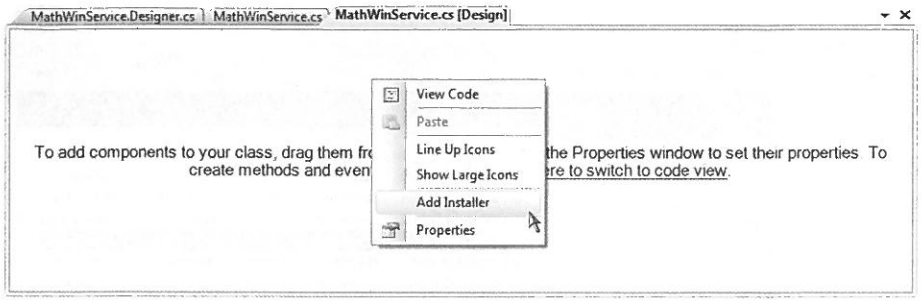


25.16. ábra: *Telepítő hozzáadása a Windows-szolgáltatáshoz*

Ha készen vagyunk, látni fogjuk, hogy két új komponenst adtunk az új tervezőfelülethez. Az első komponens (a neve az alapértelmezés szerint serviceProcessInstaller1) képviseli azt a típust, amely telepíti az új Windows-szolgáltatást a célszámítógépen. Válasszuk ki ezt a típust a tervezőben, és a Properties ablakban állítsuk az `Account` tulajdonság értékét `LocalSystemre`.

A második komponens (a neve serviceInstaller1) azt a típust képviseli, amely telepíti a Windows-szolgáltatásunkat. A Properties ablakban újra változtassuk a `ServiceName` tulajdonságot `MathOrderService`-re (ahogy már kitalálhattuk, ezt a barátságos megjelenített nevet láthatjuk a regisztrált Windows-szolgáltatás neveként), a `StartType` tulajdonságot állítsuk `Automatic` értékre, és adjunk felhasználóbarát leírást a Windows-szolgáltatásról a `Description` tulajdonságban. Most lefordíthatjuk az alkalmazást.

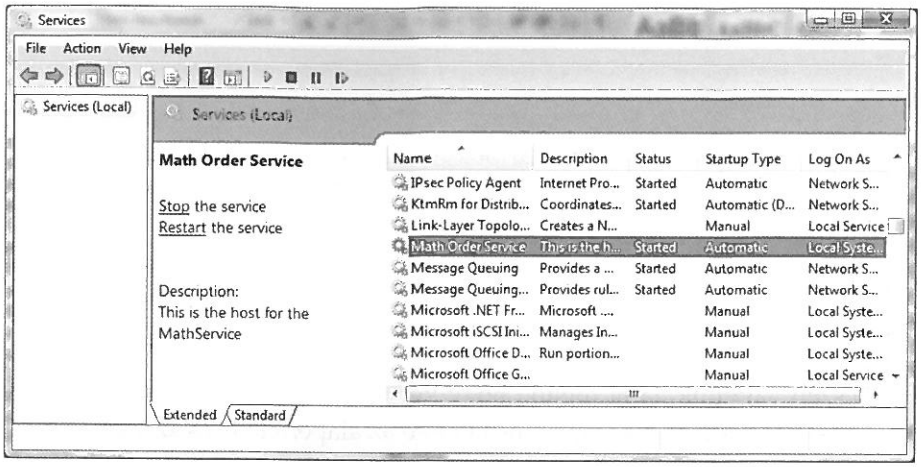
A Windows-szolgáltatás telepítése

A Windows-szolgáltatást a hosztgépre a hagyományos telepítőprogrammal (pl. `*.msi` telepítővel) vagy az `installutil.exe` parancssori eszközzel telepíthetjük. A Visual Studio 2008 parancssorával lépjünk a MathWindowsService-Host projekt \bin\Debug könyvtárába. Írjuk be a következő parancsot:

```
installutil MathWindowsServiceHost.exe
```

Ha a telepítés sikeres volt, megnyithatjuk a Szolgáltatások appletet a Vezérlőpult Felügyeleti eszközök mappájában. Itt ábécésorrendbe rendezett listában látnunk kell a Windows-szolgáltatásunk barátságos nevét. Ha megtaláltuk, elindíthatjuk a szolgáltatást a helyi gépen (lásd a 25.17. ábrát).

Ha a szolgáltatás működik, az utolsó lépés a kliensalkalmazás elkészítése, amely használhatja a szolgáltatásokat.



25.17. ábra: *A WCF-szolgáltatásunkat hosztoló Windows-szolgáltatás megtekintése*

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

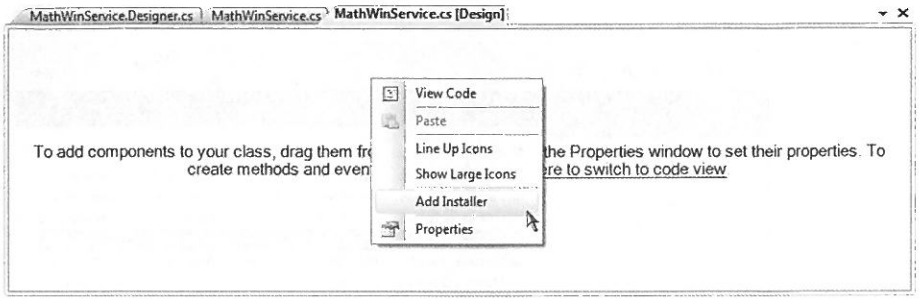
Szolgáltatás aszinkron hívása

Készítsünk új parancssori alkalmazást MathClient névvel, és a Visual Studio Add Service Reference opciójával állítsuk a Service Reference értékét a futó WCF-szolgáltatásra, amelyet a háttérben futó Windows-szolgáltatás hosztol. De még ne kattintsunk az OK gombra (lásd a 25.18. ábrát)!

Az Add Service Reference párbeszédablak bal alsó sarkában található egy Advanced gomb. Kattintsunk a gombra, és további proxykonfigurációs beállításokat tekinthetünk meg (lásd a 25.19. ábrát).

A párbeszédablakban a Generate Asynchronous Operators jelölőnégyzet bekapcsolásával választhatjuk olyan kód generálását, amellyel aszinkron módon hívhatunk távoli metódusokat. Egyelőre jelöljük be ezt az opciót.

