvin", Color = "White", Speed = 43,
        Make = "Ford" }
    };
    var makesColors = from c in myCars select new { c.Make, c.Color };
    return makesColors; //Nem!
}

```

Mint ahogy a visszatérési érték nem lehet implicit típusú, hogyan tudjuk mégis visszaadni a makesColors objektumot egy külső hívónak?

Lekérdezéseredmények transzformálása Array típusú

Ha lekérzett adatot szeretnénk visszaadni a hívónak, az egyik megoldás az, hogy a lekérdezés eredményét standard CLR Array objektummá alakítjuk a `ToArray<T>` bővítő módszer használatával. Ezért, ha a következőképpen módosítanánk a lekérdező kifejezésünket:

```
// A visszatérést érték most már Array típusú.
static Array GetProjectedSubset()
{
    car[] myCars = new car[] {
        new Car { PetName = "Henry", Color = "Silver", Speed = 100,
            make = "BMW"},
        new Car { PetName = "Daisy", Color = "Tan", Speed = 90,
            make = "BMW"},
        new Car { PetName = "Mary", Color = "Black", Speed = 55,
            make = "VW"},
        new Car { PetName = "Clunker", Color = "Rust", Speed = 5,
            make = "Yugo"},
        new Car { PetName = "Melvin", Color = "White", Speed = 43,
            make = "Ford"}
    };

    var makesColors = from c in myCars select new { c.Make, c.Color };

    // Képezzük le a névlelen objektumok halmazát Array objektumokká.
    // Itt a generikus paramétertípus típuskövetkeztetésére
    // támaszkodunk, hiszen nem ismerjük a típus típusát!
    return makesColors.ToArray();
}
```

akkor a `main()` módszerből a következőképpen hívhatnánk meg és dolgozhatnánk fel az adatokat:

```
Array objs = GetProjectedSubset();
foreach (object o in objs)
{
    Console.WriteLine(o); // Meghívja a ToString() metódust minden
    // névlelen objektumra.
}
```

Egy egyszerű `System.Array` objektumot kellett használnunk a fenti példában, és nem használhattuk a `C#`-tömb deklarációs szintaxisát, ugyanis nem ismerjük a tartalmazott típus alapípusát. Nem adtuk meg továbbá a generikus `ToArray<T>` metódus paraméterípusát, hiszen a fordítási idő előtt nem ismerjük az adatípus (túl késő lenne nekünk, fejlesztőknek is, ha csak akkor derülne ki számunkra a tartalmazott típus).

A nyilvánvaló probléma ezzel a megoldással az, hogy elvesztjük az erős típusainkat, hiszen minden, az Array tömbben található objektumról azt tételezzük fel, hogy object típusú. Mindamellelt, ha olyan LINQ-eredményhalmazmal kell visszatérnünk, amely egy leképézés eredménye, az adatokat kötelezően Array típusú (vagy az Enumerab1e típus tagjai révén más megfelelő tárolóvá) kell alakítani.

Forráskód A LinqRetva1ues projekt a 14. fejezet alkönyvtárában található.

Összefoglalás

A LINQ egymáshoz kapcsolódó technológiák készlete, amelyek segítségével egyszerűen, szimmetrikus módon lehet változatos adattormákat kezelni. A LINQ bármelyik olyan típust tudja kezelni, amelyik implementálja az Enumerab1e<T> interfészt, akár egyszerű tömbökről legyen szó, akár generikus vagy nem generikus adatgyűjteményekről.

A LINQ-technológiákkal történő munka számos új C# 2008 nyelvi elemmel végezhető el. Egy LINQ lekérdező kifejezés például bármekkora számú eredményhalmaz-típussal visszatérhet, így általábanosan elterjedt módszer a var kulcsszó használata az adattípus reprezentálására. Használhatunk lambda kifejezéseket, objektuminitializáló szintaxist és névtelen típusokat is erős és tömör LINQ-lekérdezők készítésére.

Fontosabb azonban, hogy a C# LINQ-lekérdezősopertorok valójában a system.Linq.Enumerab1e típus statikus tagjainak hívására szolgáló egyszerűsítések. A legtöbb Enumerab1e tag Func<T> metódusreferencia-típusokon dolgozik, amelyek bemenetként kaphatnak literálismetódus-címeket, névtelen metódusokat vagy lambda kifejezéseket a lekérdezés kiértékeléséhez.

Programozás NET- szereklvénnyekkel

4. rész

A.NET-szerelvények

T I Z E N Ö T Ö D I K F E J E Z E T

A könyvben eddig létrehozott minden alkalmazás hagyományos, „önálló” alkalmazás volt, minthogy az egyedi programlogikát egyetlen végrehajtható fájl (*.exe) foglalta magában. A .NET platform egyik legjobb jellemzője azonban a *bináris újrafelhasználás*, amelynek során az alkalmazások a különböző külső szerelvények (más néven kódkönyvtárak) típusait használják. Ebben a fejezetben megvizsgáljuk a .NET-szerelvények elkészítésének, telepítésének és beállításának alapvető részleteit.

Először a .NET-névterek felépítésével, majd az egyfájlos és a többfájlos szerelvények, valamint a „privát” és „megosztott” szerelvények közötti különbségekkel ismerkedünk meg. Ezután megnezzük, hogy a .NET-futtatórendszer hogyan oldja fel a szerelvények helyét, végül tanulmányozzuk a GAC (Global Assembly Cache — globális szerelvénytár), az alkalmazáskonfigurációs fájlok (*.config fájlok), a kibocsátó házirenddel rendelkező szerelvények és a system.configuration névtér szerepét.

Egyedi névtér definíciója

Először nézzük meg az egyedi .NET-névterek létrehozását. A könyv korábbi részében olyan kisebb tesztprogramokat készítettünk, amelyek a .NET universeum létező névtérét (különösen a system névtérét) használják. Amikor saját egyedi alkalmazásainkat készítjük, célszerű a kapcsolódó típusokat egyedi névtérben csoportosítani. A C# nyelvben ezt a feladatot a namespace kulcsszóval oldhatjuk meg. Ez különösen .NET *.dll szerelvények létrehozásakor fontos, hiszen a többi fejlesztő az egyedi névtérünket fogja importálni ahhoz, hogy a típusainkat használni tudja.

Tételezzük fel, hogy geometriai osztályok gyűjteményének a fejlesztésén dolgozunk, az osztályok neve: square, circle és hexagon. A hasonlóságok alapján egyedi névtérben szeretnénk csoportosítani az osztályokat. Kétféle megközelítés áll a rendelkezésünkre. Dönthetünk úgy, hogy az osztályokat a következő módon, egyetlen fájlban (shapelib.cs) definiáljuk:

```
// shapes1b.cs
using System;

namespace MyShapes
{
    // Circle osztály
    class Circle { /* Interesting methods... */ }
    // Hexagon osztály
    class Hexagon { /* More interesting methods... */ }
    // Square osztály
    class Square { /* Even more interesting methods... */ }
}
```

A MyShapes névtér a típusok fogalmi „tárolójaként” működik. A másik lehetőség az, ha a névtérrel több C#-fájlra osztjuk. Ennek érdekében az osztálydefiniciókat ugyanabban a névtérben kell elhelyeznünk:

```
// circle.cs
using System;
namespace MyShapes
{
    // Circle osztály
    class Circle { }
}

// hexagon.cs
using System;
namespace MyShapes
{
    // Hexagon osztály
    class Hexagon { }
}

// square.cs
using System;
namespace MyShapes
{
    // Square osztály
    class Square { }
}
```

Ha egy névtérben egy másik névtér szeretne objektumot használni, a using kulcsszót a következőképpen alkalmazhatjuk:

```
// Használjuk a MyShape névtérben definiált típusokat.
using System;
using MyShapes;
```

```

namespace MyApp
{
    class ShapeTester
    {
        static void Main(string[] args)
        {
            Hexagon h = new Hexagon();
            Circle c = new Circle();
            Square s = new Square();
        }
    }
}

```

A típus teljesen meghatározott neve

Nincs felírtlenül szükség a `C#` using kulcsszavának használatára, ha külső névtérben definiált típust határozunk meg. Ehelyett alkalmazhatjuk a típus teljesen meghatározott nevét is: ez a típus neve, amely elé az öt definiáló névtér nevét fűzzük (lásd 1. fejezet):

```
// Figyeljünk meg, hogy nem "használjuk" a MyShapes névtérét.
using System;
```

```

namespace MyApp
{
    class ShapeTester
    {
        static void Main(string[] args)
        {
            MyShapes.Hexagon h = new MyShapes.Hexagon();
            MyShapes.Circle c = new MyShapes.Circle();
            MyShapes.Square s = new MyShapes.Square();
        }
    }
}

```

Rendszerint nem szükséges a teljesen meghatározott nevet használni. Nemcsak több gépelést igényel, de sem a fordított kód méretét, sem a futási időt nem csökkent. Igazából a köztes nyelvi kódban a típusok *mindig* a teljesen meghatározott névvel szerepelnek. Ebből következően a `C#` using kulcsszavával csupán gépelési időt takaríthatunk meg.

A teljesen meghatározott nevek azért hasznosak (és időnként nélkülözhetlenek), mert segítségével elkerülhetők a névtérkezőések, amelyek több névtér használatkor úgy fordulhatnak elő, ha a névtérekben a típusok neve megegyezik. Tételizzzuk fel, hogy az új névtér neve `My3DShapes`, és három osztályt definiál, amelyek három dimenzióban jelentik meg az alakzatokat:

```
// Egy másik shapes névtér...
using System;

namespace My3DShapes
{
    // 3D Circle osztály
    class Circle { }
    // 3D Hexagon osztály
    class Hexagon { }
    // 3D Square osztály
    class Square { }
}
```

Ha a ShapeTester osztályt a lentiék szerint módosítjuk, az eredmény több fordítási hiba lesz, ugyanis mindkét névtér azonos névvel rendelkező típusokat definiál:

```
// Bőven van kétértelmezés!
using System;
using MyShapes;
using My3DShapes;

namespace MyApp
{
    class ShapeTester
    {
        static void Main(string[] args)
        {
            // Melyik névtérre hivatkozom?
            Hexagon h = new Hexagon(); // Fordítási hiba!
            Circle c = new Circle(); // Fordítási hiba!
            Square s = new Square(); // Fordítási hiba!
        }
    }
}
```

A félreérthetőséget a típus teljesén meghatározott névvel lehet megszüntetni:

```
// Megoldottuk a félreérthetőséget.
static void Main(string[] args)
{
    My3DShapes.Hexagon h = new My3DShapes.Hexagon();
    My3DShapes.Circle c = new My3DShapes.Circle();
    MyShapes.Square s = new MyShapes.Square();
}
```

Definiálás álnemek használatával

A C# using kulcsszavának segítségével elkészíthetjük a típus teljesén meg-határozott nevének álnemét. Ezzel definiálhatunk egy token-t, amely a fordítás során a típus teljes nevére cserélődik:

```
using System;
using MyShapes;
using My3DShapes;

//A félreérthetőség feloldása egyedí álnév használatával.
using The3DHexagon = My3DShapes.Hexagon;

namespace MyApp
{
    class ShapeTester
    {
        static void Main(string[] args)
        {
            // Ez valóban egy My3DShapes.Hexagon típust hoz létre.
            The3DHexagon h2 = new The3DHexagon();
            ...
        }
    }
}
```

A using kulcsszó ezen szintaxisával egy hosszú névterhez is készíthetünk álnemét. Az alaposztálykönyvtárban az egyik hosszabb névter a System.Run-time.Serialization.Formatter, amely magában foglalja a BinaryFormatter tagot. Igény szerint a következőképpen készíthetjük el a BinaryFormatter egy példányát:

```
using MyAlias = System.Runtime.Serialization.Formatter;

namespace MyApp
{
    class ShapeTester
    {
        static void Main(string[] args)
        {
            MyAlias.BinaryFormatter b = new MyAlias.BinaryFormatter();
        }
    }
}
```

illetve a hagyományos using direktívát is alkalmazhatjuk:

```
using System.Runtime.Serialization.Formatters.Binary;

namespace MyApp
{
    class ShapeTester
    {
        static void Main(string[] args)
        {
            BinaryFormatter b = new BinaryFormatter();
        }
    }
}
```

Megjegyzés A C# biztosít egy másik mechanizmust is az azonos nével rendelkező névterek közötti utközések feloldására, amelyhez a névtér alnévminősítőjét (::) és a globális tokenet kell használnunk. Szerencsére az ilyen típusú névutközések ritkán fordulnak elő. Erről bővebb információk a http://msdn.microsoft.com/webhelyen/a_„Working_with_the_C#_2.0_Command_Line_Compiler”_című_cikkemben_találhatók.

Beágyazott névterek létrehozása

Típusaink szervezésekor szabadon definiálhatunk névtereket más névterekben. A .NET-alapozzállykönyvtárak több helyen alkalmazzák ezt a módszert, így biztosítják a típusok szervezésének mélyebb szintjét. A `collections` névtér például a `system.nvtérben` találjuk, tehát a `system.collections.nvtér` áll a rendelkezésünkre. Ha szeretnénk gyököknévtérrel készíteni, amely tartalmazza a `My3DShapes` névtér, a forráskódot a következőképpen módosíthatjuk:

```
// Névtér beágyazása.
namespace Chapter15
{
    namespace My3DShapes
    {
        // 3D Circle osztály
        class Circle { }
        // 3D Hexagon osztály
        class Hexagon { }
        // 3D Square osztály
        class Square { }
    }
}
```

A gyökérnévtér szerepe legtöbbszor csak annyi, hogy újabb hatókörszintet biztosít, és ilyenkor a saját hatókörében közveletlenül nem definiál típusokat (mint például a chapter15 névtér). Ekkor a beágyazott névtér az alábbi tömör formában definiálhatjuk:

```
// Névtér beágyazása (második eset).
namespace chapter15.My3DShapes
{
    // 3D Circle osztály
    class Circle { }
    // 3D Hexagon osztály
    class Hexagon { }
    // 3D Square osztály
    class Square { }
}
```

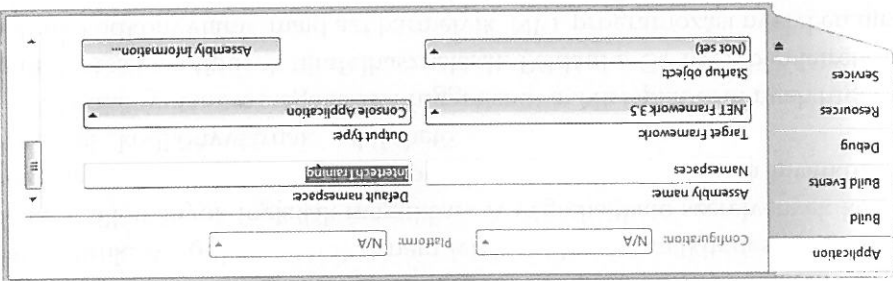
Miután a chapter15 gyökérnévtérbe ágyaztuk a My3DShapes névtér, a létező using direktívákat és típusállandókat kell módosítanunk:

```
using chapter15.My3DShapes;
using The3DHexagon = chapter15.My3DShapes.Hexagon;
```

A Visual Studio 2008 „alapértelmezett névtére”

Az alapértelmezés szerint a Visual Studio 2008 segítségével létrehozott új C#-projektekből az alkalmazás alapértelmezett névtérének és a projektnek a neve megkegyezik. Ha a Project > Add New Item paranccsal új elemeket illesztünk a projekthez, a rendszer a típusokat automatikusan az alapértelmezett névtérben helyezi el. Ha módosítani szeretnénk az alapértelmezett névtér nevét (például a cégünk nevére), a projekt Properties ablakában az Application fülön válasz-

szuk a Default namespace lehetőséget.



15.1. ábra: Az alapértelmezett névtér beállítása

A módosítást követően a projektbe illesztett új elemeket a rendszer az inter-techtraning névtérben helyezi el (ha másik névtérben lenne szükség ezekre a típusokra, a megfelelő using direktívát kell alkalmazni).

Forráskód A Namespaces projekt megtalálható a 15. fejezet alkönyvtárában.

A .NET-szerelvények szerepe

A .NET-alkalmazások tesztöléges számú *szerelvény* összeillesztésével készülnek. A szerelvényt olyan, verzióval rendelkező, önleíró bináris fájlnak tekint-hetjük, amelyet a CLR hosztol. Annak ellenére, hogy a .NET-szerelvények ugyanazzal a fájlkiterjesztéssel (* .exe vagy * .dll) rendelkeznek, mint a ko-rábbi Win32-binárisok (a korábbi COM-kiszolgálókat is beleértve), valójában kevés közös tulajdonsággal rendelkeznek.

A szerelvények a kód újrafelhasználását segítik

Az előzőekben a parancssori alkalmazások készítésekor úgy tűnhetett, hogy az alkalmazások *összes* funkcionálitását a felépíteni kívánt végrehajtható sze-relvény foglalta magában. Valójában az alkalmazások kihasználják a folya-matosan hozzáférhető .NET-kódkönyvtár, az mscorlib.dll (a C#-fordító auto-matikusán hivatkozik az mscorlib.dll könyvtárra) és néhány esetben a sys-tem.windows.forms.dll több típusát.

A *kódkönyvtár* (más néven *osztálykönyvtár*) egy olyan * .dll fájl, amely a külső alkalmazás által használni kívánt típusokat foglalja magában. Amikor végrehajtható szerelvényeket készítünk, az alkalmazás létrehozása során két-szégletlenül több, a rendszer által biztosított, valamint egyedi kódkönyvtárat használunk. A kódkönyvtárakat nem feltétlenül * .dll fájlkiterjesztéssel ren-delkező állományok foglalják magukban. A végrehajtható szerelvények külső végrehajtható fájlokban definiált típusokat használhatnak. Így a hivatkozott * .exe fájl „kódkönyvtárnak” tekinthető.

A kódkönyvtár csomagolásától függetlenül a .NET platform nyelvtüggel-lenül biztosítja a típusok újrafelhasználását. Például a C# nyelvben létrehoz-hatunk kódkönyvtárat, majd azt bármelyik .NET programozási nyelvben újra-alkalmazhatjuk. Más nyelvben írt osztályokat használhatunk, sőt származtat-hatunk is belőlük. A C# nyelvben definiált össosztályt a Visual Basic nyelvben

szerkesztett osztály kiterjesztheti. A Pascal .NET nyelvből definiált interfészeket C# nyelvből definiált struktúrák valósíthatják meg, és így tovább. A le-nyel-nyel, hogy amikor egyéltlen masszív végrehajtható fájl több .NET-szerel-vényre kezdünk bontani, a kód újrafelhasználásának *nyelvszemléletes* módját va-lósítjuk meg.

A szerelvények típushatárokat definiálnak

A *típus teljesen meghatározott nevet* a nevéhez (például `console`) illesztett típus-névter- (például `system`) prefixummal képezzük. Tulajdonképpen a szerel-vény, amelyben a típus elhelyezkedik, a típus további azonosítását szolgálja. Ha például két egyedi névvel rendelkező szerelvényünk van (mondjuk a `my-cars.dll` és a `yourcars.dll`), amelyek mindegyike a `sportscar` nevű osztályt magában foglaló névteret (`carlibrary`) definiálja, a .NET univerzumban a rendszer két egyedi típusnak tekintti őket.

A szerelvények verzióval rendelkező egységek

A .NET-szerelvények négy részből álló numerikus verziószámmal rendelkez-nek. Ennek formátuma a következő: `///<major>.<minor>.<build>.<revision>` (ha explicit módon nem határozzunk meg verziószámot, a rendszer automatikusan a `0.0.0.0` verziószámot rendeli a szerelvényhez). Ez a szám az opcionális *nyilvános kulcsértékkel* együtt lehetővé teszi, hogy egyetlen gépen ugyanazon szerelvény több verziója is egyúttal létezzen. Hivatalosan azok a szerel-vények, amelyek nyilvános kulcsinformációkat biztosítanak, az *erős névvel* ren-delkező szerelvények. Az erős név alkalmazásával a CLR biztosíthatja, hogy a hívó ügyfél számára a rendszer a szerelvény megfelelő verzióját töltsse be.

A szerelvények önleírók

A szerelvények részben *önleírók*, mivel minden olyan külső szerelvényt rögzí-tenek, amelyeknek a megfelelő működéshez szükségük van a meghatározott szerelvény hozzáféréseéhez. Tehát, ha a szerelvény megkívánja a `system.win-dows.Forms.dll` és a `system.drawing.dll` használatát, a *szerelvénymanifesztum* dokumentálja ezeket. A manifesztum a szerelvényt leíró metadatok (név, verzió, szükséges külső szerelvények stb.) összessége (lásd 1. fejezet).

A manifesztum adatain kívül a szerelvény metaadatok is tartalmaz, amelyek ismeretük az egyes típusok összetételét (tagneveket, megvalósított interfészeket, alaposztályokat, konstruktorokat és így tovább). Mivel a szerelvényeket a rendszer ilyen részletességgel dokumentálja, a CLR *nem* fordul a Win32 rendszerleíró adatbázishoz a szerelvények elérési útjainak feloldásakor (ez elég radikális eltérés a Microsoftt korábbi COM programozási modelljéhez képest), hanem teljesen új módszert alkalmaz a külső kódkönyvtárak helyének feloldása során.

A szerelvények konfigurálhatók

A szerelvényeket „privát” vagy „megosztott” szerelvényként telepíthetjük. A privát szerelvények ugyanabban a könyvtárban (illetve alkönyvtárban) helyezkednek el, mint amelyben az ügyfélalkalmazás használja őket. Ezzel szemben a megosztott szerelvények olyan könyvtárak, amelyeket egyetlen számtalig több alkalmazás használ, és meghatározott könyvtárba, a *globalis szerelvénytárba* (GAC) kell telepíteni őket.

A szerelvények telepítésétől függetlenül szabadon készíthetünk XML-alapú konfigurációs fájlokat. A konfigurációs fájlok segítségével utasíthatjuk a CLR-t, hogy meghatározott helyen keresse a szerelvényeket, töltsse be az ügyfél számára a kijelölt szerelvény adott verzióját, vagy a helyi számtaligep tetszőleges könyvtárban, hálózati helyen vagy URL-en kutasson. A következőkben sok újdonságot megtudhatunk az XML konfigurációs fájlokkal kapcsolatban.

A .NET-szerelvények felépítése

Az alábbiakban a szerelvények összetevőit vizsgáljuk meg. Szerkezeti szempontból a .NET-szerelvény (*.dll vagy *.exe) a következő elemeket tartalmazza:

- Win32-fejlec
- CLR-fejlec
- köztes nyelvi kód

- típusmetaadat

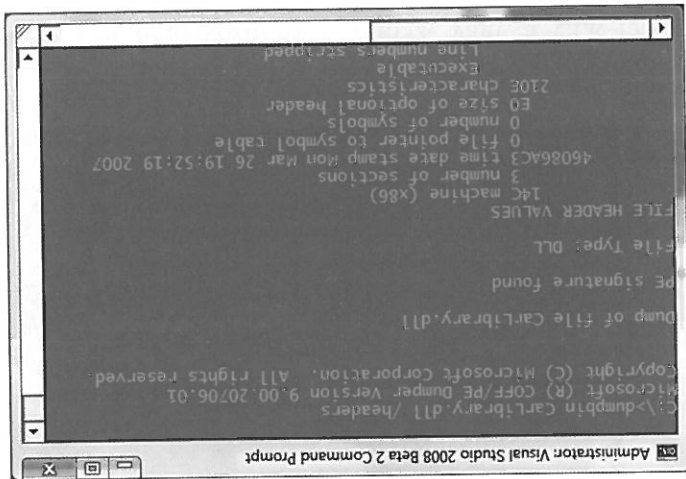
- a szerelvénymanifesztum

- opcionális beágyazott erőforrások

Az első két elem (a Win32- és a CLR-fejlec) tulajdonképpen adatblokk, ezeket rendszerszintű figyelmen kívül szoktuk hagyni, ám röviden mégis érdemes át-tanulmányozni őket.

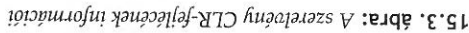
A Win32-fejlec

A Win32-fejlec mutatja meg, hogy a szerelvényt a Windows operációs rendszer család tagjai betölthetik, és dolgozhatnak vele. A fejleccadatok azonosítják az alkalmazástípust (konzolalapú, grafikusfelület-alapú vagy *.dll kódkönyvtár), amelyet a Windows operációs rendszer hosztol. Ha a dumpbin.exe segédprogrammal (a Visual Studio 2008 parancssorán keresztül) megnyitjuk a .NET-szerelvényt, a /headers kapcsolóval megnézhetjük a szerelvény Win32-fejleccadait. A 15.2. ábrán a készíttendő carl.library.dll szerelvény Win32-fejleccadait (illetve az adatok egy része) láthatók.



15.2. ábra: A szerelvény Win32-fejleccének információi

szerelevény belül CLK-típus-információi jelennek meg (lásd a 15.3. ábrán).



.NET-fejlesztésként nem kell foglalkoznunk a Win32- vagy a CLR-fejlesztéssel, hanem a .NET-keretrendszerrel foglalkozhatunk. Csak annyit érdemes megjegyezni, hogy minden .NET-szerelvény magában foglalja ezeket az adatokat, amelyek a háttérben a .NET-futtatórendszer és a Windows operációs rendszer számára szükségesek.

erőforrásokat felosztani nemzettközi programok készítésekor. A szatellitizálás-
vényekre ebben a könyvben nem térek ki, a Windows Presentation Foundation
vizsgálatakor azonban megnézzük, hogyan lehet alkalmazás-erőforrásokat a
szerelevényekbe ágyazni.

Egyfájlos és többfájlos szerelevények

Technikailag a szerelevény több *modulból* épülhet fel. A modul valójában nem
más, mint egy érvényes .NET bináris fájl általános kifejezése. Ha a szerelevény
egyetlen modulból áll, akkor a (logikai) szerelevény és a mögöttes (fizikai) bi-
náris között egy-egy megfelelés van (innen az *egyfájlos szerelevény* elnevezés).
Az egyfájlos szerelevények az összes szükséges összetevőt (fejlec-információ,
közös nyelvi kód, típusmetaadat, manifesztum és a megfelelő erőforrások)
egyetlen *.exe vagy *.dll csomagban foglalják magukban. A 15.4. ábrán egy
egyfájlos szerelevény összetételét láthatjuk.

Egyfájlos szerelevény
CarLibray.dll

Oníráš
Típusmetaadat
Közös nyelvi kód
(Teljeslegesen) Erőforrások

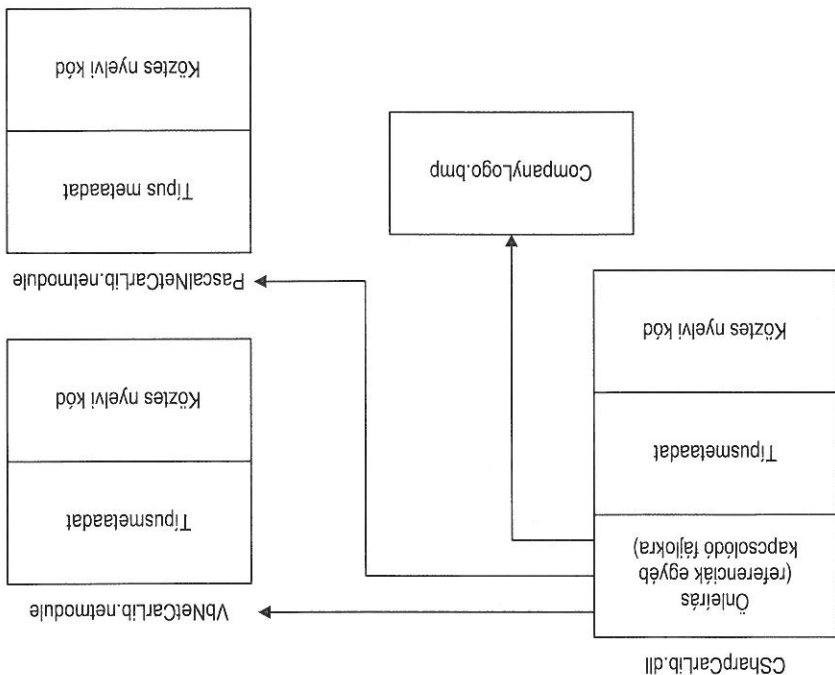
15.4. ábra: Egyfájlos szerelevény

Másrészt a többfájlos szerelevény .NET dll-ek egy olyan halmaza, amelynek
elemei egyetlen logikai egységként települnek, és közös verziószámot kap-
nak. Hivatatosan a *.dll fájlok egyike az *elsődleges modul*, amely magában
foglalja a szerelevényszintű manifesztumot (valamint a közös nyelvi kódot, a
metaadatokat, a fejlec-információkat és az opcionális erőforrásokat). Az el-
sődleges modul manifesztuma rögzíti azokat a kapcsolódó *.dll fájlokat,
amelyektől függ.

Többfajlos szerelvényekben névadási konvenciók alapján a másodlagos modulok *.netmodule kiterjesztést kapnak; ez a CLR esetén azonban nem követelmény. A másodlagos *.netmodule fájlok szintén tartalmaznak köztes nyelvi kódot és típusmetaadatot, valamint a *modulszintű menfeszítumot*, amely meghatározott modul szükségességet külső szerelvényeit rögzíti.

A többfajlos szerelvények a tartalomleltetés rendkívül hatékony módját biztosítják. Tétellezzük fel például, hogy számítógépünk egy három modulból álló, távoli többfajlos szerelvényre hivatkozik, ahol az elsődleges modult az ügyfélprogramra telepítettük. Ha az ügyfélprogramnak a másodlagos távoli *.netmodule egyik típusára van szüksége, a CLR igény szerint leölti a bináris fájlt a helyi számítógép egy speciális helyére, a *letöltési gyorsítótárba*. Ha az összes *.netmodule fájl mérete 5 megabyte, a letöltés gyorsítótár válik az előnyűk az egyetlen 15 megabyte-os fájllétoítésével szemben.

Többfajlos szerelvény



15.5. ábra: Az elsődleges modul a szerelvény onírásában rögzíti a másodlagos modulokat

A többfajlos szerelvények másik haszna, hogy a modulok készítését több .NET programozási nyelv segítségével biztosítják (ennek nagyvállalatok esetén lehet jelentősége, ahol az egyes osztályok meghatározott .NET-nyelv használatát részesítik előnyben). Az egyes modulok fordítását követően a C# parancssoros fordítóval egyetlen logikai szerelvényben kapcsolhatjuk össze a modulokat.

A többfajlos szerelvényt alkotó modulokat *nem* a szó szoros értelmében kapcsolja össze a rendszer egyetlen (nagyobb) fájlban. Ehelyett a szerelvényeket az elsődleges modul manifesztumában a logikailag kapcsolódó információk kötik össze. A 15.5. ábrán három modulból álló többfajlos szerelvény látható, amelyben a modulok mindegyike egyedi .NET programozási nyelv alkalmazásával készült.

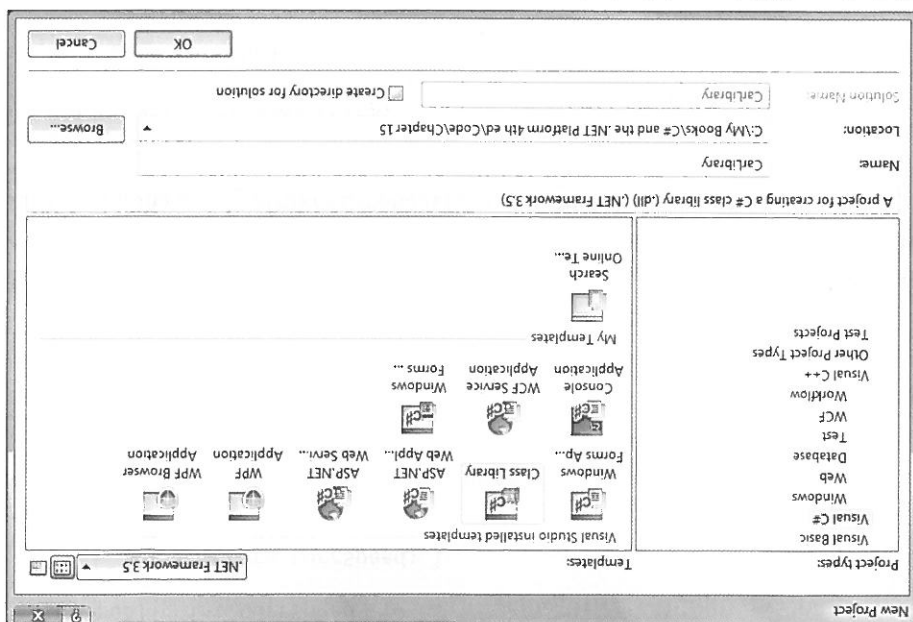
A továbbiakban a kódkönyvtár-készítés és -beállítás részeit vesszük szemügyre.

Egy állományból álló szerelvény készítése és használata

A .NET-szerelvények világának felfedezéséhez először készítsünk egy olyan egyfajlos *.dll szerelvényt (carl.library nével), amely a nyilvános típusok kisebb gyűjteményét foglalja magában. Ha a Visual Studio 2008 segítségével szerelnünk létrehozni a kódkönyvtárat, egyszerűen válasszuk a class library projekt típust (lásd a 15.6. ábrát).

A carl.library tervezése a car nevű absztrakt össztállyal kezdődik, amely egyedi tulajdonságok révén rendelkezésre álló védett adattagokat definiál (igény szerint szabadon alkalmazhatjuk az automatikus tulajdonság szintaxist, lásd a 13. fejezetet). Az osztály egyetlen absztrakt metódussal, a turboBoost() metódussal rendelkezik, amely egyedi felsorolás (EngineState) segítségével mutatja be a gépjármű motorjának aktuális állapotát.

15.6. ábra: C#-kódkönyvtár létrehozása



```

    public int CurrSpeed
    {
        get { return currSpeed; }
        set { currSpeed = value; }
    }
    public int MaxSpeed
    {
        get { return maxSpeed; }
        public EngineState EngineState
        { get { return engnState; } }
    }
}

```

Feltételezzük, hogy a car típusnak két közvetlen leszármazottja van, a `miniVan` és a `Sportscar` osztály. A megfelelő üzenet megjelenítésével mindkét osztály felüldefiniálja az `absztrakt TurboBoost()` metódust.

```

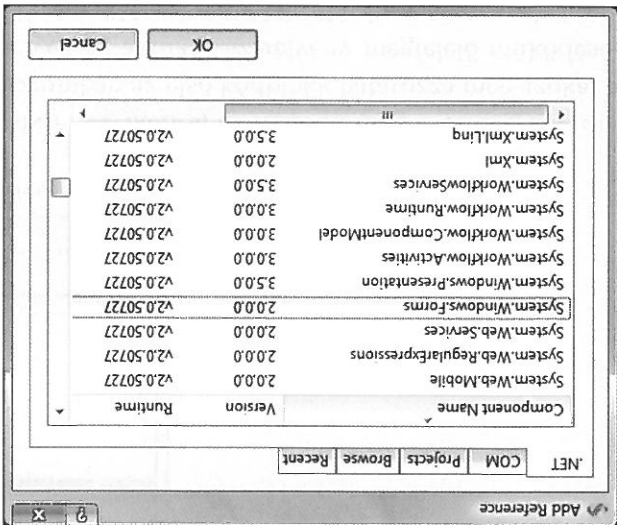
using System;
using System.Windows.Forms;

namespace CarLibrary
{
    public class Sportscar : car
    {
        public Sportscar(){}
        public Sportscar(string name, int max, int curr)
            : base (name, max, curr){ }
        public override void TurboBoost()
        {
            MessageBox.Show("Ramming speed!", "Faster is better...");
        }
    }

    public class miniVan : car
    {
        public miniVan(){}
        public miniVan(string name, int max, int curr)
            : base (name, max, curr){ }
        public override void TurboBoost()
        {
            // A minivanek gyenge turbóképeségeket rendelkezik!
            engnState = EngineState.engnDead;
            MessageBox.Show("Time to call AAA", "Your car is dead");
        }
    }
}

```

A származtatott osztályok a windows Forms messagebox osztályának a segítségével valóstítják meg a TurboBoost() metódust. Az osztályt a system.windows.forms.dll szerelvény definiálja. Ahhoz, hogy a szerelvény a külső szerelvényben definiált típusokat alkalmazhassa, a carLibrary projekteknek hivatkozni kell a bináris fájlra az Add Reference párbeszédpanellel (lásd a 15.7. ábrát), amelyhez a Visual Studio Projects > Add Reference menüponton keresztül férhetünk hozzá.



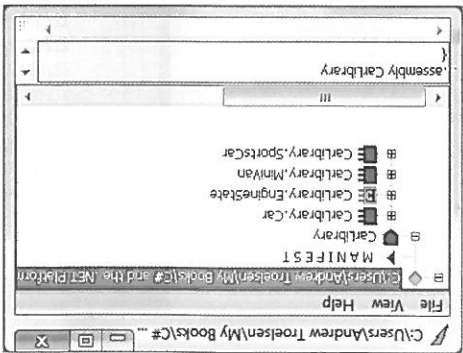
15.7. ábra: A külső .NET-szerelvények hivatkozásának kezdete

Nagyon fontos tudnivaló: az Add Reference párbeszédpanel .NET-fülsen megjelenő szerelvények nem képviselik a számítógépünkön tárolt összes szerelvényt. Az Add Reference párbeszédpanelben *nem* jelennek meg az egyedi szerelvények, és a panel *nem* mutatja a globális szerelvénytárban (GAC) található szerelvényeket sem. Ehelyett a párbeszédpanel a Visual Studio 2008 előreprogramozott általános szerelvényeit listázza. Ha olyan alkalmazást készítünk, amely az Add Reference párbeszédpanelben nem látható szerelvényt használ, a Browse fülon manuálisan kell a kérdéses *.dll vagy *.exe fájlhoz navigálnunk.

Megjegyzés Ne feledkezzünk meg róla, hogy az Add Reference párbeszédpanel Recent fülen megtalálható a korábban hivatkozott szerelvények listája. Praktikus lehetőség, hiszen sok .NET-projekt a külső könyvtárak ugyanazon központi halmazzát alkalmazza.

A manifesztum

Mielőtt az ügyfélalkalmazásból használnánk a CarLibrary.dll fájlt, ellenőriz-
zük a kódkönyvtár belső felépítését. Tegyük fel, hogy lefordítottuk a prog-
tet, majd ezt követően töltjük be a CarLibrary.dll fájlt az ilasm.exe prog-
ramba (lásd a 15.8. ábrát).



15.8. ábra: Az ilasm.exe programban megjelentetett CarLibrary.dll

Kattintsunk kétszer a MANIFEST ikonra, és nyissuk meg a CarLibrary.dll
manifesztumát. A manifesztumban az első kódblokk határozza meg azokat a
külső szerelvényeket, amelyek az aktuális szerelvény megfelelő működésé-
hez szükségesek. A CarLibrary.dll az mscorlib.dll és a System.Windows.
Forms.dll szerelvények típusait alkalmazta, ezek mindegyikét a manifesztum
rögzíti a .assembly extern kulcsszó segítségével:

```
.assembly extern mscorlib  
{  
  .pubtoken = (B7 7A 5C 56 19 34 E0 89 )  
  .ver 2:0:0:0  
}  
.assembly extern System.Windows.Forms  
{  
  .pubtoken = (B7 7A 5C 56 19 34 E0 89 )  
  .ver 2:0:0:0  
}
```

Mind egyik .assembly extern blokkot a .pubtoken és a .ver direktíva mi-
nősíti. A .pubtoken utasítás csak akkor jelenik meg, ha a szerelvényt *erős*
nével állítottuk be (az erős nevekkel kapcsolatos részleteket lásd később).
A .ver token (természetesen) a numerikus verziót azonosítja.

A külső referenciák katalogizálása után a szerelvény szintű attribútumokat azonosító egyedi tokeneket találjuk. Ha megvizsgáljuk a Visual Studio AssemblyInfo.cs fájlt (amelyet a Solution Explorer Properties ikonjának kibővítésével nézhetünk meg), látni fogjuk, hogy az attribútumok a szerelvény olyan alapvető jellemzőit képviselik, mint például a vállalat neve, védjegye és így tovább.

Az AssemblyInfo.cs fájlban definiált attribútumok különböző .custom tokeneket (mint az [AssemblyTitle]) módosítanak az önleírásban (az attribútumokat lásd a 16. fejezetben):

```
.assembly CarlLibrary
{
    ...
    .custom instance void [mscorlib]
        System.Reflection.AssemblyTitleAttribute::ctor(string) =
        ( 01 00 0A 43 61 72 4C 69 62 72 61 72 79 00 00 )
        // ...CarlLibrary..
        .hash algorithm 0x00008004
        .ver 1:0:0:0
}
.module CarlLibrary.dll
```

Az .assembly token jelzi a szerelvényünk barátságos nevét (CarlLibrary), míg a .module token meghatározza a modul nevét (CarlLibrary.dll). Az AssemblyInfo.cs [AssemblyVersion] attribútumának megfelelően a .ver token meghatározza a szerelvényhez rendelt verziószámot.

A köztes nyelvi kód

A szerelvény nem tartalmaz platformfüggő gépi utasításokat; ehelyett platformfüggetlen köztes nyelvikód- (CIL-) utasításokat foglal magába. Amikor a .NET-futtatórendszer a memóriába tölti a szerelvényt, a rendszer a köztes nyelvi kódot (a JIT-fordító segítségével) a célplatform számára érthető utasításokra fordítja. Ha a sportscart osztály turboboot() metódusára két szerelvényünk, az idasm.exe új ablakban megjeleníti a CIL-kulcsszavakat, amelyek megvalósítják az alábbi metódust:

```
.method public hidebysig virtual instance void
    Turboboot() cil managed
{
    // code size 18 (0x12)
    .maxstack 8
    IL_0000: nop
```

```
IL_0001: ldstr "Ramming speed!"
IL_0006: ldstr "Faster is better..."
IL_000b: call valuetype
[System.Windows.Forms.DialogResult]
[System.Windows.Forms.MessageBox]
::Show(string, string)
IL_0010: pop
IL_0011: ret
} // a Sportscar::TurboBoost metódus vége
```

A `method` címke azonosítja a `sportscar` típusal definiált metódust. A típus-sok által definiált tagváltozókat a `field` címke jelöli. A `car` osztály védett adatok halmazát definiálta, például a `currspeed` változót:

```
.field family int32 currspeed
```

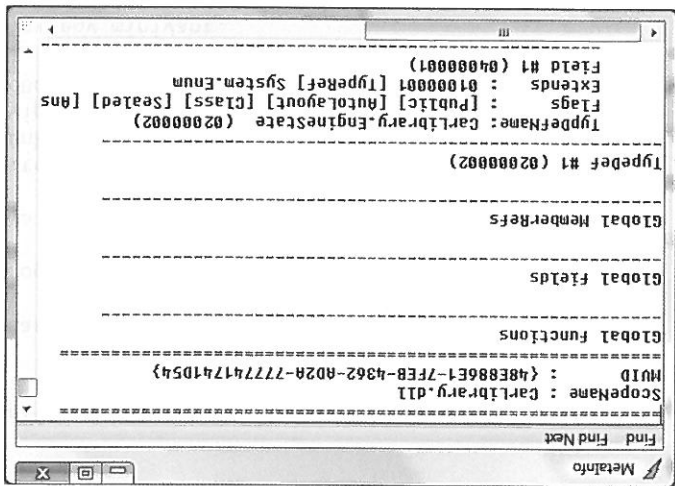
A tulajdonságokat a `property` címke jelzi. A nyilvános `currspeed` tulajdonság-`get` leíró köztes nyelvi kód a következő (a tulajdonság lekérdező/módosító blokkját a `get` és a `set` címkek jelölik):

```
.property instance int32 currspeed()
{
    .get instance int32 CarLibary.Car::get_Currspeed()
    .set instance void CarLibary.Car::set_Currspeed(int32)
} // a Car::Currspeed tulajdonság vége
```

A tulajdonság `get/set` blokkja normál (és rejtett) metódusokat tartalmaz meg a szerelevényen belül (ebben az esetben a `get_Currspeed()` és a `set_Currspeed()` metódusokat). (A köztes nyelvi kód szintaxisát és szemantikáját lásd a 19. fejezetben.)

A típusmetaadat

Ha megnyomjuk a `Ctrl + M` billentyűkombinációt, az `Idasm.exe` megjelenti a `CarLibary.dll` szerelevényben az összes típus metaadatait (lásd a 15.9. ábrát). A szerelevény-metaadatok a .NET platform rendkívül fontos jellemzői, és több technológia (objektumsorosítás, késői kötés, kiterjeszthető alkalmazások stb.) gerincéeként szolgálnak (lásd a 16. fejezetben). A `CarLibary.dll` szerelevény felépítése után készítsünk el néhány olyan ügyfélalkalmazást, amely a típusainkat használja.



15.9. ábra: A CarLibray.dll típusainak metaadatai

Forráskód A CarLibray projekt megtalálható a 15. fejezet alkönyvtárában.

C#-ügyfélalkalmazás készítése

Mivel a CarLibray típusok mindegyikét a public kulcsszóval deklaráltuk, más szerelvények is használhatják a típusokat. A típusokat a C# internal kulcsszó alkalmazásával is definiálhatjuk (valójában ez az alapértelmezett C# hozzáférési mód). A belső típusokat kizárólag az a szerelvény használhatja, amelyben azokat definiáltuk. A külső ügyfélalkalmazások az internal kulcszóval megjelölt típusokat nem láthatják és nem is példányosíthatják.

Megjegyzés A .NET biztosítja a „barátságos szerelvények” meghatározását. Ezek lehetővé teszik, hogy a belső típusokat szerelvények meghatározott csoportja használhassa. A részletek a .NET Framework 3.5 SDK dokumentációjában az internalvisibilityattribute osztály leírásánál találhatók.

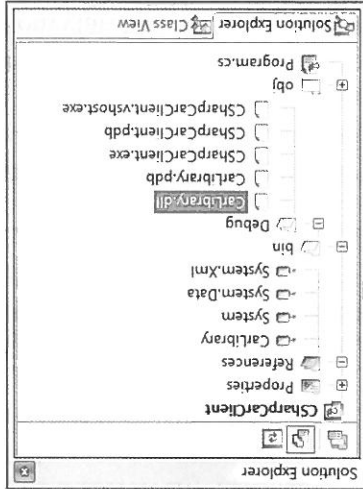
A típusok alkalmazásához készítsünk új parancssoralkalmazás-projektet (csharpconsole). Ezután, az Add Reference párbeszédpanel Browse fülének segítségével állítsunk be egy referenciát a CarLibray.dll szerelvényre (ha a Visual Studio programmal fordítottuk a CarLibray.dll szerelvényt, akkor a CarLibray projekt könyvtár \bin\Debug alkönyvtárában találjuk a fájlt). Ekkor elkezdhetjük az ügyfélalkalmazás építését, amely külső típusokat fog használni. Módosítsuk a kiindulási C#-fájlt a következőképpen:

```
using System;

// Ne felejtünk el importálni a CarLibrary névteret!
using CarLibrary;
```

```
namespace CSharpCarClient
{
    public class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** C# CarLibrary Client App *****");
            // készítsünk egy sportautót.
            Sportscar viper = new Sportscar("viper", 240, 40);
            viper.TurboBoost();
            // készítsünk egy minivan-t.
            Minivan mv = new Minivan();
            mv.TurboBoost();
            Console.ReadLine();
        }
    }
}
```

A kód pontosan olyan, mint az eddig fejlesztett alkalmazásaink kódja. Az egyetlen érdekes pont, hogy a C#-ügyfélalkalmazás külön egyedi szerver-venyben definiált típusokat használ. Ezután futtassuk a programot. Várakozásainknak megfelelően a program futtatásával több üzenetdoboz jelenik meg a képernyőn.



15.10. ábra: A Visual Studio 2008 a privát szerelevényeket az ügyfélkönyvtárba másolja

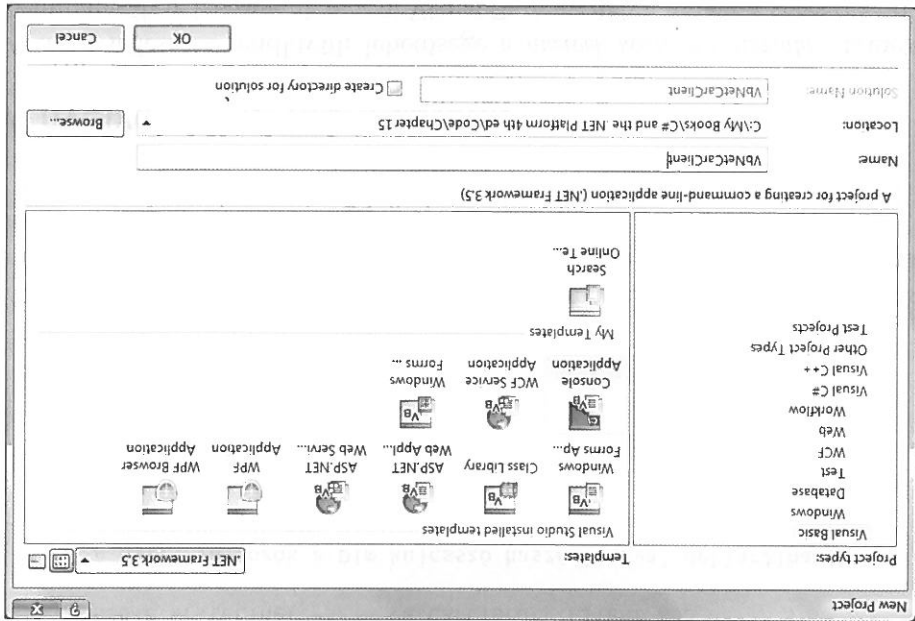
A Visual Studio 2008 a carLibrary.d11 fájlt másolatát elhelyezte a csHarpCarC1ent projekt könyvtár \bin\Debug könyvtárban. Ezt ellenőrizhettük, ha a Solution Explorer Show All Files gombjára kattintunk (lásd a 15.10. ábrát).

A későbbiekben látni fogjuk, hogy a carLibrary.d11 szerverelvényt a rendszer „privát” szerverelvényként telepítette. Ezért a CLR az aktuális ügyfélalkalmazás (csHarpCarC1ent.exe), a .NET-bináris helyi másolatát tölti be.

Forráskód A csHarpCarC1ent projekt megtalálható a 15. fejezet alkönyvtárában.

Visual Basic ügyfélalkalmazás készítése

A .NET platform nyelvfüggetlen viselkedésének bemutatására készítsünk egy másik parancssori alkalmazást (vbNetCarC1ent), ezúttal a Visual Basic használatával (lásd a 15.11. ábrát). A projekt létrehozása után állítsunk be egy referenciát a carLibrary.d11 fájlra az Add Reference párbeszédpanel segítségével. Ehhez a Project > Add Reference menüponton keresztül férhetünk hozzá.



15.11. ábra: Visual Basic parancssori alkalmazás készítése

A C# nyelvhez hasonlóan a Visual Basic is megköveteli, hogy az aktuális fájlban használt névtéreket felsoroljuk. A Visual Basicben az Imports kulcsszó használatos a C# using kulcsszáva helyett. Ígyuk tehát a következő Imports utasítást a Modul1.vb kód fájlba:

```
Imports CarLibrary
```

```
Module Modul1
    Sub Main()
        End Sub
    End Module
```

A main() metódust a Visual Basic modul típuson belül definiáljuk (ennek semmi köze nincs a többfájlos szerelvény *.netmodule fájljához). Röviden: a modulokkal a Visual Basic a lezárt osztályokat jelöli, amelyek kizárólag statikus metódusokat foglalnak magukban. Hogy a Visual Basic szintaxisával gyakoroljuk a minivan és a sportscar típusokat, módosítsuk a main() metódust a következőképpen:

```
Sub Main()
    Console.WriteLine("***** VB CarLibrary Client App *****")
    ' A helyi változók a Dim kulcsszó használatával deklarálhatók.
    Dim myMiniVan As New MiniVan()
    myMiniVan.TurboBoost()
    Dim mySportscar As New Sportscar()
    mySportscar.TurboBoost()
    Console.WriteLine()
End Sub
```

Az alkalmazás fordítása és futtatása során ismét üzenetdobozok jelennek meg a képernyőn. Az új ügyfélalkalmazás a carlibrary.d11 fájl saját helyi mäsolatát tárolja a bin\Debug könyvtárban.

A nyelvek közti származtatás

A .NET-fejlesztés rendkívüli lehetősége a *nyelvek közti származtatás*. Ennek bemutatásához készítsünk egy új Visual Basic osztályt, amely a sportscar típusból származik (ezt a C# nyelv segítségével hoztuk létre). Először adjunk egy új osztályfájlt Performancecar.vb névvel az aktuális Visual Basic alkalmazásunkhoz (válasszuk a Project > Add Class menüpontot).

Módosítsuk a kiindulási osztály definícióját: az `Inherits Sportscar` segítségével származtatassuk az osztályt a `Sportscar` típusból. Továbbá definiáljuk felül az `absztrakt TurboBoost()` metódust az `overrides` kulcsszó használatával:

```
Imports CarLibrary

' Ez a VB-típus a C# Sportscar típusból származik.
Public Class PerformanceCar
    Inherits Sportscar
    Public Overrides Sub TurboBoost()
        Console.WriteLine("Zero to 60 in a cool 4.8 seconds...")
    End Sub
End Class
```

Az új osztálytípus teszteléséhez a következőképpen módosítsuk a `modul main()` metódusát:

```
Sub Main()
    ...
    Dim dreamCar As New PerformanceCar()

    ' Használjuk az inherited tulajdonságot.
    dreamCar.PetName = "Hank"
    dreamCar.TurboBoost()
    Console.ReadLine()
End Sub
```

A `dreamCar` objektum a származási hierarchia bármely nyilvános tagját hívhatja (például a `PetName` tulajdonságot), függetlenül attól, hogy az osztályt más nyelven és más szerverelvényben definiáltuk.

Forráskód A `VbNetCarCLI` ent projekt megtalálható a 15. fejezet alkönyvtárában.

Több állományból álló szerverelvény készítése és használata

A következőkben a többfájlos szerverelvények készítésének folyamatait vesszük szemügyre. A többfájlos szerverelvény olyan kapcsolódó modulok gyűjteménye, amelyeket egyetlen logikai egységként telepítünk, és látunk el verziószám-mal. Jelenleg a Visual Studio IDE nem támogatja a C# többfájlos szerverelvény projekttsablonját. Így tehát a parancssoros fordítót (`csc.exe`) kell használnunk.

A folyamat bemutatásához hozzuk létre az `AirVehicle`-es többfajlos szerelvényt. Az elsődleges modul (`AirVehicle.cs`, `dl1`) kizárólag a `Helicopter` nevű osztálytípust foglalja magában. A kapcsolódó manifesztumok (ez szintén az `AirVehicle.cs`, `dl1` fájlban található) egy további `*.netmodule` állományról (`ufo.netmodule`) is tudomása van, amely (természetesen) az `ufo` nevű osztályt tartalmazza. Noha fizikailag mindkét osztálytípust különálló bináris fájlok tartolják, egyetlen névterbe, az `AirVehicle`-esbe csoportosítjuk őket. Végül, mindkét osztályt a `C#` nyelv alkalmazásával hozzuk létre (bár más nyelveket is használhatnánk). Indítsunk el egy szövegszerkesztőt (például a Jegyzettömöt), majd az `ufo.cs` fájlban hozzuk létre a következő `ufo` osztálydefiniót:

```
using System;

namespace AirVehicles
{
    public class ufo
    {
        public void AbductHuman()
        {
            Console.WriteLine("Resistance is futile");
        }
    }
}
```

Ha .NET-modulba szeretnénk fordítani ezt az osztályt, navigáljunk az `ufo.cs` fájl tartalmához könyvtárához, majd gépeljük be a következő parancsot a `C#-fordító` számára (a `/target` kapcsoló modula opciója utasítja a `csc.exe` programot, hogy `*.netmodule`, ne pedig `*.dll` vagy `*.exe` fájlt készítsen):

```
csc.exe /t:module ufo.cs
```

Az `ufo.cs` fájl tartalmához könyvtárban azt tapasztaljuk, hogy új fájl, az `ufo.netmodule` fájl jelent meg (érdemes ellenőrizni). Ezt követően hozzunk létre egy `Helicopter.cs` nevű fájlt, amely a következő osztálydefiniót foglalja magában:

```
using System;

namespace AirVehicles
{
    public class Helicopter
    {
        public void Takeoff()
        {
            Console.WriteLine("Helicopter taking off!");
        }
    }
}
```

Mivel a többfajlos szerelvény elsődleges moduljának tervezett neve `airvehic` és az `airvehic.dll`, a `/t:library` és az `/out: opciókkal` kell a `helicopter.cs` fájlt fordítani. Ha a szerelvény manifestumában szeretnénk megjeleníteni az `ufo.netmodule` binárist, akkor az `/addmodule` kapcsolót kell alkalmaznunk. A következő parancssal véghezvük el a feladatot:

```
csc /t:library /addmodule:ufo.netmodule
/out:airvehic.dll helicopter.cs
```

A parancs végrehajtása után a könyvtár az elsődleges `airvehic` és `ufo.netmodule` binárisokat is tartalmazza.

Az `ufo.netmodule` fájl

Az `ufo.netmodule` fájl a `ufo.netmodule` meg az `ufo.netmodule` fájl. A `*.netmodule-ok` *manifestumot* tartalmaznak, ennek célja pusztán azoknak a külső szerelvényeknek a listázása, amelyekre a forráskód hivatkozik. Ha az `ufo` osztály csak a `console.WriteLine()` metódust hívja meg, akkor a következőket látjuk:

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 2:0:0:0
}
module ufo.netmodule
```

Az `airvehic.dll` fájl

Az `ufo.exe` segítségével nyissuk meg az elsődleges `airvehic.dll` modult, és vizsgáljuk meg a szerelvény szintű manifestumot. A `file` token a többfajlos szerelvény kapcsolódó moduljait sorolja fel (ebben az esetben az `ufo.netmodule` fájl). A `class` külső tokenek segítségével dokumentálhatjuk azoknak a külső típusoknak a nevét, amelyekre a másodlagos modulban (`ufo`) hivatkozunk.