A párbeszédablakban másik fontos lehetőség az Add Web Reference gomb. Ha már rendelkezünk némi háttértudással az XML-szolgáltatások készítéséről a Visual Studio 2005-ös vagy korábbi verziójában, tudjuk, hogy az Add Service Reference lehetőség helyett Add Web Reference állt a rendelkezésünkre. Ha erre a gombra kattintunk, olyan proxykódot kapunk, amellyel `*.asmx` fájlban leírt hagyományos webszolgáltatással tudunk kommunikálni.



25.16. ábra: *Telepítő hozzáadása a Windows-szolgáltatáshoz*

Ha készen vagyunk, látni fogjuk, hogy két új komponenst adtunk az új tervezőfelülethez. Az első komponens (a neve az alapértelmezés szerint serviceProcessInstaller1) képviseli azt a típust, amely telepíti az új Windows-szolgáltatást a célszámítógépen. Válasszuk ki ezt a típust a tervezőben, és a Properties ablakban állítsuk az `Account` tulajdonság értékét `LocalSystemre`.

A második komponens (a neve serviceInstaller1) azt a típust képviseli, amely telepíti a Windows-szolgáltatásunkat. A Properties ablakban újra változtassuk a `ServiceName` tulajdonságot `MathOrderService`-re (ahogy már kitalálhattuk, ezt a barátságos megjelenített nevet láthatjuk a regisztrált Windows-szolgáltatás neveként), a `StartType` tulajdonságot állítsuk `Automatic` értékre, és adjunk felhasználóbarát leírást a Windows-szolgáltatásról a `Description` tulajdonságban. Most lefordíthatjuk az alkalmazást.

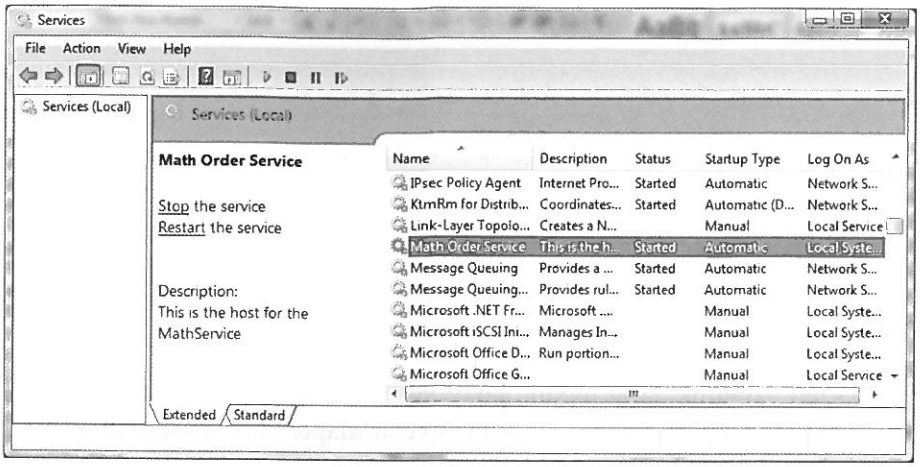
A Windows-szolgáltatás telepítése

A Windows-szolgáltatást a hosztgépre a hagyományos telepítőprogrammal (pl. `*.msi` telepítővel) vagy az `installutil.exe` parancssori eszközzel telepíthetjük. A Visual Studio 2008 parancssorával lépjünk a MathWindowsService-Host projekt \bin\Debug könyvtárába. Írjuk be a következő parancsot:

```
installutil MathWindowsServiceHost.exe
```

Ha a telepítés sikeres volt, megnyithatjuk a Szolgáltatások appletet a Vezérlőpult Felügyeleti eszközök mappájában. Itt ábécésorrendbe rendezett listában látnunk kell a Windows-szolgáltatásunk barátságos nevét. Ha megtaláltuk, elindíthatjuk a szolgáltatást a helyi gépen (lásd a 25.17. ábrát).

Ha a szolgáltatás működik, az utolsó lépés a kliensalkalmazás elkészítése, amely használhatja a szolgáltatásokat.



25.17. ábra: *A WCF-szolgáltatásunkat hosztoló Windows-szolgáltatás megtekintése*

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

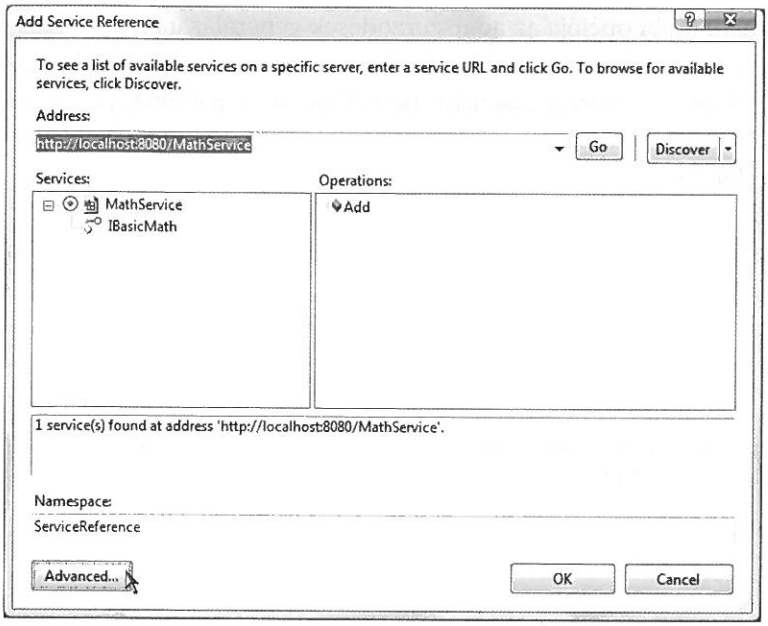
Szolgáltatás aszinkron hívása

Készítsünk új parancssori alkalmazást MathClient névvel, és a Visual Studio Add Service Reference opciójával állítsuk a Service Reference értékét a futó WCF-szolgáltatásra, amelyet a háttérben futó Windows-szolgáltatás hosztol. De még ne kattintsunk az OK gombra (lásd a 25.18. ábrát)!

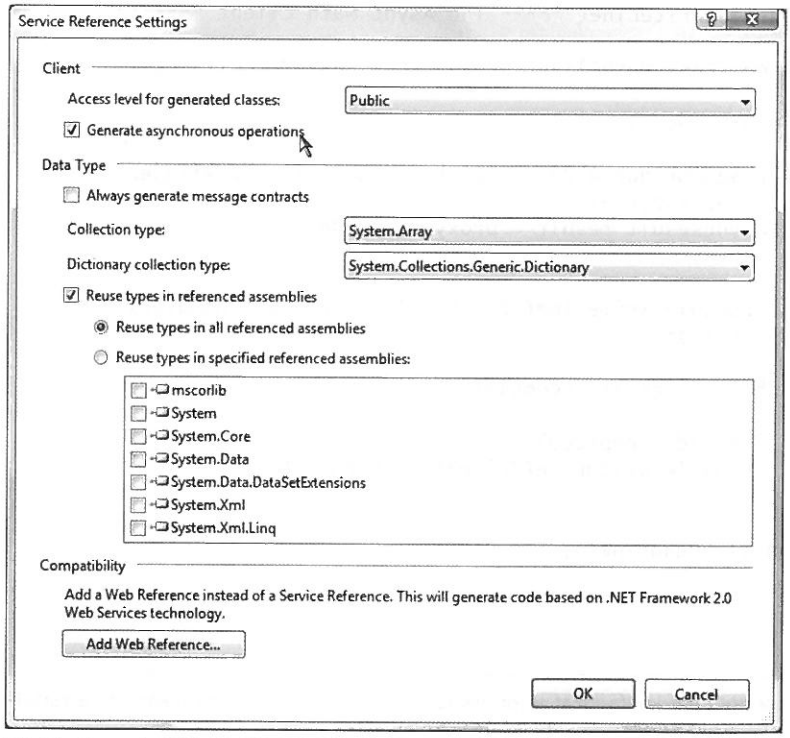
Az Add Service Reference párbeszédablak bal alsó sarkában található egy Advanced gomb. Kattintsunk a gombra, és további proxykonfigurációs beállításokat tekinthetünk meg (lásd a 25.19. ábrát).

A párbeszédablakban a Generate Asynchronous Operators jelölőnégyzet bekapcsolásával választhatjuk olyan kód generálását, amellyel aszinkron módon hívhatunk távoli metódusokat. Egyelőre jelöljük be ezt az opciót.

A párbeszédablakban másik fontos lehetőség az Add Web Reference gomb. Ha már rendelkezünk némi háttértudással az XML-szolgáltatások készítéséről a Visual Studio 2005-ös vagy korábbi verziójában, tudjuk, hogy az Add Service Reference lehetőség helyett Add Web Reference állt a rendelkezésünkre. Ha erre a gombra kattintunk, olyan proxykódot kapunk, amellyel `*.asmx` fájlban leírt hagyományos webszolgáltatással tudunk kommunikálni.



25.18. ábra: *Referencia beállítása az ourMathService-re*



25.19. ábra: *Speciális ügyféloldali proxykonfigurációs beállítások*

A párbeszédablak többi opciója az adatszerződések generálásának beállításaira szolgál, amelyeket a fejezet későbbi részében még megvizsgálunk. Feltétlenül kapcsoljuk be a Generate Asynchronous Operators jelölőnégyzetet, és kattintsunk az OK gombra minden párbeszédablakban, hogy visszatérjünk a Visual Studio IDE-hez.

A proxykód olyan további metódusokat tartalmaz, amelyekkel az elvárt `Begin/End` aszinkron meghívási móddal hívhatjuk a szolgáltatásszerződés minden tagját (lásd az előző kötet 18. fejezetében). Az alábbiakban egy egyszerű implementációt találunk, amely az erősen típusos `AsyncCallback` metódusreferencia helyett `lambda` kifejezést használ.

```
using System;
using MathClient.ServiceReference;
using System.Threading;

namespace MathClient
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** The Async Math Client *****\n");

      using (BasicMathClient proxy = new BasicMathClient())
      {
        proxy.Open();

        // Számok hozzáadása aszinkron módon lambda kifejezés
        // segítségével.
        IAsyncResult result = proxy.BeginAdd(2, 3,
          ar =>
        {
          Console.WriteLine("2 + 5 = {0}", proxy.EndAdd(ar));
        }, null);

        while (!result.IsCompleted)
        {
          Thread.Sleep(200);
          Console.WriteLine("Client working...");
        }
      }
      Console.ReadLine();
    }
  }
}
```

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

WCF-adatszerződések tervezése

A fejezet utolsó példájában WCF-*adatszerződéseket* készítünk. Az előző WCF-szolgáltatások nagyon egyszerű metódusokat definiáltak primitív CLR-adattípusokkal történő műveletekhez. Ha bármilyen webszolgáltatás-központú kötéstípust használunk (`basicHttpBinding`, `wsHttpBinding` stb.), a bejövő és kimenő adattípusokat a rendszer automatikusan XML-elemekké formázza a `System.Runtime.Serialization.XmlFormatter` típus segítségével, amelyet a `System.Runtime.Serialization.dll` szerelvény definiál. Ugyancsak ide tartozik, hogy ha TCP-alapú kötést használunk (mint a `netTcpBinding`), az egyszerű adattípusok paraméterei és visszatérési értékei kompakt bináris formátumot kapnak.

Megjegyzés A WCF futtatókörnyezet szintén automatikusan kódol bármilyen, `[Serializable]` attribútummal jelölt típust.

Ha paraméterként vagy visszatérési értékként egyedi típusokat használó szolgáltatásszerződéseket definiálunk, ezeket a típusokat nekünk kell adatszerződéssel meghatároznunk. Tulajdonképpen az adatszerződés `[DataContract]` attribútummal ellátott típus. Minden mezőt, amelyet az ajánlott szerződésben várhatóan használni fogunk, hasonló módon a `[DataMember]` attribútummal jelöljük.