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 2:0:0:0
}
```

```

    {
        .assembly AirvehICLES
        ...
        .hash algorithm 0x00008004
        .ver 0:0:0
    }
    .file ufo.netmodule
    ...
    .class extern public AirvehICLES.Ufo
    {
        .file ufo.netmodule
        ...
        .file ufo.netmodule
        .class 0x02000002
    }
    .module AirvehICLES.d11

```

Az egyetlen elem, amely összekapcsolja az AirvehICLES.d11 és az ufo.netmodule fájlokat, a szerelvény manifestuma. A két bináris fájlt a rendszer nem ísüli egyetlen *.d11 fájlba.

A többfájlos szerelvény használata

A többfájlos szerelvényre hivatkozó más kódokat általában nem érdekli, hogy a szerelvény, amellyel dolgoznak, több modulból épül fel. Ebben a példában egy szerűen parancssorból hozunk létre egy új C#-üggyfélalkalmazást. A következő moduldefinióval készítük el a Client.cs fájlt. Ha készen vagyunk, a fájlt ugyanarra a helyre mentjük, ahova a többfájlos szerelvényt.

```

using System;
using AirvehICLES;

class Program
{
    static void main()
    {
        Console.WriteLine("***** Multifile Assembly Client *****");
        Helicopter h = new Helicopter();
        h.Takeoff();

        // Ez, szükség esetén, betölti a *.netmodule fájlt.
        ufo u = new ufo();
        u.AbductHuman();
        Console.ReadLine();
    }
}

```

A szerelevény lefordításához a C# nyelv parancssori fordítóját, a csc.exe programot használjuk, az alábbi módon:

```
csc /r:airvehicles.dll Client.cs
```

Többfájlos szerelevények hivatkozása során a fordítónak kizárólag az elsődleges modul nevére van szüksége (a *.netmodul-es fájlokat igény szerint a CLR tölti be, ha az ügyfélprogram forráskódja használja őket). Onmagukban a *.netmodul-es fájlok nem rendelkeznek saját verziószámmal, és a CLR közvetlenül nem tudja betölteni ezeket a fájlokat. Az egyedi *.netmodul-es fájlokat csak az elsődleges modul töltheti be (például a szerelevény manifestumát tartalmazó fájl).

Megjegyzés A Visual Studio 2008 lehetővé teszi, hogy többfájlos szerelevényre hivatkozzunk. Az Add Referencés párbeszédpanel segítségével válasszuk ki az elsődleges modult. A kapcsolódó *.netmodul-es fájlok a folyamat során lemasóolódnak.

Nagyon nagy a valószínűsége annak, hogy a szerelevényeink 99,99 százaléka egyfájlos lesz. Ennek ellenére a többfájlos szerelevények hasznosak, ha nagy-méretű bináris fájlt szeretnénk modúláris egységekre bontani (továbbá távoli leöltési helyzetekben praktikusak még). A következőkben a privát szerelevény fogalmát formalizáljuk.

Forráskód A műtífti Leassembly projekt megtalálható a 15. fejezet alkönyvtárában.

Privát szerelevények

A fejezetben eddig elkészített szerelevényeket technikailag *privát szerelevényként* telepítettük. A privát szerelevényeknek ugyanabban a könyvtárban (az ügynevezett *alkalmazáskönyvtárban*) vagy alkönyvtárban kell elhelyezkedniük, mint amelyben az ügyfélalkalmazás található. Amikor a cshtarpcar-Client.exe és a vbnetcarClient.exe készítésékor a referenciát a carLib-rary.dll szerelevényre állítottuk, a Visual Studio 2008 ez követően elhelyezte a carLib-rary.dll másolatát az ügyfélalkalmazás könyvtárában (az első fordítást követően).

Ha az ügyfélprogram az ebben a külső szerelvényben definiált típusokat használja, a CLR egyszerűen betölti a `CarLibrary.dll` helyi másolatát. Mivel a .NET-futtatórendszer a hivatkozott szerelvények keresésekor nem fordul a rendszerleíró adatbázishoz, a `CsharpCarClient.exe` (vagy a `VbnetCarClient.exe`) és a `CarLibrary.dll` szerelvényeket másik könyvtárba helyezhetjük, és futtathatjuk az alkalmazást (ezt nevezzük *Xcopy-telépítésnek*).

Kizárólag privát szerelvényekkel működő alkalmazás eltávolítása (vagy replikálása) értelmelen: egyszerűen töröljük (vagy másoljuk le) az alkalmazás könyvtárát. A COM-alkalmazásoktól eltérően nem kell aggodnunk az elárvult rendszerleíróadatbázis-bejegyzések miatt. Sőt, amint sem kell nyugtalankodnunk, hogy a privát szerelvények eltávolításával a számlító gép más alkalmazásai működésképtelené válnak.

A privát szerelvény azonosítója

A privát szerelvények teljes azonosítóját a barátságos név és a numerikus verzió szolgáltatója, amelyek mindegyikét a szerelvény manifestuma rögzíti. A *barátságos név* egyszerűen annak a modulnak a neve, amely a szerelvény manifestumát tartalmazza, minusz a fájl kiterjesztése. Ha például megvizsgáljuk a `CarLibrary.dll` szerelvény manifestumát, a következőket találjuk:

```
.assembly CarLibrary
{
    ...
    .ver 1:0:0:0
}
```

A privát szerelvények elszigetelt természetéből adódóan nem véletlen, hogy a szerelvény helyének meghatározásakor a CLR nem bajlódik a verziószám-mal. A rendszer azt feltételezi ugyanis, hogy a privát szerelvények verzióját nem kell ellenőrizni, hiszen az ügyfélalkalmazás az egyetlen olyan entitás, amely „tud” a szerelvény létezéséről. Ezért (nagyon) valószínű, hogy egy számlító gép ugyanannak a privát szerelvénynek több másolatát is őrzi a különféle alkalmazáskönyvtárakban.

A keresési sorrend (probing) megadása

A .NET-futtatórendszer a keresési sorrend segítségével deríti fel a privát szerelevény helyét. A folyamat nem olyan bonyolult, mint ahogy elsőre tűnhet. A szerelevény betöltésének kérese lehet *implicit* vagy *explicit* kéres. Implicit betöltési kéres akkor fordul elő, amikor a CLR a manifesztumot alkalmazzza arra, hogy a szerelevények helyét tartalmazó információkat `assembly extern` tokenek segítségével definiálja.

```
// Implicit betöltési kéres.
assembly extern CarLibrary
{ ... }
```

Explicit betöltési kérest a `system.Reflection.Assembly` osztálytípus `Load()` vagy `LoadFrom()` metódusával, programmozottan kezdeményezhetünk, rendszertípus szerint a típustagok késői kötése és dinamikus hívása céljából (lásd részletesen a 16. fejezetben). A következő kód az explicit betöltési kérest példázza:

```
// Explicit betöltési kéres a barátságos név alapján.
Assembly asm = Assembly.Load("CarLibrary");
```

A CLR mindkét esetben megszerezzi a szerelevény barátságos nevét, és az ügyfélalkalmazás könyvtárában megkezdzi a `CarLibrary.dll` fájl keresését. Ha nem találja a fájlt, a meggyezző barátságos névvel (`CarLibrary.exe`) a rendszer megpróbálja megkeresni a végrehajtható szerelevényt. Ha egyik fájl sem található meg az alkalmazás könyvtárában, a futtatórendszer feladja a próbálkozásokat, és futásidőben a `FileNotFoundException` kivételt küldi vissza.

Megjegyzés Ha a rendszer a kért szerelevény másolatát nem találja az ügyfélalkalmazás könyvtárában, a CLR megpróbálja a szerelevény barátságos nevével meggyezző ügyfélalkönyvtárat megtalálni (például a `C:\MyClient\CarLibrary` könyvtár). Ha ebben a könyvtárban megtalálható, a CLR betölti a memóriába a keresett szerelevényt.

Privát szerelevények beállítása

A .NET-alkalmazásokat egy szerverten telepíthetjük, csak a szükséges szerelevényeket kell a felhasználó számítógépén egyetlen mappába másolni, a kapcsolódó tartalom csoportosításához pedig valószerűleg inkább több alkönyvtárat definiálnunk.

Tételezzük fel, hogy a C:\MyApp alkalmazáskönyvtár tartalmazza a cssharpcarcl\ent.exe fájlt. A könyvtárban található a MyLibraries alkönyvtár, amely magában foglalja a carl\brary.d11 fájlt.

A két könyvtár közötti kapcsolattól függetlenül a CLR nem keres a MyLibraries alkönyvtárban, ha csak nem alkalmazunk konfigurációs fájlt. A konfigurációs fájlok különféle XML-elemeket tartalmaznak, amelyek lehetővé teszik, hogy a keresési sortrendet befolyásoljuk. A konfigurációs fájlok nevének egyeznie kell az indítóalkalmazások nevével, a fájlkiterjesztésük *.config, és az ügyfélalkalmazás könyvtárba kell telepítenünk őket. Ha tehát a cssharpcarcl\ent.exe számára készítettünk konfigurációs fájlt, a fájlt neve cssharpcarcl\ent.exe.config lesz, és (ebben a példában) a C:\MyApp könyvtárban találjuk.

A folyamat bemutatásához a C meghajtón a Windows Intéző segítségével hozzuk létre a MyApp könyvtárat. Másoljuk a cssharpcarcl\ent.exe és a carl\brary.d11 fájlt az új könyvtárba, és kattintsunk kétszer a végrehajtható fájlra a program futtatásához. A programnak ezen a ponton sikeresen futnia kell (a szerelevényeket nem regisztráltuk). A következő lépés az, hogy a C:\MyApp könyvtárban létrehozzuk a MyLibraries alkönyvtárat (lásd a 15.12. ábrát), és a carl\brary.d11 fájlt ide másoljuk.

Próbáljuk meg ismét futtatni az ügyfélprogramot. Mivel a CLR az alkalmazás könyvtárban közveletlenül nem találta carl\brary nevű szerelevényt, a képernyőn a FileNotFoundException nem kezelt kivétel értesítése jelenik meg. Ha a CLR-t a MyLibraries alkönyvtár kipróbálására szeretnénk utasítani, hozzuk létre a cssharpcarcl\ent.exe.config konfigurációs fájlt, majd mentjük *ugyanabba* a könyvtárba, amely a cssharpcarcl\ent.exe alkalmazást tartalmazza, ebben az esetben a C:\MyApp könyvtárba. Nyissuk meg a fájlt, és pontosan az alábbi tartalmat írjuk be (az XML kis- és nagybetűre érzékeny):

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="MyLibraries"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

A .NET konfigurációs fájlok mindig a <configuration> gyökércélelemmel kezdődnek. A beágyazott <runtime> elem meghatározhat egy <assemblyBinding> elemet, amelybe a <probing> elemet ágyazhatjuk. A privatePath attribútum a példakulcsa, ugyanis ennek az attribútumnak a segítségével határozzuk meg az alkalmazáskönyvtárat illetően azokat az alkönyvtárakat, amelyekben a CLR a keresést elvégzi.

A cssharpcarclient.exe.config létrehozását követően a Windows Intézőben kétszer a végrehajtható fájlra kattintva futtassuk az ügyfélprogramot. Láthatjuk, hogy a cssharpcarclient.exe problémamentesen működik (ha nem ez a helyzet, ellenőrizzük a *.config fájlt, és keressünk esetlegesen gépeleési hibákat).

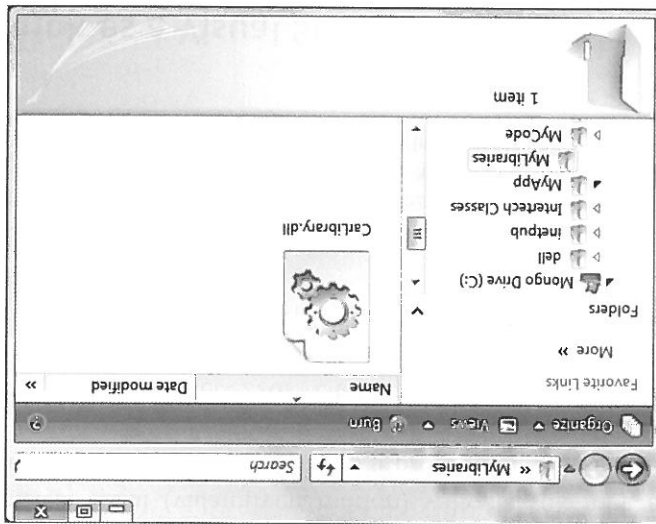
```
<probing privatePath="MyLibraries\MyLibraries\Tests" />
```

Pontosvesszővel elválasztott lista segítségével több alkönyvtárat rendelhetünk a privatePath attribútumhoz. Bár erre pillanatnyilag nincs szükségünk, de az alábbiakban látható, hogyan állítható be, hogy a CLR próbálkozon a MyLibraries és a MyLibraries\Tests alkönyvtárakban:

Megjegyzés A privatePath attribútum segítségével nem határozhatunk meg abszolút (C:\SomeFolder\SomeSubFolder) vagy relatív (..\SomeFolder\AnotherFolder) elérési útvonalat. Ha az ügyfélalkalmazás könyvtárán kívüli könyvtárat szeretnénk meghatározni, ehhez egészen más XML-elem, a <codeBase> szükséges (az elemmel kapcsolatos további részleteket a későbbiekben vizsgáljuk meg).

A <probing> elem nem határozza meg, hogy melyik szerelvény található az adott alkönyvtár alatt. Más szóval, nem mondhatjuk, hogy a carlibrary a MyLibraries alkönyvtárban, a mathut1s pedig a Bin alkönyvtárban található. A <probing> elem utasítja a CLR-t, hogy a kért szerelvényt az összes megadott alkönyvtárban az első találatig keresse.

15.12. ábra: A CarLibrary.dll most a MyLibraries alkönyvtárban található

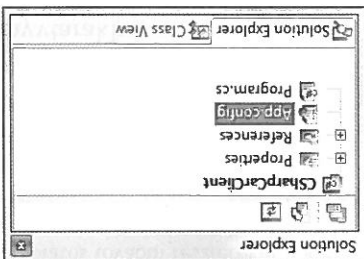


Ezt követően, tesztelési céllal (valamilyen módon) változtassuk meg a konfigurációs fájl nevét, és ismét próbáljuk futtatni a programot. Az ügyfélalkalmazás futtatása nem sikerül. A konfigurációs fájlok nevét ugyanis a hivalkozott ügyfélalkalmazás nevéből képezzük. Utolsó tesztként nyissuk meg a konfigurációs fájlt, és az összes XML-elemet újuk nagybetűkkel. A fájl mentését követően az ügyfélalkalmazás futtatása megint sikertelen lesz (ugyanis az XML-érzékeny a kis- és a nagybetűre).

Megjegyzés A CLR a keresési folyamata során az elsőként talált szereplvényt tölti be. Ha például a C:\MyApp könyvtár tartalmazza a CarLibary.dll másolatát, a rendszer ezt tölti be a memóriába, és a MyLibraries alkönyvtárban található másolatot figyelmen kívül hagyja.

Konfigurációs fájlok és a Visual Studio 2008

A Visual Studio 2008 lehetővé teszi, hogy az ügyfélalkalmazás fejlesztése közben hozzunk létre konfigurációs fájlt, de később is készíthetünk manuálisan tetszőleges szövegszerkesztővel XML konfigurációs fájlokat. Ennek bemutatóására töltjük a CSharpCarClient megoldást a Visual Studio 2008 programba, majd a Project > Add New Item menüpont segítségével illesztünk új Application Configuration File elemet a projektbe. Mielőtt az OK gombra kattintanánk, jegyezzük meg, hogy a fájl neve App.config (a fájlt ne nevezzük át). A Solution Explorer ablakban láthatjuk, hogy a rendszer az aktuális projektbe illesztette az App.config fájlt (lásd a 15.13. ábrát).



15.13. ábra: A Visual Studio 2008 App.config fájlja

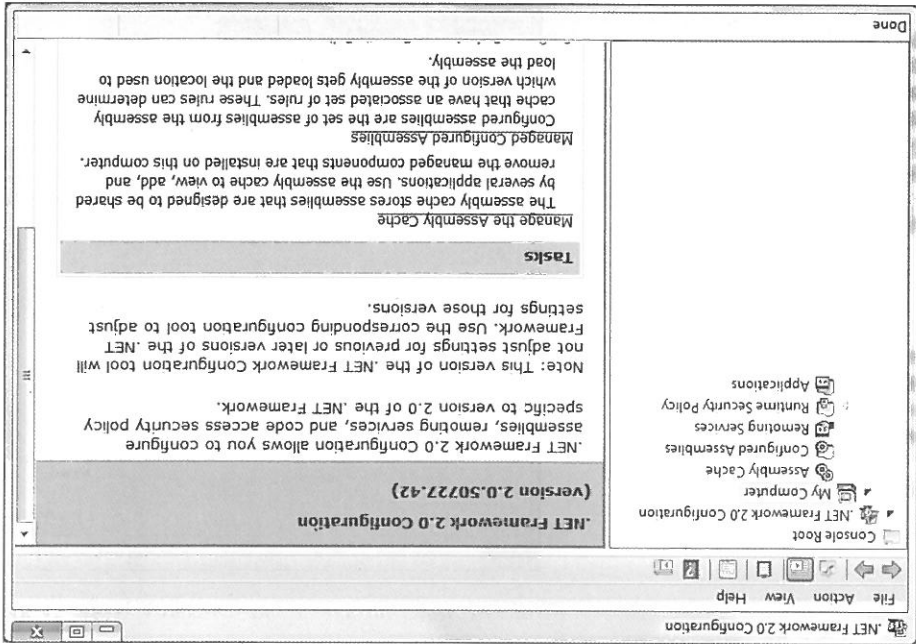
Ekkor betrhajtuk a fájlba az ügyfélalkalmazás számára szükséges XML-elemeket. A projekt fordításakor a megfelelő névadási konvenciók alapján (például CSharpCarClient.exe.config névvel) a Visual Studio 2008 automatikusan a \bin\Debug könyvtárba másolja az App.config adatait. Mindez azonban kizárólag akkor történik így, ha a konfigurációs fájl neve valóban App.config.

Ennek a megközelítésnek a segítségével csak az app.config fájlt kell karbantartanunk, és a Visual Studio 2008 gondoskodik arról, hogy az alkalmazászkönyvtár a lehető legtöbb friss konfigurációs adatot foglalja magában (még akkor is, ha történetesen átnevezzük a projektünket).

Megjegyzés Az app.config fájlok használatát a Visual Studio 2008 alkalmazásban mindig javasoljuk. Ha a Windows Intéző segítségével manuálisan adunk hozzá *.config fájlt a bin\Debug könyvtárhoz, a következő fordítás során a Visual Studio 2008 törölheti a fájlt.

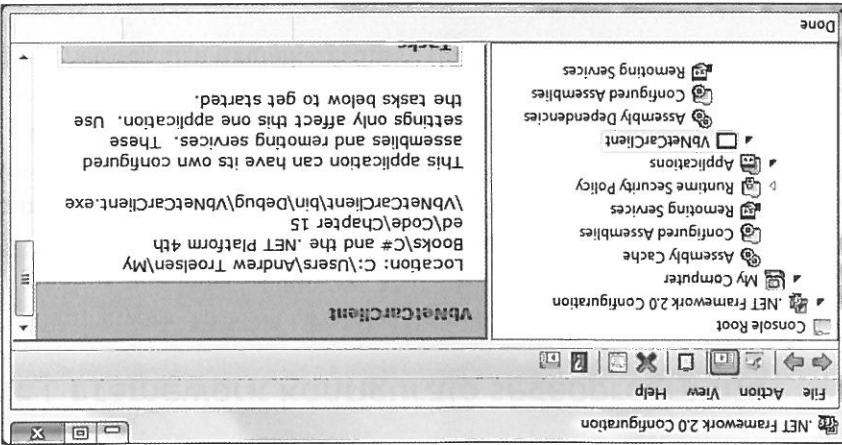
A .NET Framework konfiguráló segédprogramja

A .NET Framework 3.5 SDK tartalmaz egy olyan eszközt, amely felhasználóbarát grafikus felülettel segíti az XML konfigurációs fájlok létrehozását. A .NET keretrendszer konfiguráló (.NET Framework Configuration) segédprogramját a Control Panel Administrative Tools mappájában találjuk (ha Windows Vista operációs rendszert használunk, a baloldali panelben kattintsunk a Classic View hivatkozásra, és gyorsan megkereshetjük az Administrative Tools mappát). Ha elindítjuk a programot, láthatjuk, hogy rengeteg konfigurációs lehetőség áll a rendelkezésünkre (lásd a 15.14. ábrát).

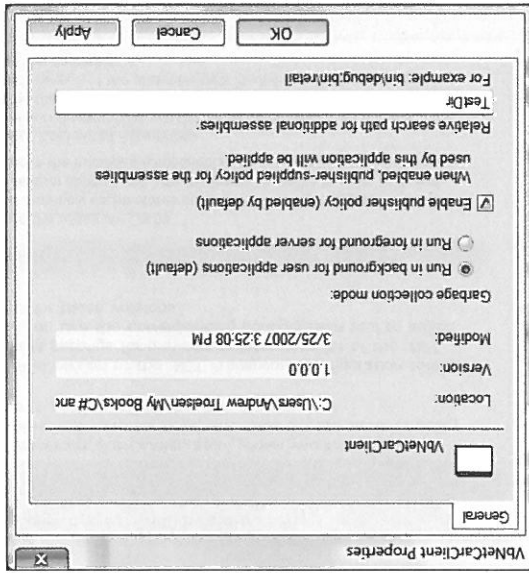


15.14. ábra: A .NET Framework konfiguráló segédprogramja

Ha a segédprogram segítségével szeretnénk elkészíteni a kliens *.config fájlt, első lépésben a programhoz kell adni a konfigurálandó alkalmazást. Ehhez kattintsunk jobb gombbal az Applications csomópontra, és válasszuk az Add menüpontot. A megjelenő párbeszédpanelben kattintsunk az Other gombra, és keressük meg a konfigurálni kívánt ügyfélprogram helyét. A mostani példánkban válasszuk a fejezet korábbi részében létrehozott VbNetCarClient.exe alkalmazást (a projekt \bin\Debug könyvtárban keressük). Ha elkészültünk, a programban a 15.15. ábrán látható új alcsomópont jelenik meg.



15.15. ábra: A VbNetCarClient.exe beállításának előkészítése



15.16. ábra: Privát keresési útvonal grafikus beállítása

Há jobb gombbal a vbnetcarclient csomóponttra kattintunk, és aktiváljuk a Properties oldalt, láthatjuk, hogy a párbeszédpanel alján szövegmező található, ahová azokat az értékeket írhatjuk, amelyeket a privatepath attribútumhoz szeretnénk rendelni. Puszta tesztelési céllal írjuk be a TestDir alkönyvtárnevét a mezőbe (lásd 15.16. ábrát fenn).

Há az OK gombra kattintunk, megnézhetjük a VbNetCarClient\bin\Debug könyvtárát, és láthatjuk, hogy a konfigurációs fájlban megjelent a megfelelő <probability> elem.

Megjegyzés A .NET Framework konfiguráló segédprogramja által készített XML-tartalmat további szerkesztésre bismasolhatjuk egy Visual Studio App.config fájlba. Ezzel a megközelítéssel kevesebbet kell gépelnünk, ha az eszközre bizzuk a kezdeti tartalom előállítását.

Megosztott szerelvények

A következőkben vizsgáljuk meg a *megosztott szerelvények* szerepét. A privát szerelvényekhez hasonlóan a megosztott szerelvények típusok és (opcionális) erőforrások gyűjteményei. A megosztott és a privát szerelvény közötti legnyilvánvalóbb különbség az, hogy a megosztott szerelvény egyetlen másolatát egyetlen gépen több alkalmazás is használhatja.

Gondoljunk csak azokra az itt létrehozott alkalmazásokra, amelyeknél be kellett állítanunk a system.windows.forms.dll szerelvény referenciáját. Ha bepillantánk ezekben az ügyfélal alkalmazásoknak az alkalmazáskönyvtárba, nem találunk a szerelvény privát másolatát. Ez annak köszönhető, hogy a system.windows.forms.dll fájlt megosztott szerelvényként telepítettük. Ha a teljes számítógépen hozzáférhető osztálykönyvtárat szeretnénk létrehozni, válasszunk ezt a módszert.

A megosztott szerelvényt nem abba a könyvtárba telepítjük tehát, mint amelyikben a szerelvényt felhasználó alkalmazás található, hanem a globális szerelvénytárba (GAC). A globális szerelvénytárat a 15.17. ábrán látható módon, az Assembly nevű Windows könyvtár (pl. C:\Windows\Assembly) egyik alkönyvtára foglalja magában.

Megjegyzés Nem telepíthetünk végrehajtható szerelvényeket (*.exe) a globális szerelvénytárba (GAC). Kizárólag a *.dll fájlkiterjesztéssel rendelkező szerelvényeket lehet megosztott szerelvényként telepíteni.

- A szerelvény barátságos neve (a barátságos név a fájlkiterjesztés nélküli szerelvénynév).
- A szerelvény verziószáma (amelyet az [AssemblyVersion] attribútum segítségével rendelünk a szerelvényhez).
- Nyilvánoskulcs-érték (amelyet az [AssemblyKeyFile] attribútum segítségével határozzunk meg).

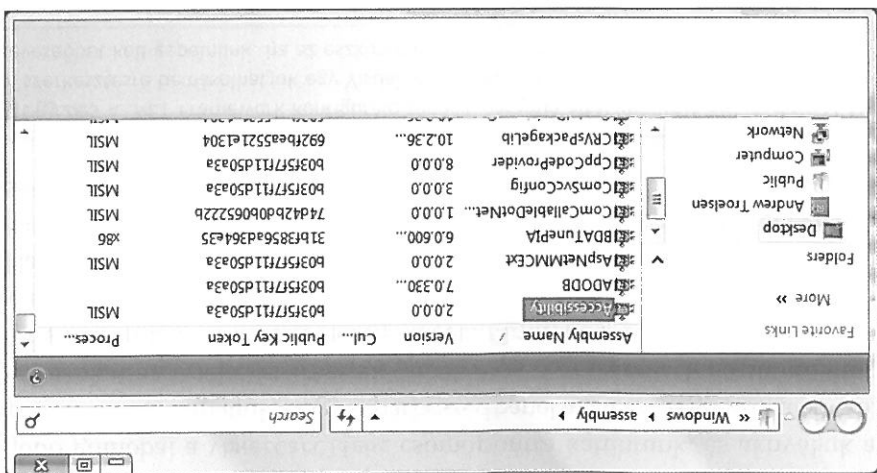
Mielőtt a szerelvényt a globális szerelvénytárba telepíthetnénk, erős nevet kell hozzárendelnünk, amely egyértelműen azonosítja a .NET bináris fájlt kibocsátótól. A „kibocsátó” lehet egyéni programozó, a vállalat egyik részlege vagy akár az egész vállalat.

Bizonyos szempontból az erős név a COM GUID (Globally Unique Identifier – globálisan egyedi azonosító) azonosítási sémájának modern .NET-ekivelense. Az alkalmazásazonosítók olyan GUID-ok, amelyek meghatározott COM-alkalmazást azonosítanak. A COM GUID értékektől (amelyek 128 bites számok) eltérően az erős nevek (részben) két kriptográfiaiilag kapcsolódó kulcsra (a nyilvános kulcsra és a titkos kulcsra) épülnek, amelyek egyediek, és jobban ellenállnak a hamisításnak, mint az egyszerű GUID.

Az erős név formálisan kapcsolódó adatok gyűjteményéből áll, amelyek nagy részét szerelvény szintű attribútumok határozzák meg:

Erős nevek megértése

15.17. ábra: A globális szerelvénytár

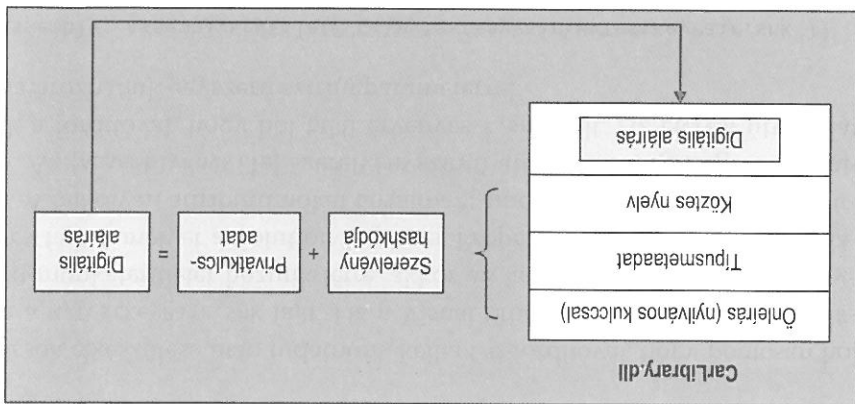


15. fejezet: A .NET-szerelvények

- Opcionális kultúraazonosító érték, amely lokalizációs célokat szolgál (az [assemblyculture] attribútummal határozzuk meg).
- Beágyazott *digitális aláírás*, amelyet a rendszer a szerevény tartalmából és a titkoskulcs-értékből hashfüggvény segítségével képez.

A szerevény erős nevének előállításához először a .NET Framework 3.5 SDK sn.exe segédprogramjának segítségével elkészítjük a nyilvános/ttkos kulcs adatokat (a részleteket lásd alább). Az sn.exe segédprogram létrehoz egy fájlt (rendszerint *.snk [Strong Name Key] fájlkiterjesztéssel), amely magába foglalja két különböző, de matematikailag kapcsolódó kulcs, a „nyilvános” és a „privát” kulcs adatait. Ha a C#-fordítóval tudatjuk az *.snk fájli helyét, akkor a fordítás során a szerevény manifestumában rögzíti a teljes nyilvános-kulcs-értéket a publickeytoken segítségével.

A C#-fordító a szerevény teljes tartalma (köztes nyelvi kód, metaadat és így tovább) alapján készít hashkódot. A *hashkód* olyan numerikus érték, amely statisztikailag egyedi a rögzített bemenet számára (lásd a 6. fejezetet). Ha tehát módosítjuk a .NET-szerevény valamely tulajdonságát (legyen az akár egyetlen karakter egy karakterlánc-literálban), a fordító teljesen új hashkódot állít elő. A hashkódot a rendszer az *.snk fájli titkoskulcs-adatával kombinálja, és elkészíti a szerevény CLR-fejlecének adataiba ágyazott digitális aláírást. A szerevény erős nevének előállítási folyamata a 15.18. ábrán látható.



15.18. ábra: Fordítási időben a rendszer elkészíti a digitális aláírást, majd a nyilvános- és titkoskulcs-adatok alapján beágyazza a szerevénybe.

A *ténylegesen privát kulcs* nem jelenik meg a manifestumban, a rendszer (a generált hashkóddal együtt) csak a tartalom digitális aláírásának elkészítésére használja. A nyilvános/ttkos kulcs adatok alkalmazásának lényege, hogy a

vállalatok, az osztályok vagy az alkalmazottak azonosítója egészen biztosan egyedi legyen a .NET univerzumban. Mindeneesetre, ha az erős név hozzárendelése elkészül, a rendszer a globális szerelvénytárba telepítheti a szerelvényt.

Megjegyzés Az erős név bizonyos védelmi szintet biztosít azokkal szemben, akik szeretnék megismisítani a szerelvény tartalmát. A legjobb gyakorlat szerint tanácsos a .NET-szerelvényeket (az *.exe szerelvényeket is) erős névvel ellátni függetlenül attól, hogy a szerelvényt a GAC-ba telepítettük-e vagy sem.

A CarLibary.dll ellátása erős névvel

Nézünk meg a fejezet korábbi részében létrehozott CarLibary szerelvény erős nevének hozzárendelési folyamatát. A folyamat első lépése a szükséges kulcsadatok generálása az sn.exe segédprogrammal. Noha az eszköz rengeteg paramettersorral paraméterrel rendelkezik, pillanatnyilag elég a -k kapcsolót ismernünk, amely utasítja az eszközt arra, hogy hozza létre a nyilvános/titkos kulcsokat magában foglaló fájlt. A C meghajtón készítjük el a MyTest-Keypair nevű könyvtárat, majd a Visual Studio parancssorának segítségével navigáljunk a könyvtárhoz. A következő parancs segítségével hozzuk létre a MyTestKeypair.snk fájlt:

```
sn -k MyTestKeypair.snk
```

A kulcsok elkészülése után tudatunk kell a C#-fordítóval, hogy pontosan hol találja a MyTestKeypair.snk fájlt. Ha a Visual Studio 2008 segítségével új C#-projekt-munkaterületet hozunk létre, akkor az tartalmazni fog egy Assembly-Info.cs fájlt (amelyet a Solution Explorer Properties csomópontja alatt találunk). A fájlt olyan attribútumokat tartalmaz, amelyek magát a szerelvényt írják le. Az [AssemblyKeyFt1e] szerelvény szintű attribútum segítségével tudathatjuk a fordítóval, hogy hol talál érvenyes *.snk fájlt. Az elérési útvonalat meghatározhatjuk egyszerű sztringparaméterrel:

```
[assembly: AssemblyKeyFt1e(@"C:\MyTestKeypair\MyTestKeypair.snk")]
```

Megjegyzés Ha manuálisan határozzuk meg az [AssemblyKeyFt1e] attribútumot, a Visual Studio 2008 figyelmeztetést küld, amelyben arra hívja fel a figyelmeűnket, hogy használjuk a csc.exe /keyFt1e opciót, vagy pedig generáltsuk a kulcspárt a Visual Studio 2008 Pro-pertes ablaka segítségével. Az IDE alkalmazásával hamarosan pontosan ezt fogjuk tenni (tehát nyugodtan hagyjuk figyelmen kívül a figyelmeztetést).

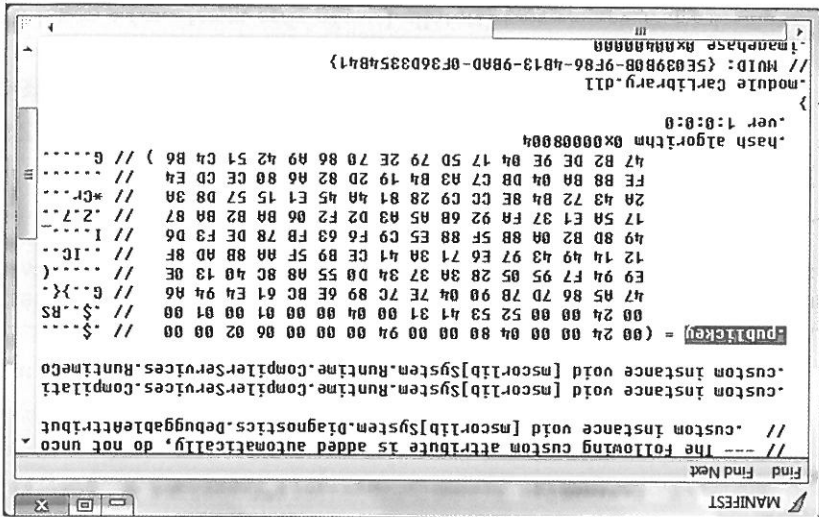
Mivel a megosztott szerelvény verziója fontos az erős név szempontjából, ezért elengedhetetlen, hogy verziószámot válasszunk a carlbrary.dll szá-mára. Az AssemblyInfo.cs fájlban találjuk az [AssemblyVersion] attribútumot. Az attribútum kezdeti értéke 1.0.0.0:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

A .NET-ben a verziószám négy részből áll (<major>, <minor>, <build>, <revision>). Magunk dönthetünk a verziószám meghatározásáról, de utasíthatjuk a Visual Studio 2008 alkalmazást, hogy automatikusan helyettesítse tokennel növelje a build és a revision számokat minden fordítás végrehajtásakor, és ne meghatáro-zott build és revision értéket alkalmazzon. Nézzük meg erre a következő példát:

```
// Formátum: <Major number>.<Minor number>.<Build number>.<Revision number>
// A verziószám mindegyik részének esetében az érvényes érték 0
// és 65535 közé esik.
[assembly: AssemblyVersion("1.0.*")]
```

Ezen a ponton a C#-fordítónak minden információ a birtokában van, amelyre az erős névadatok elkészítéséhez szükséges van (mivel nem határozzunk meg egyedi kultúraazonosító értéket az [AssemblyCulture] attribútum segítségével, az aktuális számítógép kultúráját "örököljük", amely ebben az esetben amerikai angol).

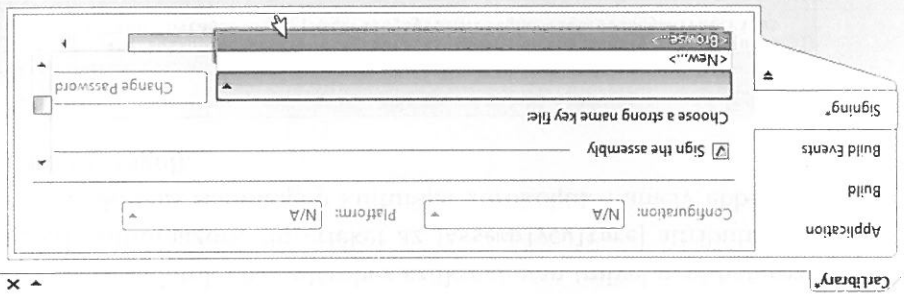


15.19. ábra: Az erős névvel ellátott szerelvény az önleírásban rögzíti a nyilvános kulcsot

Fordítsuk le a carllibrary kódkönyvtárat, majd az `itlasm.exe` programmal nyissuk meg a szerelvény manifestumát. Láthatjuk, hogy új `.publikey cím-ke dokumentálja az összes nyilvánoskulcs-információt`, a `.ver token` pedig az `[AssemblyVersion]` attribútummal meghatározott verziószámot rögzíti (lásd a 15.19. ábrát fenn).

Erős nevek hozzárendelése a Visual Studio 2008 segítségével

Mielőtt a GAC-ba telepítenénk a `carllibrary.dll` szerelvényt, fontos tudni, hogy a Visual Studio 2008 a projekt Properties oldalának segítségével lehetővé teszi az `*.snk` fájlnak helyének meghatározását (valamint új `*.snk` fájl generálását az `sn.exe` kézi futtatása helyett). A `carllibrary` projektben erre nincs szükség, de a példa kedvéért válasszuk a signing csomópontot, adjuk meg az `*.snk` fájl elérési útvonalát, majd jelöljük be a `Sign the assembly` jelölőnégyzetet (lásd a 15.20. ábrát).



15.20. ábra: Az `*.snk` fájl meghatározása a Properties oldalon

Megjegyzés A Properties szerkesztő Application fülén találjuk az Assembly Information gombot. Ha rákattintunk, a párbeszédpanelben beállíthatunk több szerelvény szintű attribútumot, például a verziószámot, a szerzői joggal kapcsolatos információkat stb.

Megosztott szerelvények telepítése/eltávolítása a globális szerelvénytárba

Az utolsó lépés a (mostanra erős névvel ellátott) `carllibrary.dll` telepítése a globális szerelvénytárba. Ennek legegyszerűbb módja, ha a Windows Intéző segítségével a `C:\Windows\Assembly` könyvtárba húzzuk a szerelvényt

(egy gyors tesztelés megér). A szerverény másolása és beillesztése a GAC-ablakba nem működik. Szó szerint át kell húznunk a *.dll fájlt a globális szerverénytárlakába.

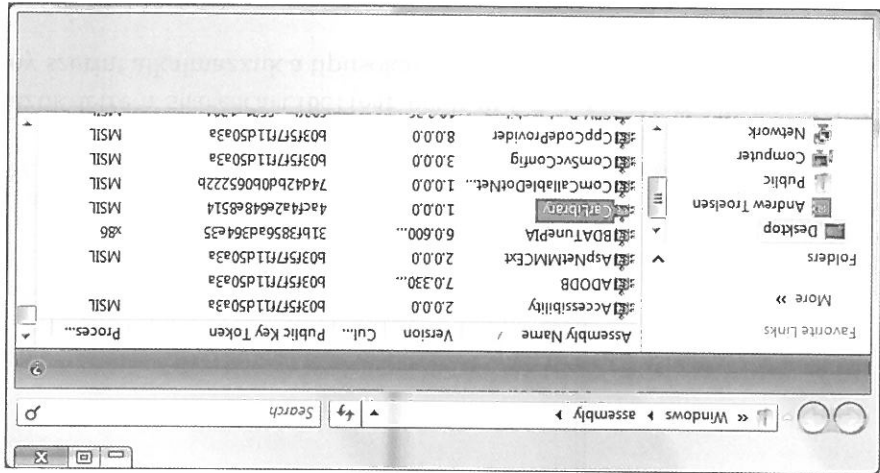
A szerverény áthúzása tesztelésre alkalmas, ezzel szemben a .NET Framework 3.5 SDK a gacutil.exe parancssori segédprogrammal a globális szerverénytárlak tartalmának felügyeletét biztosítja. A 15.1. táblázat a gacutil.exe néhány idevágó opcióját mutatja (a /? kapcsoló segítségével az összes opciót megtekinthetjük).

Opció	Szerepe
-------	---------

/?	Erős névvel rendelkező szerverényt telepít a globális szerverénytárlakba.
/u	El távolítja a szerverényt a globális szerverénytárlakból.
/?	Megjeleníti a globális szerverénytárlakban található szerverényeket (vagy a meghatározott szerverényt).

15.1. táblázat A gacutil.exe különböző lehetőségei

Valamelyik módszerrel telepítjük a carlibrary.dll szerverényt a globális szerverénytárlakba (lásd a 15.21. ábrát).



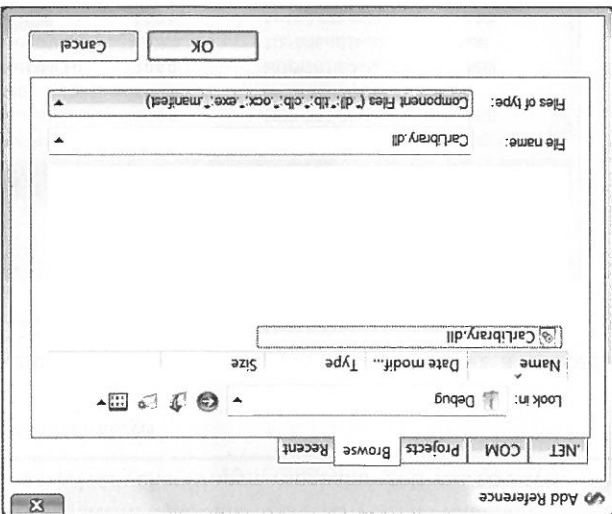
15.21. ábra: Erős névvel rendelkező, megosztott CarLibrary (version 1.0.0.0)

Megjegyzés Jobb gombbal rákattintunk a szerverény ikonjára, hogy megtekinthessük a Properties oldalt, valamint el is távolíthatjuk a helyi menüből (a gacutil.exe /u kapcsolójának GUI-megfelelője) a szerverény meghatározott verzióját.

A megosztott szerelvények felhasználása

Amikor olyan alkalmazást készítettünk, amely megosztott szerelvényekkel dolgozik, az egyetlen különbség a privát szerelvények használatához képest az, hogy hogyan hivatkozunk a könyvtárra a Visual Studio 2008 segítségével. Az eszköz tekintetében nincs különbség (továbbra is az Add Reference párbeszédpanelt alkalmazzuk). A párbeszédpanel azonban *nem* teszi lehetővé, hogy úgy hivatkozunk a szerelvényre, hogy a C:\Windows\Assembly könyvtárhoz navigálunk. Ez tehát azt jelenti, hogy ahogy sem hivatkozhatunk a kiemelt szerelvényre.

Ha olyan szerelvényre kell hivatkoznunk, amelyet a globális szerelvénytárba telepítettünk, a Browse fül segítségével az *eredeti* projekt \bin\Debug könyvtárát kell megkeresnünk (lásd a 15.22. ábrát).



15.22. ábra: A szerelvényekre úgy hivatkozhatunk, ha a projekt \bin\Debug könyvtárhoz navigálunk

Hozzuk létre a SharedCarLibClient nevű új C# parancssori alkalmazást, és igény szerint alkalmazzuk a típusokat:

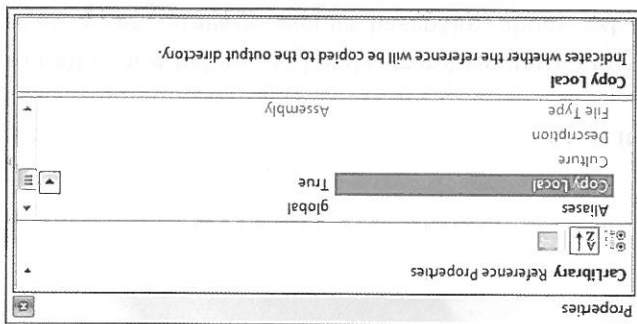
```
using CarLibrary;

namespace SharedCarLibClient
{
    class Program
    {
    }
```

```
static void Main(string[] args)
{
    Console.WriteLine("***** Shared Assembly Client *****");
    SportsCar c = new SportsCar();
    c.TurboBoost();
    Console.ReadLine();
}
```

Az ügyfélalkalmazás fordítását követően a Windows Intéző segítségével keresztük meg azt a könyvtárat, amelyben a `sharedcarlib\client.exe` található. Láthatjuk, hogy a Visual Studio 2008 *nem* másolta a `carlibrary.dll`-t az alkalmazás könyvtárába. Ha olyan szerelvényre hivatkozunk, amelynek manifestuma `pub1ckey` értéket tartalmaz, a Visual Studio 2008 felfigyel, hogy az erős nével ellátott szerelvényt a globális szerelvénytárba telepítettük, ezért nem másolja be a bináris fájlt a könyvtárba.

Ha szeretnénk, hogy a Visual Studio 2008 a megosztott szerelvényt az ügyfélalkalmazás könyvtárába másolja, kiválaszthatjuk a szerelvényt a Solution Explorer References csomópontjából, és a Properties ablakban a Copy Local tulajdonság értékét True-ra állíthatjuk (lásd a 15.23. ábrát). Így a rendszer a következő fordításkor az ügyfélkönyvtárba másolja a `*.dll` fájlt.



15.23. ábra: A Copy Local tulajdonság a rendszert az erős névvel ellátott kódkönyvtár másolására kényszerítheti

A SharedCarLibClient manifestuma

Amikor a szerevény számára erős nevet generálunk, a szerevény manifesztuma a teljes nyilvános kulcsot rögzíti. Ha az ügyféllakalmazás erős névvel ellatott szerevényre hivatkozunk, a manifesztum a teljes nyilvános kulcs tömörített hashértékét rögzíti, és a publikus kulcsok címkével jelöli. Ha az 1dasm.exe

segítségével megnyitjuk a `sharedcarlibcli.exe` manifestumát, a következőket találhatjuk a fájlban (természetesen a nyilvánoskulcs-töken különbözőzik, mert azt a rendszer a nyilvános kulcs értéke alapján számítja ki):

```
.assembly extern CarLibrary
{
  .publickeytoken = (21 9E F3 80 C9 34 8A 38)
  .ver 1:0:0:0
}
```

Ha összehasonlítjuk az ügyfélalkalmazás manifestumában rögzített nyilvánoskulcs-töken értéket a globális szerelvénytárban megjelendő nyilvánoskulcs-töken értékével, láthatjuk, hogy a két érték hajszálpontosan megegyezik. A nyilvános kulcs képviseli az erős névvel ellátott szerelvény azonosítójának egyik tényt be, amelynek nyilvánoskulcs-értéke a `219EF380C9348A38` hashértékkel rendelkezik. Ha a CLR nem talál az ilyen feltételeknek megfelelő szerelvényt a globális szerelvénytárban (és a `carlibrary` privát szerelvényt sem találja az ügyfélalkalmazás könyvtárában), a rendszer a `FileNotFoundException` kivételt dobja.

Forráskód A `sharedcarlibcli.exe` alkalmazás megtalálható a 15. fejezet alkönyvtárában.

A megosztott szerelvények konfigurálása

A privát szerelvényekhez hasonlóan a megosztott szerelvényeket is beállít-hatjuk az ügyfélalkalmazás konfigurációs fájljával. Természetesen, mivel a megosztott szerelvényeket jól ismert helyre (a globális szerelvénytárba) telepítjük, ezért nem szükséges a `<privatepath>` elemet használni, ahogy azt a privát szerelvények esetében tettük (ha azonban az ügyfélalkalmazás megosztott és privát szerelvényeket is használ, a konfigurációs fájlban megtalál-juk a `<privatepath>` elemet).

Az alkalmazás konfigurációs fájljait a megosztott szerelvényekkel együtt használhatjuk, ha szeretnénk a CLR-t a meghatározott szerelvény másik ver-ziójához kötni, és így gyakorlatilag figyelmen kívül hagyjuk az ügyfélalkal-mazás manifestumában rögzített értéket. Ez több okból is hasznos lehet.

Képzeliük el például, hogy a szerelvény 1.0.0.0 verzióját szállítottuk, és ezt követően később felfedezünk egy óriási hibát a kódban. Az egyik javítási intézkedés az lehet, ha újrapépjük az ügyfélalkalmazást úgy, hogy a hibamentesített szerelvény javított verziójára (mondjuk az 1.1.0.0 verzióra) hívunk zón, majd a frissített ügyfélalkalmazást és az új könyvtárat minden egyes cél-számútgépre elküldjük.

A másik lehetőség az, hogy leszállítjuk az új kódkönyvtárat és a *.config fájlt, amely automatikusan az új (hibamentes) verzió alkalmazására utasítja a futtatórendszert. Ha az új verziót telepítjük a globális szerelvénytárba, az eredeti ügyfélprogram újrafordítás és ismételt küldés nélkül fog működni.

Nézzünk egy másik példát: leszállítottuk a hibamentes szerelvény első verzióját (1.0.0.0), majd egy-két hónap múlva bővíttük a kérdéses szerelvény funkcionálitását, és előállítottuk a 2.0.0.0 verziót. Nyilvánvaló, hogy az 1.0.0.0 verzióval fordított, már létező ügyfélalkalmazások számára az új típusok ismeretlenek, hiszen forráskódjuk nem hívkozik az új összetevőkre.

Az új ügyfélalkalmazások azonban szeretnénk a 2.0.0.0 verzióban található új funkciókra hivatkozni. A .NET platformon nyugodtan elküldhettük a célszámítógépekre a 2.0.0.0 verziót, és ezt a régebbi 1.0.0.0 verzió mellett használhatjuk. Ha szükséges, az alkalmazás konfigurációs fájlja segítségével a létező ügyfélalkalmazásokat dinamikusan átállíthatjuk a 2.0.0.0 verzió be-töltésére (így hozzáférhetünk a kifinomultabb funkciókhoz) anélkül, hogy új-ra kellene fordítani és telepíteni az ügyfélalkalmazást.

Az aktuális megosztott szerelvény befagyasztása

Hogy dinamikusan összekapcsolhassuk egy megosztott szerelvény megháta-rozott verzióját, nyissuk meg a Windows Intézőt, és a carl.library projékt je-lenlegi lefordított verzióját (1.0.0.0) másoljuk be egy különálló alkönyvtárba (az alábbi példában a "CarlLibrary Version 1.0.0.0" könyvtárba), így ezzel a verzió befagyasztását jelképezzük (lásd a 15.24. ábrát).

A megosztott szerelevény 2.0.0.0 verziójának elkészítése

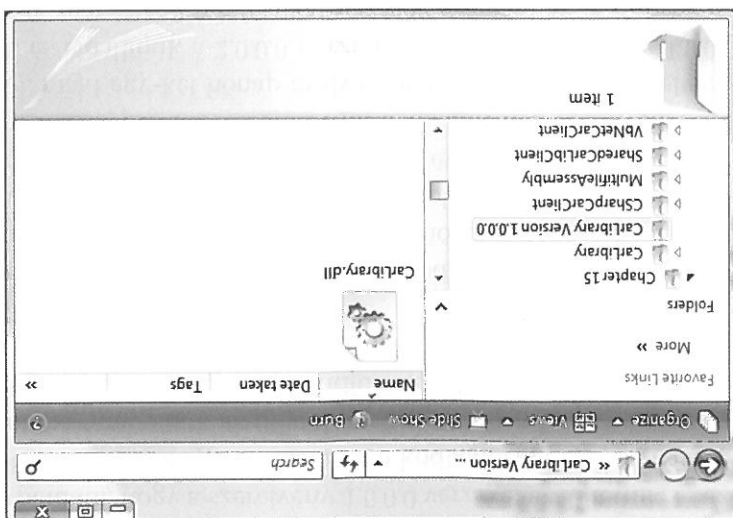
Nyissuk meg a CarLibrary projektet, és bővítsük a forráskódot a MusicMedia nevű új felsorolással, amely négy lehetséges zenei eszközt definiál:

```
// A zeneforrást tartalmazza.
public enum MusicMedia
{
    musicCd,
    musicTape,
    musicRadio,
    musicMp3
}
```

Adjunk a Car típushoz egy új nyilvános metódust, amely lehetővé teszi, hogy a hívó bekapcsolja a megadott médialejátszók egyikét (a Car.cs fájlban min-denképpen importáljuk a System.Windows.Forms névtérét):

```
public abstract class Car
{
    ...
    public void TurnOnRadio(bool musicOn, MusicMedia mm)
    {
        if (musicOn)
            MessageBox.Show(string.Format("Jamming {0}", mm));
    }
}
```

15.24. ábra: A CarLibrary.dll aktuális változatának befügyasztása



```
else
{
    MessageBox.Show("quiet time...");
}
}
```

Módosítsuk a car osztály konstruktorait, hogy megjelenítsünk egy MessageBox elemet, amely igazolja, hogy valóban a CarLibrary 2.0.0 verziót használjuk:

```
public abstract class Car
{
    ...
    public Car()
    {
        MessageBox.Show("CarLibrary version 2.0.1");
    }
    public Car(string name, short max, short curr)
    {
        MessageBox.Show("CarLibrary version 2.0.1");
    }
    petName = name; maxSpeed = max; currSpeed = curr;
}
...
}
```

Végül az újrafordítás előtt mindenképpen frissítsük a szerverű verzióját 2.0.0-ra, azaz módosítsuk az [AssemblyVersion] attribútumnak átadott értékét:

```
// CarLibrary 2.0.0 verzió (most már zenével!)
[assembly: AssemblyVersion("2.0.0.0")]
```

Há megnézzük a projekt \bin\Debug könyvtárát, láthatjuk a szerverű új verzióját (2.0.0), míg az 1.0.0 verziót a CarLibrary Version 1.0.0 könyvtárban találja a rendszer. A korábbiaknak megfelelően telepítsük az új szerverűt a globális szerverűtárba. Most ugyanannak a szerverűnek két különböző verziója áll a rendelkezésünkre (lásd a 15.25. ábrát).
Ha a Windows Intézőben kettő az ikonjára kattintunk, és futtatjuk az aktuális sharedCarLibClient.exe programot, a "CarLibrary Version 2.0!" üzenet dobott nem látni, ugyanis a manifestum kimondottan az 1.0.0 verziót kéri. Hogyan oldható meg az, hogy a CRL a 2.0.0 verzióhoz ragaszkodjon?

Megjegyzés A Visual Studio 2008 az alkalmazások fordításakor automatikusan alaphelyeztetbe állítja a referenciákat. Ezért, ha a Visual Studio 2008-cal futtatunk a sharedCarLibClient.exe programot, az a CarLibrary.dl1 2.0.0 verzióját használja. Ha véletlenül így futtatjuk az alkalmazást, törljük az aktuális CarLibrary.dl1 referenciát, és válasszuk az 1.0.0 verziót (amelyet a korábbi javaslat alapján a CarLibrary Version 1.0.0 könyvtárban helyeztünk el).

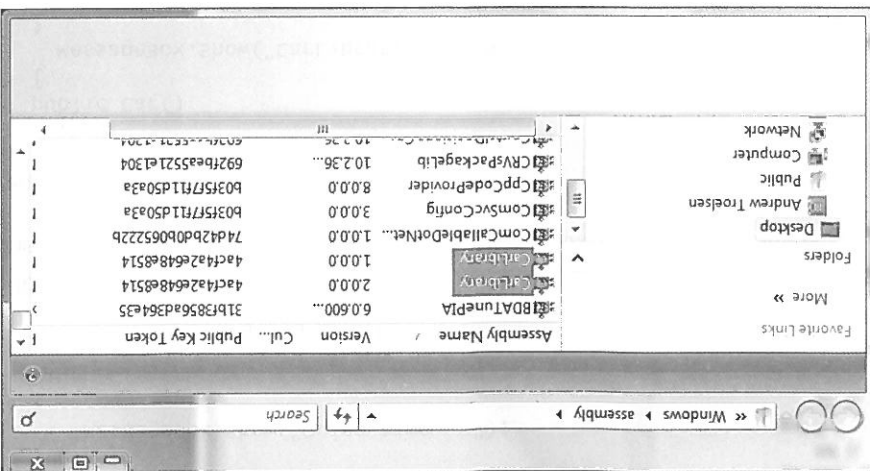
```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
```

vénytár segítségével deríthetjük ki.

az ügyfélalkalmazás manifestumában az `ildasm.exe` vagy a globális szerel-
kules tokenértéke elter majd a következő példában látható címkektől. Az értéket
amely a következő XML-adatokat foglalja magában. Természetesen a nyilvános
ban hozzuk létre a `sharedcarlibclient.exe.config` nevű új konfigurációs fájlt,
zót (a `newversion` attribútummal). A `sharedcarlibclient` alkalmazás könyvtár-
ribútummal), illetve a globális szerelvénytárban található, betölteni kívánt ver-
határozza a *pillanathyilag* a manifestumában található verziót (az `oldversion` att-
ribútummal). A `<dependentAssembly>` elem definiálja a `<bindingRedirect>` elemet, amely meg-
lve teljesen elhagyhatjuk, ha a számítógép alapértelmezett kultúrája megfelel).
laminant egy opcionális kultúraazonosító attribútumot (értéke üres sztring lehet, il-
listázott szerelvényének barátságos nevét (ebben a példában a `carlibarayt`), va-
identit> elemet, amely meghatározza az ügyfélalkalmazás manifestumában
alkalmazzuk a `<dependentAssembly>` elemet. Létre kell hoznunk az `<assembly-
megosztott szerelvény másik verzióját töltse be, készítsünk konfigurációs fájlt, és
Ha azt szeretnénk, hogy a CLR a manifestumában listázott verzió helyett a`

Dinamikus átírányítás a megosztott szerelvények meghatározott verzióba

15.25. ábra: A megosztott szerelvény különböző verziói egymás mellett



```

<assemblyIdentity name="CarLibrary"
  publicKeyToken="219ef380c9348a38"
  culture="neutral"/>
  <bindingRedirect oldVersion="1.0.0.0"
    newVersion="2.0.0.0"/>
</dependentAssembly>
</assemblyBinding>
</runtime>
</configuration>

```

A Windows Intézőben kattintsunk kétszer a végrehajtható fájlra, és futtassuk a `sharedcarlibclient.exe` programot. A képernyőn üzenet fog tájékoztatni arról, hogy a rendszer betöltötte a 2.0.0.0 verziót.

Az ügyfélszolgálati konfigurációs fájljában több `<dependentAssembly>` elem is megjelenhet. Bár a példában erre nincs igazán szükség, de tetelezzük fel, hogy a `sharedcarlibclient.exe` manifesztuma magában foglalja a `math-library` szerverlány 2.5.0.0 verziójának referenciáját. Ha a `CarLibrary` 2.0.0.0 verzióján kívül) szeretnénk a `MathLibrary` 3.0.0.0 verziójára áttérni, a `shared-carlibclient.exe.config` fájlt a következőképpen kellene módosítanunk:

```

<configuration>
  <runtime>
    <assemblyBinding>
      <dependentAssembly>
        <assemblyIdentity name="CarLibrary"
          publicKeyToken="219ef380c9348a38"
          culture="neutral"/>
        <bindingRedirect oldVersion="1.0.0.0"
          newVersion="2.0.0.0"/>
      </dependentAssembly>
      <!-- Controls Binding to CarLibrary -->
      <dependentAssembly>
        <assemblyIdentity name="MathLibrary"
          publicKeyToken="219ef380c9348a38"
          culture="neutral"/>
        <bindingRedirect oldVersion="2.5.0.0"
          newVersion="3.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>

```

Megjegyzés A régi verziószámok tartományát az `oldVersion` attribútum segítségével határozhatjuk meg; a `<bindingRedirect oldVersion = „1.0.0.0-1.2.0.0” newVersion = „2.0.0.0”/>` például tájékoztatja a CLR-t, hogy a 2.0.0.0 verziót alkalmazza a 1.0.0.0-1.2.0.0 régi értéktartományban.

A .NET Framework konfiguráló programja – ismét

Reményeinknek megfelelően a megosztott szerelvények konfigurációs adatait a .NET Framework grafikus konfiguráló segédprogramjának a segítségével is elkészíthetjük. A privát szerelvények *.config fájljainak létrehozásához hasonlóan a folyamat első lépése az, hogy a konfigurálni kívánt *.exe fájlra hivatkozzunk. Ehhez töröljük az előbb készített sharedcarlibclient.exe.config fájlt, majd kattintsunk jobb gombbal az Applications csoportra, és adjuk a projekthez a sharedcarlibclient.exe referenciáját. Ezt követően kattintsunk az összeadás jelre (+), és a kibővített csoportban válasszuk a Configured Assemblies csoportot, majd a segédprogram jobb oldalán kattintsunk a Configure an Assembly hivatkozásra.

Ekkor megjelenik egy párbeszédpanel, amely lehetővé teszi, hogy barátságos felhasználófelület-elemek segítségével létrehozzuk a <dependentassembly> elemet. Először kapcsoljuk be a Choose an assembly from the list of assemblies this application uses rádiógombot, majd kattintsunk a Choose Assembly gombra.

A megjelenő párbeszédpanelben nemcsak az ügyfélalkalmazás manifestumában listázott szerelvényeket látjuk, de azokat a szerelvényeket is, amelyekre ezek a szerelvények hivatkoznak. Válasszuk például a carlibrary szerelvényt. Ha a Finish gombra kattintunk, a képernyőn megjelenik a Properties oldal, amely az ügyfélalkalmazás manifestumának ezt az apró jellemzőjét mutatja. Itt a Binding Policy fül segítségével elkészíthetjük a <dependentassembly> elemet.

Ha a Binding Policy fület választjuk, a Requested Version szövegmezőben beállíthatjuk az oldversion attribútumot (1.0.0.0), a New Version szövegmezőben zöben pedig a newversion attribútumot (2.0.0.0). Miután véglegessítettük a be-

```
<?xml version="1.0"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="CarLibrary"
          publicKeyToken="219ef380c9348a38"
          </>
        <publisherPolicy apply="yes"
          </>
        <bindingRedirect oldVersion="1.0.0.0"
          newVersion="2.0.0.0"
          </>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

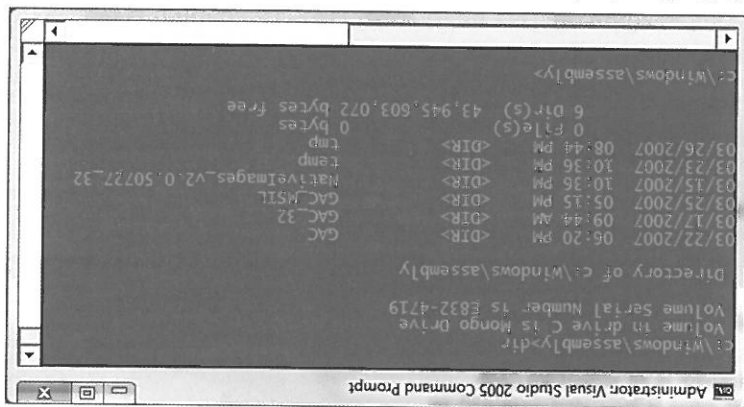
A GAC belső felépítése

A GAC belső felépítése

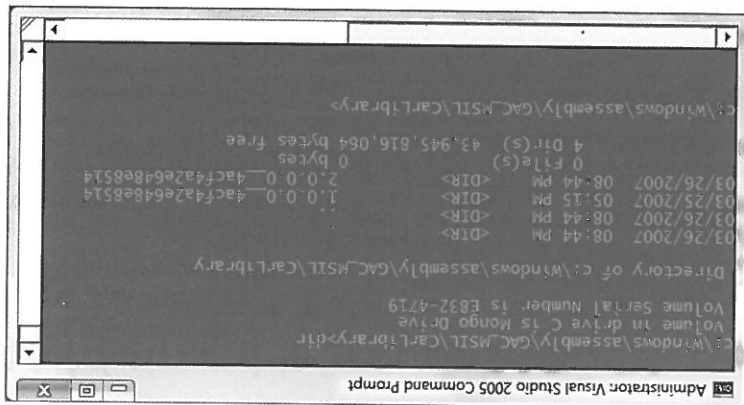
A kibocsátó házirend tanulmányozása előtt vizsgáljuk meg a globális szerverlenti belső felépítést. Ha a Windows Intéző segítségével megnézzük a globális szerverlenti tárat, a megosztott szerverlenti egyes verzióit képviselő ikonokat találunk. A grafikus shell az sfusion.d11 COM-kiszolgáló révén áll a rendelkezésünkre. Az ikonok mögött bonyolult (de átlátható) könyvtárstruktúra rejlik. Indítsuk el a parancssort, és navigáljunk az Assembly könyvtárhoz:

```
cd c:\windows\assembly
```

Irtuk be a dir parancsot a parancssorba. A GAC_MISL nevű könyvtárat fogjuk itt találni (lásd a 15.26. ábrát).



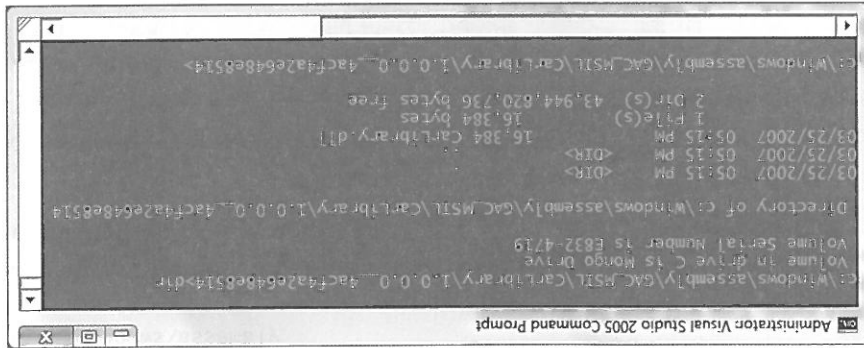
15.26. ábra: A rejthet GAC_MISL alkönyvtár



15.27. ábra: A rejthet CarLibrar alkönyvtár belüli

Navigáljunk a GAC_MSIL könyvtárhoz, és ismét adjuk ki a `dir` parancsot. A képernyőn megjelenő azokrak az alkönyvtáraknak a listáját, amelyeknek a neve megegyezik az `sfusion.d11` ikonjainak nevével. Menjünk a `CarlLibrary` alkönyvtárhoz, és megint adjuk ki a `dir` parancsot (lásd a 15.27. ábrát fenn).

A globális szerelvénytár a megosztott szerelvény minden verziója számátalkönyvtárat készít. Az alkönyvtárak névadási konvenciója a következő: `<assembly-verzió>_neutral_kulcs_token`. Ha a `CarlLibrary` szerelvény 1.0.0.0 verziójának könyvtárhoz lépünk, a könyvtárban a `kodkonyvtar_má-`solatát találjuk (lásd a 15.28. ábrát).



15.28. ábra: A `CarlLibrary.dll` belső másolata a globális szerelvénytárban

Ha erős nével ellátott szerelvényt telepítettünk a globális szerelvénytárba, az operációs rendszer az Assembly alkönyvtár alatt található könyvtárstruktúrát bővíti. Így a CLR a meghatározott szerelvény több verziójával dolgozhat, és elkerülheti az azonos nével rendelkező *.d11 fájlok miatt várható ütközéseket.

Kibocsátó házirenddel rendelkező szerelvény

A következő konfigurációs témakör a *kibocsátó házirenddel rendelkező szerelvények* szerepe. A *.config fájlokat a megosztott szerelvény meghatározott verzióához köthetjük, és megkerülhetjük az ügyfélalkalmazás manifestumában rögzített verziót. Képzeliük azonban el, hogy rendszergazdaként egy meghatározott számítógép *összes* ügyfélalkalmazásában ismét be kell állítanunk a `CarlLibrary.d11` szerelvény 2.0.0.0 verzióját.

A konfigurációs fájl névadási konvencióinak köszönhetően ugyanazt az XML-tartalmat több helyre kellene másolnunk (feltéve, ha tisztában vagyunk azzal, hogy hol találjuk a carlbrary szerelvényt használó futtatható fájlkat). Ez karbantartási rémálom lenne.

A kibocsátó házirend lehetővé teszi a meghatározott szerelvény kibocsátója (saját magunk, osztályunk vagy vállalatunk) számára, hogy a *.config fájl bináris verzióját szállítsa, ezt pedig a kapcsolódó szerelvény legújabb verziójával együtt a globális szerelvénytárba lehet telepíteni. Ennek a megközelítésnek az előnye az, hogy az ügyfélalkalmazás-könyvtáraknak nem kell meghatározott *.config fájlokat tárolnia. Ehelyett a CLR beolvassa az aktuális manifesztumot, és megpróbálja megkeresni a kért verziót a globális szerelvénytárban. Ha azonban a CLR kibocsátó házirenddel rendelkező szerelvényt talál, megvizsgálja a beágyazott XML-adatokat, és a globális szerelvénytár szintjén végrehajtja a kért átirányítást.

A kibocsátó házirenddel rendelkező szerelvényeket az a1.exe (assembly linker) .NET-segédprogrammal készíthetjük el. A kibocsátó házirenddel rendelkező szerelvények létrehozásához csupán a következő bemeneti paraméterek szükségesek:

- az átirányítási információkat magában foglaló *.config vagy *.xml fájl elérési útvonala,
- az eredményül kapott kibocsátó házirenddel rendelkező szerelvény neve,
- a kibocsátó házirenddel rendelkező szerelvény digitális aláírását tartalmazó *.snk fájl elérési útvonala,
- a készülő kibocsátó házirenddel rendelkező szerelvényhez rendelt verziószámok.

Ha a carlbrary.d11 használatát szabályozó kibocsátó házirenddel rendelkező szerelvényt hozunk létre, a következő parancsot kell használnunk (a parancssori ablakban egyetlen sorba kell beírnunk):

```
a1 /link: CarlbraryPolicy.xml /out:policy.1.0.Carlbrary.d11 /keyf:C:\Mykey\mykey.snk /v:1.0.0.0
```


A <code>Base> elem

A <code>Base> elem

Az alkalmazás-konfigurációs fájlok *kódalapokat* határozhatnak meg. A <code>Base> elemmel utasíthatjuk a CLR-t, hogy az alárendelt szerelvényeket teljes egészben helyen keresse (például hálózati végpontokon vagy az ügyfélalkalmazás könyvtárán kívül található lokális könyvtárban).
Ha a <code>Base> elemhez rendelt érték távoli számítógépre mutat, a rendszer a szerelvényt igény szerint letölti a globális szerelvénytár meghatározott könyvtárba, a *letöltési gyorsítótárba*. A <code>Base> elemről betöltött szerelvényeket erős névvel kell ellátnunk (hiszen a CLR másként hogyan tudna távoli szerelvényeket telepíteni a globális szerelvénytárba). Számítógépünk letöltési gyorsítótárának tartalmát a gacutil.exe segédprogram /ldi kapcsolójának segítségével tudjuk megnézni:

gacutil /ldi

Megjegyzés Technikailag a <code>Base> elem segítségével kereshetünk erős névvel nem rendelkező szerelvényeket. A szerelvény elérési útjait relatív módon kell megadni (ezért alig valószínűleg több, mint a <privatepath> elem alternatívája).

Ha működés közben szeretnénk látni a <code>Base> elemet, hozzuk létre a <code>Base> parametrsorát alkalmazást, állítsuk be a CarLibary.dll szerelvényi fájlt:

```
using CarLibary;

namespace CodeBaseClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Fun with CodeBases *****");
            SportsCar c = new SportsCar();
            Console.WriteLine("Sports car has been allocated.");
            Console.ReadLine();
        }
    }
}
```

Mint ahogy a carltribrary.d11 szerelvényt telepítettük a globális szerelvénytárba, ebben a formájában futtathatjuk a programot. A <codebase> elem alkalmazásának bemutatásához hozzunk létre új könyvtárat a C meghajtón (például C:\MyAsms könyvtár), majd helyezzük ide a carltribrary.d11 2.0.0 másolatát. Adjuk az app.config fájlt a codebaseconfig projektéhez (a korábban ismertett módon), majd készítjük el a következő XML-tartalmat (a .pub1ickey-token értéke eltérő lesz, szükség szerint ellenőrizzük az értéket a globális szerelvénytárban):

```
<configuration>
  <runtime>
    <assemblyBinding>
      <dependentAssembly>
        href="file:///C:/MyAsms/CarlTbrary.d11" />
        <codebase version="2.0.0"
          pub1ickeytoken="219ef380c9348a38" />
        <assemblyIdentity name="CarlTbrary"
          pub1ickeytoken="219ef380c9348a38" />
        <dependentAssembly>
          <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
```

A <codebase> elemet az <assemblyIdentity> elembe ágyazzuk, amely a name és pub1ickeytoken attribútumok segítségével határozza meg a pub1ickeytoken értékekhez kapcsolódó barátságos nevet. A <codebase> elem határozza meg a be-tölteni kívánt szerelvény verzióját és elérési útvonalat (a href tulajdonság al-kalmazásával). Ha a globális szerelvénytárból töröljük a carltribrary.d11 2.0.0 verzióját, az ügyfélalkalmazás továbbra is problémamentesen működik, ugyanis a CLR megtalálja a külső szerelvényt a C:\MyAsms könyvtárban.

Megjegyzés Ha a fejlesztői gépünkön a szerelvényeket tetszőleges könyvtárakban helyezük el, ezzel tulajdonképpen újra létrehozzuk a rendszerteirő adatbázist (és a hozzá kapcsolódó DLL-poklot), minthogy ha áthelyezzük vagy átnevezzük a bináris fájlok könyvtárát, az aktuális kötés sikertelen lesz. Ennek fényében kezeljük óvatosan a <codebase> elemet.

A <codebase> elem akkor rendkívül hasznos, ha távoli hálózaton működő számítógép szerelvényeire hivatkozunk. Tetelezzük fel, hogy hozzáférési en-gedéllyel rendelkezünk a http://www.IntertechTraining.com cím egyik könyvtárhoz. Ha a helyi számítógép globális szerelvénytárának letöltési gyorsítótárába szeretnénk letölteni a távoli *.d11 fájlt, a <codebase> elemet a következőképpen kell módosítanunk:

```
<codebase version="2.0.0.0"
```

```
href=
```

```
"http://www.intertechning.com/Assemblies/CarLibrarary.dll" />
```

Forráskód A CodeBaseElement alkalmazás megtalálható a 15. fejezet alkönyvtárában.

A System.Configuration névtér

A bemutatott összes konfigurációs fájl jól ismert XML-elemeket alkalmazott, amelyek segítségével a CLR feloldja a külső szereplvények elérési útvonalat. Az ismert elemeken kívüli megoldható az is, hogy az ügyfélalkalmazás konfigurációs fájlja alkalmazásspecifikus adatokat tároljon, amelyek nem kapcsolódnak a kötési heurisztikához. Ezért a .NET Framework olyan névteret biztosít, amellyel programozottan olvashatjuk az ügyfélalkalmazás konfigurációs fájljainak adatait.

A System.Configuration névtér a típusok olyan kisebb gyűjteményét foglalja magában, amelyekkel az ügyfélalkalmazás konfigurációs fájljának egyedi adatait olvashatjuk. Az egyedi beállításokat egy <appsetting> elem hatókörében kell elhelyeznünk. Az <appsetting> elem több <add> elemet tartalmaz, amelyek programozottan a megszerezni kívánt kulcs/érték párokat definiálják.