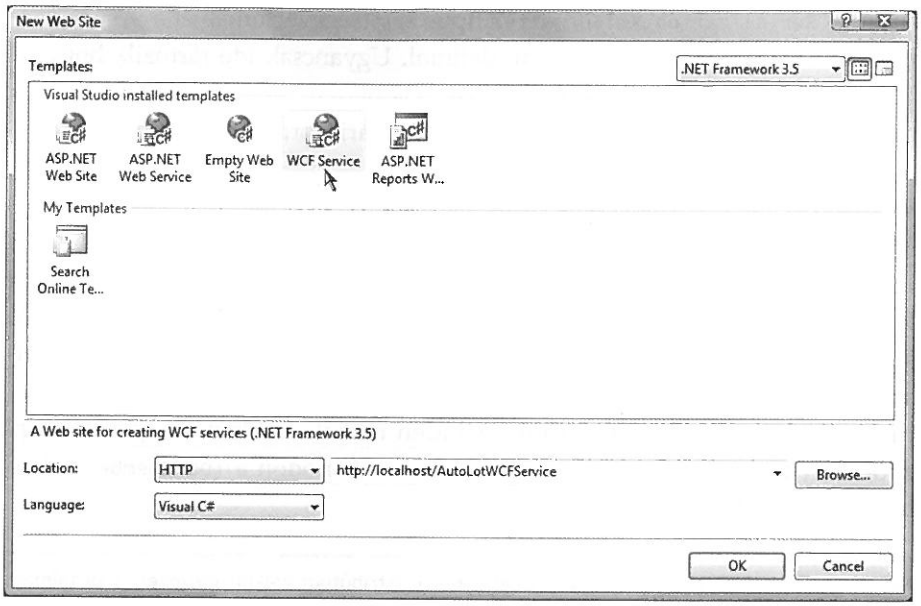
Megjegyzés Ha az adatszerződés `[DataMember]` attribútum nélküli mezőket is tartalmaz, ezeket a WCF-futtatókörnyezet nem sorosítja.

Az adatszerződések készítésének bemutatására hozzunk létre teljesen új WCF-szolgáltatást, amely együttműködik a 22. fejezetben készített AutoLot adatbázissal. Ez az utolsó WCF-szolgáltatás a webalapú WCF Service sablonnal készül. Az ilyen típusú WCF-szolgáltatás automatikusan az IIS-beli virtuális könyvtárba kerül, és a hagyományos .NET XML-webszolgáltatásához hasonlóan működik. Ha megértjük ezen WCF-szolgáltatás felépítését, nem jelent problémát létező WCF-szolgáltatás áthelyezése új IIS-beli virtuális könyvtárba.

Megjegyzés Ez a példa feltételezi, hogy valamennyire tisztában vagyunk az IIS-beli virtuális könyvtárak (és maga az IIS) szerkezetével. A 31. fejezetben megtaláljuk a részleteket.

Webközpontú WCF Service projektsablon használata

Készítsünk új WCF-szolgáltatást AutoLotWCFService néven a File > New > Web Site menüponttal, amely a következő URI-ről érhető el: http://localhost/AutoLotWCFService (lásd a 25.20. ábrát). A Location legördülő listában feltétlenül a HTTP-értéket válasszuk.



25.20. ábra: *Web-központú WCF-szolgáltatás készítése*

Ha ezzel készen vagyunk, állítsunk be referenciát a 22. fejezetben készített `AutoLotDAL.dll` szerelvényre (a Website > Add Reference menüponttal). A WCF Service Library projekthez hasonlóan rendelkezünk kezdeti példakóddal (az App_Code könyvtárban található), amelyet tanácsos törölni. Először nevezzük át az eredeti `IService.cs` fájlt `IAutoLotService.cs`-re, és definiáljuk a kezdeti szolgáltatásszerződést az átnevezett fájlban:

```
[ServiceContract]
public interface IAutoLotService
{
  [OperationContract]
  void InsertCar(int id, string make, string color, string petname);

  [OperationContract]
  void InsertCar(InventoryRecord car);
```

```
    [OperationContract]
    InventoryRecord[] GetInventory();
}
```

Ez az interfész három metódust definiál, amelyek közül az egyik `InventoryRecord` típusú tömböt ad vissza (ezt még létre kell hozni). Az InventoryDAL `GetInventory()` metódusa korábban egyszerűen egy `DataTable` objektumot adott vissza, így feltehetjük a kérdést, hogy a szolgáltatásunk `GetInventory()` metódusa miért nem ezt teszi.

Míg a WCF-szolgáltatásmetódusunk megtehetné, hogy `DataTable` típust ad vissza, a WCF célja, hogy figyelembe vegye a SOA-elveket, amelyek egyike az, hogy szerződésekkel és nem implementációkkal programozunk. Ezért ahelyett, hogy a .NET-specifikus `DataTable` típust adnánk a külső hívónak, az egyedi adatszerződést (`InventoryRecord`) adjuk vissza, amelyet a rendszer a csatolt WSDL-dokumentumban agnosztikus módon kifejt.

Az interfész az `InsertCar()` túlterhelt metódust definiálja. Az első verzió négy bejövő paramétert fogad, a második verzió pedig bemenetként `InventoryRecord` típust. Az adatszerződést a következőképpen definiálhatjuk:

```
[DataContract]
public class InventoryRecord
{
  [DataMember]
  public int ID;
  [DataMember]
  public string Make;
  [DataMember]
  public string Color;
  [DataMember]
  public string PetName;
}
```

Ha így szeretnénk implementálni az interfészt, akkor készítsünk egy hosztot, próbáljuk meghívni ezeket a metódusokat a kliensből, és azt fogjuk tapasztalni, hogy futásidejű kivétel jelentkezik. Ennek az oka az, hogy a WSDL-leírás egyik követelménye, hogy minden, egy adott végpontról elérhető metódus *egyedi névvel* kell, hogy rendelkezzen. Így, míg a metódusfelültöltés a C#-ban remekül működik, az aktuális webszolgáltatási specifikációk nem engedik meg két azonos nevű `InsertCar()` metódus használatát.

Szerencsére az `[OperationContract]` attribútum támogat egy megnevezett tulajdonságot (`Name`), amellyel meghatározhatjuk a C#-metódus megjelenítésének módját a WSDL-leírásban. Emiatt módosítsuk az `InsertCar()` második verzióját:

```
public interface IAutoLotService
{
  ...
  [OperationContract(Name = "InsertCarWithDetails")]
  void InsertCar(InventoryRecord car);
}
```

Szolgáltatásszerződés implementálása

Az `AutoLotService` típus a következő interfészt implementálja (mindenképpen importáljuk az `AutoLotConnectedLayer` és a `System.Data` névteret a kódfájlban):

```
using AutoLotConnectedLayer;
using System.Data;

public class AutoLotService : IAutoLotService
{
  private const string ConnString =
          @"Data Source=(local)\SQLEXPRESS;Initial Catalog=AutoLot"+
          ";Integrated Security=True";
  public void InsertCar(int id, string make, string color,
                        string petname)
  {
    InventoryDAL d = new InventoryDAL();
    d.OpenConnection(ConnString);
    d.InsertAuto(id, color, make, petname);
    d.CloseConnection();
  }

  public void InsertCar(InventoryRecord car)
  {
    InventoryDAL d = new InventoryDAL();
    d.OpenConnection(ConnString);
    d.InsertAuto(car.ID, car.Color, car.Make, car.PetName);
    d.CloseConnection();
  }

  public InventoryRecord[] GetInventory()
  {
    // Először olvassuk be a DataTable objektumot az adatbázisból.
    InventoryDAL d = new InventoryDAL();
    d.OpenConnection(ConnString);
    DataTable dt = d.GetAllInventory();
    d.CloseConnection();

    // A rekordokhoz készítünk List<T> listát.
    List<InventoryRecord> records = new List<InventoryRecord>();
```

```
        // Az adattáblát másoljuk az egyedi szerződéseket tartalmazó
        // List<> listába.
        DataTableReader reader = dt.CreateDataReader();
        while (reader.Read())
        {
          InventoryRecord r = new InventoryRecord();
          r.ID = (int)reader["CarID"];
          r.Color = ((string)reader["Color"]).Trim();
          r.Make = ((string)reader["Make"]).Trim();
          r.PetName = ((string)reader["PetName"]).Trim();
          records.Add(r);
        }

        // Átalakítjuk a List<T> listát InventoryRecord típusú tömbbé.
        return (InventoryRecord[])records.ToArray();
      }
    }
```

Az egyszerűség kedvéért a kapcsolatsztring értékét ahelyett, hogy a `Web.config` fájlban tárolnánk, kódban adjuk meg (amelyet a gép beállításainak megfelelően lehet, hogy módosítani kell). Mivel az adatelérési könyvtár végzi a kommunikációt az AutoLot adatbázissal, a mi dolguk csak annyi, hogy a bejövő paramétereket átadjuk az `InventoryDAL` osztálytípus `InsertAuto()` metódusának. Másik érdekesség a `DataTable` objektum értékeinek leképezése `InventoryRecord` típusok általános listájába (`DataTableReader` használatával), utána pedig a `List<T>` lista átalakítása `InventoryRecord` típusú tömbbé.

A *.svc fájl szerepe

Webközpontú WCF-szolgáltatás készítése során a projekt egy adott `*.svc` kiterjesztésű fájlt tartalmaz. Erre a fájlra minden IIS-hosztolt WCF-szolgáltatásnak szüksége van, ez írja le a szolgáltatás implementációjának nevét és helyét a telepítési ponton belül. Mivel megváltoztattuk az eredeti fájlok és WCF-típusok nevét, most módosítanunk kell a `Service.svc` fájl tartalmát is:

```
<%@ ServiceHost Language="C#" Debug="true"
                Service="AutoLotService"
                CodeBehind="~/App_Code/AutoLotService.cs" %>
```

A Web.config fájl módosítása

A szolgáltatás tesztelése előtt az utolsó feladatunk a `Web.config` fájl `<system.serviceModel>` részének módosítása. A `Web.config` fájl szerepe hasonlít a végrehajtható fájl `*.config` fájl szerepéhez, de az előbbi számos webspecifikus beállítást is vezérel (lásd később részletesen az ASP.NET-ről szóló részben). A példában nem kell mást tennünk, mint a következőképpen módosítani a WCF-specifikus részt a fájlban:

```
<system.serviceModel>
  <services>
    <service name="AutoLotService"
             behaviorConfiguration="ServiceBehavior">
      <endpoint address="" binding="wsHttpBinding"
                contract="IAutoLotService"/>
      <endpoint address="mex" binding="mexHttpBinding"
                contract="IMetadataExchange"/>
    </service>
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior name="ServiceBehavior">
        <serviceMetadata httpGetEnabled="true"/>
        <serviceDebug includeExceptionDetailInFaults="false"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

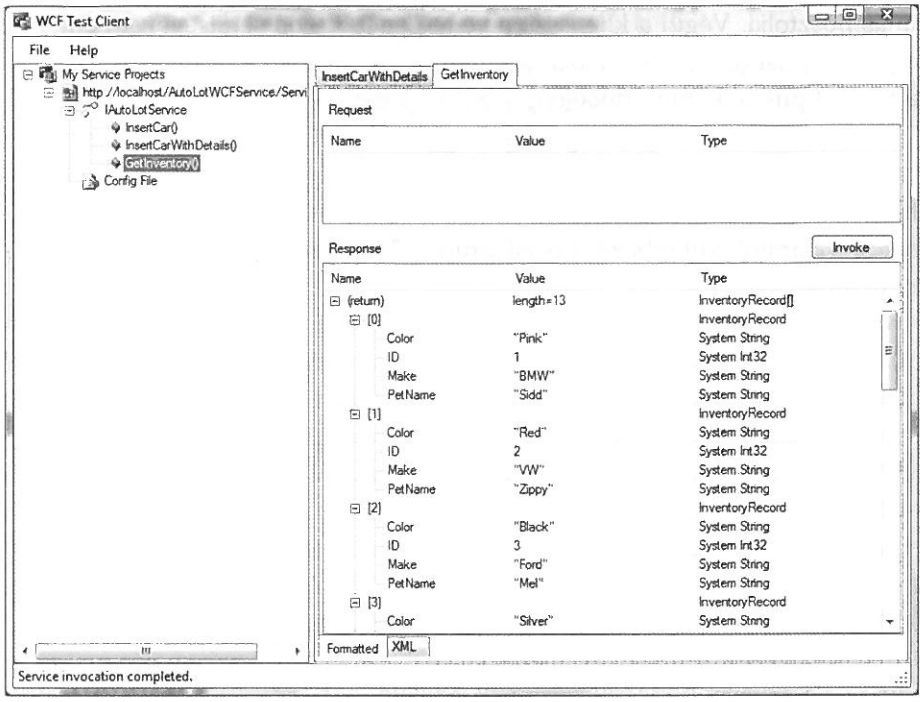
A szolgáltatás tesztelése

Most már bármilyen klienst készíthetünk a szolgáltatás tesztelésére, beleértve az `*.svc` fájl végpontjának átadását a `WcfTestClient.exe` alkalmazásnak:

```
WcfTestClient http://localhost/AutoLotWCFService/Service.svc
```

Nézzük meg a 25.21. ábrát, amely a `GetInventory()` hívásának eredményét mutatja.

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



25.21. ábra: *Web-központú WCF-szolgáltatás készítése*

Egyedi kliensalkalmazás készítése során használjuk az Add Service Refence párbeszédablakot, ahogyan ezt a fejezet korábbi részében a MagicEightBallServiceClient és a MathClient példaprojektekben tettük.

Összefoglalás

A fejezet bevezetett a Windows Communication Foundation (WCF) API használatába, amely a .NET 3.0 verziója óta az alaposztálykönyvtárak része. A WCF fő célja, hogy egységes objektummodellt kínáljon számos (korábban egymással nem kapcsolódó) elosztott API összefogására. Továbbá a fejezet elején láttuk, hogy a WCF-szolgáltatást a megadott címek, kötések és szerződések képviselik (könnyen megjegyezhető az ABC [address, binding, contract] rövidítésből).

Megnéztük, hogy egy tipikus WCF-alkalmazás három kapcsolódó szerelvényt használ. Az első definiálja a szolgáltatás funkcionalitását képviselő szolgáltatásszerződéseket és a szolgáltatástípusokat. Ezt a szerelvényt azután egyedi végrehajtható fájl, IIS-beli virtuális könyvtár vagy Windows-szolgál-

használó elküldi a megrendelést, rengeteg tevékenység indul el. Kezdhetjük a hitelkeret ellenőrzésével. Ha a felhasználó megfelel a hitelellenőrzésünknek, elindítunk egy adatbázis-tranzakciót azzal a céllal, hogy eltávolítsuk a bejegyzést az Inventory táblából, új bejegyzést hozzunk létre az Orders táblában, és frissítsük a vevő számlainformációját. Miután az adatbázis-tranzakció befejeződött, visszaigazoló e-mailt küldhetünk a vevőnek, majd meghívhatunk egy távoli XML-webszolgáltatást, hogy a rendelést eljuttassuk a kereskedéshez. Mindezek együttesen egyetlen üzleti folyamatot képviselnek.

Történeti szempontból az üzleti folyamat modellezése újabb olyan részlet volt, amelyről a programozóknak kellett gondoskodniuk gyakran egyedi kód létrehozásával azért, hogy az üzleti folyamat ne csak megfelelően modellezett, hanem magában az alkalmazásban megfelelően futtatható legyen. Szükségünk lehet kód létrehozására például a hibahelyek ellenőrzéséhez, a nyomkövetéshez és a naplózás támogatásához (hogy lássuk az adott üzleti folyamat mit csinál); tárolási támogatásához (menteni egy hosszú ideig futó folyamat állapotát); valamint egyéb műveletekhez. Ahogy saját tapasztalataink ból tudhatjuk, az ilyen infrastruktúra felépítése a semmiből rengeteg időt és manuális munkát igényel.

Tételezzük fel, hogy egy fejlesztőcsapat az alkalmazásaikhoz elkészítette egy egyedi üzleti folyamat keretrendszerét, de a munkájuk még nem teljes. A nyers C#-kódot nem könnyű elmagyarázni annak, aki nem programozó, ám *szintén* meg kell értenie az üzleti folyamatot. Igazság szerint a menedzserek, az üzletkötők és a grafikus tervezői csapat tagjai gyakran nem ismerik a programozási nyelvet. Éppen ezért szükségünk lesz egyéb modellező eszközök használatára is (mint pl. a Microsoft Visio), hogy grafikus módszerekkel, szaktudást nem igénylő kifejezésekkel bemutathassuk a folyamatokat. A nyilvánvaló probléma az, hogy két egységet kell összhangban tartanunk: ha megváltoztatjuk a kódot, frissítenünk kell a diagramot. Ha megváltoztatjuk a diagramot, frissítenünk kell a kódot.

Továbbá, ha kifinomult szoftveralkalmazást hozunk létre *100%-os kódmegközelítéssel*, a forráskód csak nyomokban tartalmazza az alkalmazás belső „folyamát". Egy tipikus .NET-program például egyedi típusok százaiból épülhet fel (nem is említve az alaposztálykönyvtárak által használt számtalan típust). Bár a programozó valószínűleg érzi, hogy melyik objektum hív meg egy másik objektumot, maga a forráskód nagyon messze áll egy olyan élethű dokumentumtól, amely a tevékenység teljes folyamatát kifejti. Noha a fejlesztőcsapat létrehozhat külső dokumentációt és munkafolyamat-ábrákat, ismét beleütközünk ugyanannak a folyamatnak a többszörös megjelenítési problémájába.

A WF szerepe

A .NET 3.0 verziójának megjelenése óta rendelkezésünkre áll a Windows Workflow Foundation API. Lényegében a WF a programozók számára deklaratív módon teszi lehetővé üzleti folyamatok tervezését, előre elkészített *tevékenységek* felhasználásával. Ahelyett, hogy az adott üzleti tevékenység és a szükséges infrastruktúra megjelenítéséhez szerelvények egyedi készletét építenénk fel, a Visual Studio 2008 WF-tervezőjével létrehozhatjuk saját üzleti folyamatunkat a tervezés ideje alatt. Így a WF lehetővé teszi egy üzleti folyamat vázának felépítését, amelyet aztán a kódban egészíthetünk ki.