Tételezzük fel például, hogy az AppConfigReaderApp parancssori alkalmazás app.config fájlja az adatbázis-kapcsolatsztringet, valamint a timesToSayHello adatelemet definiálja:

```
<configuration>
  <appsetting>
    <add key="appConst" value=
      "data source=localhost;initial catalog=Autolo;
      Integrated Security=True" />
    <add key="timesToSayHello" value="8" />
  </appsetting>
</configuration>
```

Egy ügyfélalkalmazásban ezeket az értékeket a System.Configuration.AppSettingsReader típus példányszintű GetValue() metódusával olvashatjuk be. A következő példakódban látható, hogy a GetValue() metódus első paramétere a konfigurációs fájlban a kulcs neve, a második paraméter pedig a kulcs mögöttes típusa (amelyet a C# typeOf operátornak a segítségével szerezhetünk meg):

A gépi szintű konfigurációs fájl

A megvizsgált konfigurációs fájlok egy közös ponttal rendelkeznek: csak bizonyos alkalmazásra érvenyesek (nevelük ezért egyezik meg az indítóalkalmazás nevével). Továbbá minden olyan számítógépen, ahol telepítve van a .NET, található egy `machine.config` fájl, amely rengeteg konfigurációs részletet tartalmaz (az adatok nagy részének semmi köze a külső szerezvények elérési útvonalának a feloldásához), és ezek szabályozzák a .NET platform megváltozott verziójának a működését.

Forráskód Az `AppConfigReader` alkalmazás a 15. fejezet alkönyvtárában található.

Az `AppSettingsReader` osztálytípus *nem* biztosít lehetőséget arra, hogy az alkalmazásspecifikus adatokat konfigurációs fájlba írjuk. Ha azonban programozottnak kell a konfigurációs fájlhoz új <appSettings> elemeket adnunk, a feladatot a `System.Configuration.Configuration` osztálytípus segítségével hajthatjuk végre. (További részletekért forduljunk a .NET Framework 3.5 SDK dokumentációjához.)

```
using System.Configuration;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Reading <appSettings> Data *****\n");
        // Hozzunk létre egy olvasót, és szerezzük meg a kapcsolódási
        // stringeket.
        AppSettingsReader ar = new AppSettingsReader();
        Console.WriteLine(ar.GetValue("appConst", typeof(string)));

        // Most kapjuk meg azt, hogy hányszor mondjuk üdvözlést,
        // majd tegyük is meg!
        int numberOfTimes = (int)ar.GetValue("timesToSayHello",
            typeof(int));
        for(int i = 0; i < numberOfTimes; i++)
            Console.WriteLine("Howdy!");
    }
}
```

A .NET platform keretrendszer a helyi számítógépen telepített keretrendszer-verziók számára külön konfigurációs fájlt tart karban. A .NET 2.0 verzió machine.config fájlja például a C:\WINDOWS\Microsoft.NET\Framework\2.0.50727\CONFIG könyvtárban található (a saját verzióink eltervezéséhez ASP.NET-beállításokat, különböző biztonsági részleteket, hibakeresést és egyéb lehetőségeket szabályoznak. Ha a machine.config fájlt (az <appsetting> elemekkel) az egész gépre kiterjedő alkalmazásbeállításokkal frissíteni, ezt igényeink szerint szabadon meglehet.

A fájlt a Notepad segítségével közvetlenül szerkeszthetjük, ám a helytelen módosítások lehetetlené teszik a futtatórendszer meglehetősen egyszerű módosítások. A helyzet jóval kellemetlenebb, mint a rosszul megtervezett alkalmazás-konfigurációs fájl által okozott problémák, ugyanis az alkalmazás-konfigurációs fájlban az XML-hibák csak egyetlen alkalmazást érintenek, de egy hibás XML-kód a machine.config fájlban tönkretelheti a .NET platform meglehetősen verzióját.

Összefoglalás

A fejezetben annak a részleteit vizsgáltuk meg, hogy a CLR hogyan keresi meg a külsőleg hivatkozott szerelvényeket. Először a szerelvény tartalmát tanulmányoztuk: a fejezkeket, a metaadatok, a manifesztumokat és a köztes nyelvi kódot. Ezt követően egy fájl és több fájl szerelvényeket, valamint ügyfélalkalmazásokat készítettünk.

Találkoztunk a privát és a meglehetősen szerelvényekkel is. A privát szerelvényeket az ügyfélalkalmazás alkönyvtárba másoljuk, míg a meglehetősen szerelvényeket a globális szerelvénytárba telepítjük, feltevé, ha előtte ellátunk őket erős névvel. Végül megleztük, hogy a privát és meglehetősen szerelvényeket ügyféloldali XML konfigurációs fájl vagy kibocsátó házirenddel rendelkező szerelvény segítségével paraméterezhetjük fel.

Típusreflexió, késői kötés és attribútumalapú programozás

A .NET univerzumban a telepítés alapkövei a szerelvények. A Visual Studio 2008 integrált objektumböngészőjének (és más IDE-eszközöknek) a segítségével megvizsgálhatjuk a projekt hivatkozott szerelvényeiben található típusokat. A külső eszközök – például az `ildasm.exe` és a `reflector.exe` – lehetővé teszik, hogy betekintsünk a mögöttes nyelvi kódba, a típusmetaadatokba és az adott .NET bináris szerelvényének az önleírásába. Megtánuuljuk továbbá *programozott* kinyerési ugyanezeket az információkat a `reflection` névtér segítségével. Így most az első feladatunk a reflexió szerepének és a .NET-metaadatok szükségességének a meghatározása lesz.

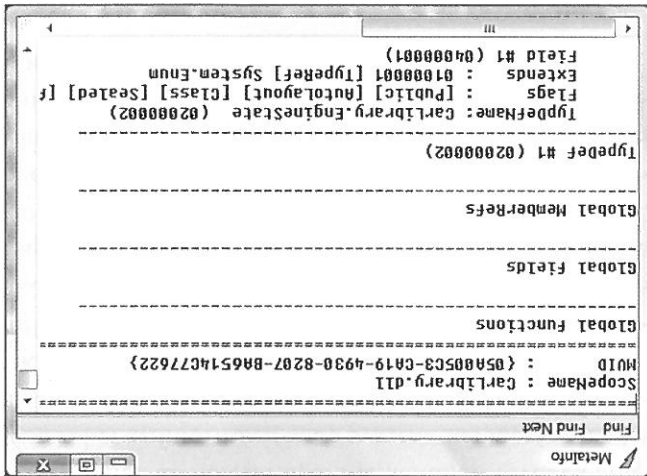
A továbbiakban a reflexióval szolgálatokkal kapcsolatos, egymással szorosan összefüggő témaköröket érintünk. Megvizsgáljuk, hogy a .NET-kliens dinamikus betöltéssel vagy késői kötéssel hogyan aktiválhat olyan típusokat, amelyekről fordítási időben nem rendelkezik információkkal. Megnézzük, hogyan szűrhatunk egyedi metaadatok a .NET-szerelvényeinkbe a rendszer által biztosított, valamint az egyedi attribútumok alkalmazásával. A fejezet a „beépülő modulok” elkészítésének bemutatásával zárul: ezeket a kiterjeszthető Windows Forms-alkalmazásokhoz illeszthetjük.

A metaadatok szükségessége

A típusok (osztályok, interfészek, struktúrák, felsorolások és metódusreferenciák) metaadatokkal történő teljes leírásának lehetősége a .NET platform kulcsfontosságú eleme. Több .NET-technológia, például az objektumsorosítás, a .NET-remoting, az XML-webszolgáltatás és a Windows Communication Foundation (WCF), megköveteli a típusok formátumának futásidői felderítését. Továbbá a nyelvközi együttműködés, több fordítói szolgáltatás és az IDE Intellisense lehetőségei mind a *hippus* konkrét leírására támaszkodnak.

Fontosságuk ellenére (vagy éppen ennek köszönhetően) a metaadatok nem tartoznak a .NET keretrendszer újdonosságai közé. A Java, a CORBA és a COM is rendelkezik hasonló fogalmakkal. A COM-típuskönyvtárak (amelyek alig többek lefordított IDL-kódnál) segítségével írhatjuk le például a COM-kiszolgálókon található típusokat. A COM-technológiához hasonlóan a .NET-kódkönyvtárak is támogatják a típusmetaadatok. A .NET-metaadatok természetesen nem mutatnak szintaktikai hasonlóságot a COM IDL-lel.

Az `itldasm.exe` segédprogramban megnevezhetjük a szerelvénytípus metaadatait a `ctrl` + `m` billentyűkombinációval (lásd az 1. fejezetet). Így, ha a könyvben készített `*.dll` vagy `*.exe` szerelvényeket (például az előző fejezetben létrehozott `CarLibrary.dll` szerelvényt) az `itldasm.exe` segítségével megnyitjuk, a `ctrl` + `m` kombinációval láthatjuk a releváns típusmetaadatot (lásd a 16.1. ábrát).



16.1. ábra: A szerelvény metaadattai

Az `itldasm.exe` részletesen jelenti meg a .NET-típusmetaadatot (a teljesen bináris formátum sokkal tömörebb). Ha a `CarLibrary.dll` szerelvény teljes metaadat-leírását listázzuk, az információkat több oldalon keresztül böngészhetünk. Am ezzel a művelettel kizárólag a papírt pazaroalnánk, nézzük meg inkább a `CarLibrary.dll` szerelvény néhány metaadat-leírásának egy-két kulcsfontosságú részletét.

Az EngineState felsorolás (részleges) metaadata

Az aktuális szerelvényben definiált típusokat a `TypeDef #n` token dokumentálja (ahol a `typedef` a *typedefinit*-t jelenti). Ha a leírni kívánt típus különálló .NET-szerelvényben definiált típusra hivatkozik, a hivatkozott típust a `TypeRef #n` token jelzi (ahol a `TypeRef` a *typusreferencia* rövidítése). A `TypeRef` token egy olyan mutató, amely a külső könyvtárban hivatkozott típus teljes metaadat-definiícióját jelöli. Röviden: a .NET-metaadat táblázatok gyűjteménye, amelyek megmutatják az összes típusdefiniíciót (`TypeDef`) és hivatkozott elemet (`TypeRef`) – ezek mindegyikét megnevezhetjük az `ildasm.exe` metaadatablakában.

A `CarLibrary.dll` szerelvényt illetően az egyetlen `TypeDef`, amellyel találkozhunk, a `CarLibrary.EngineState` felsorolás metaadat-leírása (a számok eltérőek lehetnek; a `TypeDef` számozását a rendszer abban a sorrendben végzi, amelyben a C#-fordító feldolgozza a fájlt):

```

TypeDef #1
-----
TypeDefName: CarLibrary.EngineState (02000002)
Flags       : [Public] [Autolayout] [Class] [Sealed] [AnsiClass]
Extends     : 01000001 [TypeRef] System.Enum
...
Field #2
-----
Field Name: engineActive (04000002)
Flags      : [Public] [Static] [Literal] [HasDefault] (00008056)
DefaultValue: (14) 0
CallConvtn: [FIELD]
Field type: ValueClass CarLibrary.EngineState
...

```

A rendszer a `TypeDefName` token segítségével állapítja meg a meghatározott típus nevét. Az `extends` metaadatokon dokumentálja az adott .NET-típus alapszálját (ebben az esetben a hivatkozott típus a `System.Enum`). A felsorolás minden mezőjét a `Field #n` tokenel jelöljük. Az egyszerűség kedvéért az `engineActive` metaadatot listáztuk.

A Car típus (részleges) metaadata

A car típus részleges kitérítésével a következő kérdések merülnek fel:

- A .NET-metaadat szempontjából hogyan történik a mezők definiálása?
- Hogyan történik a metódusok dokumentálása a .NET-metaadatok segítségével?
- Hogyan történik a tulajdonságok összerendelése, az őket megvalósító két külön taggal (a lekérdező és a módosító taggal)?

```

TypeDef #3
-----
TypeDefName: CarLibrary.Car (02000004)
Flags      : [Public] [Autolayout] [Class] [Abstract] [AnsiClass]
Extends    : 01000002 [TypeRef] System.Object
Field #1
-----
Field Name: petName (04000008)
Flags      : [Family] (00000004)
CallConvntn: [Field]
FieldType: String
Method #1
-----
Method Name: .ctor (06000001)
Flags      : [Public] [Hidebysig] [ReuseSlot] [SpecialName]
RVA        : 0x00002050
ImpFlags   : [IL] [Managed] (00000000)
CallConvntn: [Default]
ReturnType: Void
No arguments.
Property #1
-----
Prop.Name: PetName (17000001)
Flags      : [None] (00000000)
CallConvntn: [Property]
hasThis
ReturnType: String
No arguments.
DefaultValue:
Setter      : (06000004) set_PetName
Getter      : (06000003) get_PetName
0 Others
...

```

A car osztály metaadata jelöli a típus alapoztatályát, és olyan kapcsolókat foglal magában, amelyek leírják a típus összetételét (például, [public], [abstract] stb.). A metódusokat (pl. a car konstruktorunkat) paraméterekkel, visszatérési értékek és a nevük szerint írja le a rendszer, amely a .NET-metadatat setter/getter tokeneknek a segítségével képezi le a tulajdonságokat a belső get/set metódusaira. A származtatott car típusokat (a sportscar és a minivan) hasonló módon írhatjuk le.

A TypeRef

A szerelvény metaadatai nemcsak a belső típusokat (car, EngineState stb.) ismeretik, hanem azokat a külső típusokat is, amelyekre a belső típusok hivatkoznak. A carLibrary.dll szerelvény két felsorolt típust definiált például, így az adatok között a System.Enum típus TypeRef blokkját találjuk:

```
TypeRef #1 (01000001)
-----
Token: 0x01000001
ResolutionScope: 0x23000001
TypeName: System.Enum
MemberRef #1
-----
Member: (0a00000f) ToString:
CallConvntn: [DEFAULT]
hasThis
ReturnType: String
No arguments.
```

A definíció szerelvény leírása

Az ilDasm.exe metaadatbázisban megnevezhetjük azokat a .NET-metadatokokat is, amelyek az Assembly token segítségével magát a szerelvényt írják le. A következő (részleges) listában látható, hogy az Assembly táblázata ugyanazokat az információkat tartalmazza, amelyeket a MANIFEST ikon segítségével is megnevezhetünk. A carLibrary.dll (2.0.0.0 verzió) önleírásának részleges listája a következő:

```
Assembly
-----
Token: 0x20000001
Name : CarLibrary
Public key : 00 24 00 00 04 80 00 00 // etc...
```

A hivatkozott szereplők leírása

```

Hash Algorithm : 0x00008004
Major Version : 0x00000002
Minor Version : 0x00000000
Build Number : 0x00000000
Revision Number: 0x00000000
Locale : <null>
Flags : [sidebysidecompatible] (00000000)

```

Az Assembly tokenen, a Typedef és a TypeRef blokkokon kívül a .NET-meta-adat AssemblyRef #n tokenek segítségével dokumentálják a külső szereplőket. Mint ahogy a CarLibary.dll alkalmazza a MessageBox típust, az adatok között találjuk a System.Windows.Forms osztály AssemblyRef tokenjét:

AssemblyRef #2

```

Token: 0x23000002
Public key or Token: b7 7a 5c 56 19 34 e0 89
Name: System.Windows.Forms
Version: 2.0.3600.0
Major Version: 0x00000002
Minor Version: 0x00000000
Build Number: 0x00000010
Revision Number: 0x00000000
Locale: <null>
HashValue Blob:
Flags: [none] (00000000)

```

Sztringliterálok leírása

A .NET-metaadatok utolsó érdekessége az, hogy az alapkódban minden egyes sztringliterált a user strings token dokumentál, például:

user strings

```

70000001 : (11) L"jamming {0}"
70000019 : (13) L"quiet time..."
70000035 : (23) L"CarLibary Version 2.0!"
70000065 : (14) L"ramming speed!"
70000083 : (19) L"Faster is better..."
700000ab : (16) L"Time to call AAA"
700000cd : (16) L"Your car is dead"

```

A .NET-metaadat rengeteg információt hordoz, és az adott alapkódban található összes belesőleg definiált (és külsőleg hivatkozott) típust listázza.

Felmerülhet a következő kérdés: Hogyan hasznosíthatjuk ezt az információt az alkalmazásainkban? Vagy pedig ellenkezőleg: Miért foglalkozzunk egyáltalán a metaadattal? A két álláspont megismeréséhez vizsgáljuk meg a .NET reflexió szolgáltatásokat.

Megjegyzés A MetaInfo ablakban több CustomAttribute token megjelenik, amelyek az alapkódban alkalmazott attribútumokat dokumentálják. A .NET-attribútumok szerepével a fejezet későbbi részében foglalkozunk.

A reflexió

A .NET univerzumban a *reflexió* futási időben végrehajtott típusmeghatározás. A reflexió szolgáltatás egy barátságos objektummodellen keresztül ugyanazokat a metaadat-információkat teszi programozotlan elérhetővé, amelyeket az `ildasm.exe` segítségével is megjeleníthetünk. A reflexióval listázhatjuk például az adott szerelvényben (vagy *.netmodule fájlban) található összes típust, beleértve az adott típus által definiált metódusokat, mezőket, tulajdonságokat és eseményeket. Dinamikusan felfedezhetjük az adott típus által támogatott interfészeket, a metódus paramétereit és más kapcsolódó részleteket (összetevőket, névtérinformációkat, önleírásokat és így tovább).
Mint a névterek általában, a `system.reflection` névtér is számos kapcsolódó típust foglal magában. A 16.1. táblázat kulcsfontosságú elemeket tartalmaz.

Típus	Valós jelentése
-------	-----------------

Assembly
Ez az osztály olyan metódusokat tartalmaz, amelyek segítségével betölthetjük, vizsgálhatjuk és kezelhetjük a szerelvényt.

AssemblyName
Ennek az osztálynak a segítségével a szerelvény azonosítója mögötti részleteket fedezhetjük fel (verzióinformációk, kultúrainformációk stb.).

EventInfo
Ez az osztály egy adott esemény adatait tárolja.

FieldInfo
Ez az osztály egy adott mező adatait tárolja.

MemberInfo
Ez az absztrakt osztály definiálja az `EventInfo`, a `FieldInfo`, a `MethodInfo` és a `PropertyInfo` típusok közös viselkedését.

Tipus	Valós jelentése
ISAbstract	Ezeknek a tulajdonságoknak a segítségével fedezhetjük fel a hivatkozott típus néhány alapvető jellemzőjét
ISArray	
ISClass	(pl. hogy absztrakt metódus, tömb vagy beágyazott osztály-c).
ISCOMObject	
ISenum	
ISgenericTypeDefinition	
ISGenericParameter	
ISInterface	
ISPrimitive	
ISNestedPrivate	
ISNestedPublic	
ISsealedValueType	

A System.Type osztály

Ahhoz, hogy a `System.Reflection` névtér alkalmazásával programozottan olvassuk a .NET-metaadatot, meg kell ismerkednünk a `System.Type` osztállyal.

Tipus	Valós jelentése
MethodInfo	Ez az osztály egy adott metódus adatait tárolja.
Module	Ez az osztály hozzáférést biztosít egy többfajlos szerelvény egyes moduljaihoz.
ParameterInfo	Ez az osztály egy adott paraméter adatait tárolja.
PropertyInfo	Ez az osztály egy adott tulajdonság adatait tárolja.

16.1. táblázat: A `System.Reflection` névtér néhány tagja