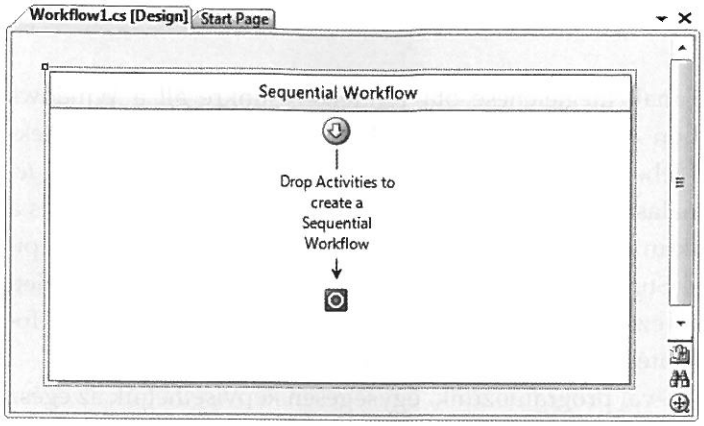
Amikor a WF-API-val programozunk, egységesen képviselhetjük az egész üzleti folyamatot, valamint a folyamatot definiáló kódot. Mivel egyetlen WF-dokumentummal megjeleníthetjük a folyamatot vezérlő kódot, valamint létrehozhatjuk a folyamat barátságos vizuális ábrázolását, kizárható, hogy a dokumentumok szinkronizálása megbomoljon. A WF-dokumentum világosan bemutatja magát a folyamatot.

A WF építőkockái

Egy munkafolyamat-alkalmazás létrehozása „másmilyen", mint egy tipikus .NET-alkalmazásé. Eddig például minden példakód új projektworkspace létrehozásával kezdődött (leggyakrabban a parancssori konzolalkalmazásával), és a forráskód hosszadalmas elkészítésével mutatta be a programot. A WF-alkalmazás is egyedi kódot foglal magában, de egyúttal *közvetlenül a szerelvényben* hozzuk létre magát az üzleti folyamatot. Nézzük meg a 26.1 ábrát, amely bemutatja a Visual Studio 2008 által létrehozott kezdeti munkafolyamat-diagramot új Sequential Workflow parancssorikonzolalkalmazás-projekt választásakor.

A tervező használatával (és a Visual Studióba épített különböző WF-központú eszközökkel) modellezhetjük a folyamatunkat, és végül olyan kódot készíthetünk, amely a megfelelő körülmények között végrehajtja a folyamatot. A későbbiekben részletesen megvizsgáljuk ezeket az eszközöket.

A WF sokkal több, mint egy kellemes tervező, amely lehetővé teszi egy üzleti folyamat tevékenységeinek modellezését. A WF-diagram készítésekor a tervezőeszközök létrehozzák azt a forráskódot, amely a folyamatunk vázát képviseli. A legfontosabb az, hogy a vizuális WF-diagram végrehajtható kódja nem pusztán egyszerű Visio-szerű statikus kép. A WF-technológiát más .NET-technológiákhoz hasonlóan .NET-szerelvények, névterek és típusok képviselik.



26.1. ábra: *Üres szekvenciális munkafolyamat-diagram tervezője*

A WF futtatókörnyezete

A WF-diagram valós típusoknak és egyedi kódnak felel meg, így a WF-API magában foglal egy futtatómotort a munkafolyamat beolvasásához, végrehajtásához, eltávolításához és egyéb kezelési műveleteihez. A WF-futtatómotor hosztolható bármely .NET-alkalmazástartományban, ám egyetlen alkalmazástartomány a WF-motor egyetlen futó példányát foglalhatja magában.

Az alkalmazástartomány egy Win32-folyamaton belüli partíció (lásd az előző kötet 17. fejezetét), amely egy .NET-alkalmazás és a külső kódkönyvtárak számára a hosztot jelenti. A WF-motor beágyazható egyszerű parancssori programba, GUI asztali alkalmazásába (Windows Forms vagy Windows Presentation Foundation [WPF]), vagy hozzáférhetővé tehetjük WCF-szolgáltatásból vagy XML-webszolgáltatásból.

Ha olyan üzleti folyamatot modellezünk, amelyet a rendszerek széles választéka használ, akkor lehetőségünk van WF létrehozására C#-osztálykönyvtárprojekten belül. Így az új alkalmazások egyszerűen a `*.dll` fájlunkra hivatkoznak ahhoz, hogy az üzleti folyamatok előre definiált gyűjteményét újrafelhasználják. Így értelemszerűen nem kell többször létrehoznunk ugyanazt a WF-et.

A WF-API teljes értékű objektummodellel rendelkezik, amely lehetővé teszi számunkra, hogy programozottan együttműködjünk a futtatómotorral, valamint az általunk tervezett munkafolyamatokkal.

A WF alapvető szolgáltatásai

A tervezőeszközökön, a tevékenységeken és a futtatómotoron kívül a WF rendelkezik beépített szolgáltatásokkal is, amelyek teljessé teszik a munkafolyamat-alkalmazás általános keretrendszerét. A szolgáltatások felhasználásával sokat „örökölhetünk" a leginkább szükséges WF-infrastruktúrákból, ahelyett, hogy időt és energiát pazarolnánk ezeknek az infrastruktúráknak a manuális felépítésére. A 26.1. táblázat bemutatja a WF-API beépített belső szolgáltatásait.

A 26.1 táblázatban látható négy belső szolgáltatást együtt *alapvető szolgáltatások*nak nevezzük. A WF-API-k a szolgáltatások mindegyikéhez alapértelmezett implementációt biztosítanak, kettő közülük kötelező (az ütemezés és a tranzakciók), míg a nyomkövetés és a rögzítés opcionális, továbbá alapértelmezés szerint a rendszer nem regisztrálja őket a futtatómotorral. Minthogy a .NET-alaposztálykönyvtárak minden alapvető szolgáltatás támogatásához rendelkeznek típusokkal, ezeket lehetőségünk van lecserélni saját egyedi megvalósításainkra. A hosszú futásidejű munkafolyamatok rögzítésének módját is testre szabhatjuk. Ha szeretnénk a szolgáltatás alapműködését új funkcionalitással bővíteni, ez is módunkban áll.

Szolgáltatás	Jelentés
Rögzítés	Ez a szolgáltatás lehetővé teszi WF-példány külső forrásba (pl. adatbázisba) történő mentését. Ez akkor nagyon hasznos, ha egy hosszú futásidejű üzleti folyamat hosszú ideig tétlen.
Tranzakció	A WF-példányokat tranzakciós környezetben is vizsgálhatjuk, és biztosíthatjuk, hogy a munkafolyamat minden része – vagy a munkafolyamat egy részhalmaza – önálló elemi részként sikeresen (vagy sikertelenül) működjön.
Nyomkövetés	A szolgáltatás elsődlegesen egy WF-tevékenység hibakeresésére és optimalizálására szolgál, lehetővé teszi egy adott munkafolyamat tevékenységeinek a megfigyelését.
Ütemezés	A szolgáltatás lehetővé teszi annak szabályozását, hogy a WF-futtatómotor hogyan kezeli a munkafolyamatok szálait.

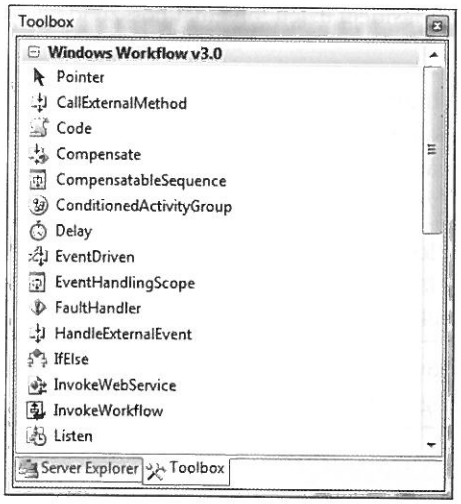
26.1. táblázat: *A WF belső szolgáltatásai*

Amikor létrehozzuk a WF-futtatómotor egy példányát azzal a céllal, hogy az egyik munkafolyamatunkat futtassuk, lehetőségünk van az `AddService()` metódus meghívására, hogy beillesszük a nyomkövetés vagy a rögzítés szolgáltatásobjektumokat (mint pl. a `SqlWorkflowPersistanceService` és a `SqlTrackingService`), valamint valamely általunk tervezett egyedi szolgáltatást. Most már képesek vagyunk egy adott WF-példány futtatására, és a lehetővé tehetjük a kiegészítő szolgáltatások számára a példány élettartamának felügyeletét.

Ebben a fejezetben nem hozzuk létre az alapvető szolgáltatások egyedi megvalósításait, és nem mélyedünk bele ezek alapértelmezett funkcionalitásába sem, pusztán a munkafolyamat-alkalmazás építőelemeire összpontosítunk, és megvizsgálunk több WF-tevékenységet. Az alapvető szolgáltatásokkal kapcsolatos további információkért forduljunk a .NET Framework 3.5 SDK dokumentációjához.

A WF-tevékenységek első megközelítésben

A WF célja az, hogy deklaratív módon tegye lehetővé egy üzleti folyamat modellezését, amelyet majd a WF-futtatómotor hajt végre. A WF tekintetében egy üzleti folyamat tetszőleges számú *tevékenységből* áll. A WF-tevékenység a teljes folyamat egy elemi „lépése". Amikor egy új WF-alkalmazást hozunk létre a Visual Studio 2008-cal, az eszköztárban egy Windows Workflow területet találunk, amely a beépített tevékenységek ikonjait tartalmazza (lásd a 26.2. ábrát).



26.2. ábra: *A Windows Workflow eszköztára*

Megjegyzés A Visual Studio 2008 a tevékenységek eszköztárát .NET 3.0-s és .NET 3.5-ös tevékenységkategóriákra osztja. A Windows Workflow csomópont alatti tevékenységek lehetővé teszik, hogy együttműködjünk a Windows Communication Foundtaion (WCF-) szolgáltatásokkal.

A .NET 3.5 több olyan beépített tevékenységet nyújt, amelyekkel üzleti folyamatainkat modellezhetjük, ezek mindegyikét valós típusokra képezhetjük le a `System.Workflow.Activities` névtérben, és így megjelenítésük és irányításuk kóddal történik. A fejezetben több ilyen a beépített tevékenységeket alkalmazunk. A 26.2. táblázat a kapcsolódó működés szerint csoportosítva ismerteti néhány hasznos tevékenység magas szintű működését.

Tevékenység	Jelentés
`CodeActivity`	Ez a tevékenység a munkafolyamatokban végrehajtható egyedi kódegységet képviseli.
`If ElseActivity`, `WhileActivity`	Ezek a tevékenységek alapvető ciklus- és elágazástámogatásokat nyújtanak a munkafolyamatban.
`InvokeWebServiceActivity`, `WebServiceInput Activity`, `WebServiceOutputActivity`, `WebServiceFaultActivity`	Ezek a tevékenységek lehetővé teszik a munkafolyamat számára, hogy együttműködjön az XML-webszolgáltatással.
`SendActivity`, `ReceiveActivity`	Ezek a tevékenységek lehetővé teszik, hogy együttműködjünk a Windows Communication Foundation szolgáltatásaival. Ne feledjük, hogy ez a két tevékenység .NET 3.5-specifikus.
`ConditionedActivity`, `GroupActivity`	Ezek a tevékenységek biztosítják a kapcsolódó tevékenységek csoportjának definiálását, amelyeket a rendszer akkor hajt végre, ha egy adott feltétel teljesül.
`DelayActivity`, `SuspendActivity`, `TerminateActivity`	Ezek a tevékenységek egy munkafolyamatban lehetővé teszik a várakozási idők, valamint a tevékenység szüneteltetésének vagy leállításának definiálását.
`EventDrivenActivity`, `EventHandlingScopeActivity`	Ezek a tevékenységek a munkafolyamatban lehetővé teszik CLR-események hozzárendelését adott tevékenységhez.

Tevékenység	Jelentés
`ThrowActivity`, `FaultHandlerActivity`	Ezek a tevékenységek lehetővé teszik kivételek előidézését és kezelését a munkafolyamatban.
`ParallelActivity`, `SequenceActivity`	Ezek a tevékenységek lehetővé teszik tevékenységek sorozatának párhuzamos vagy szekvenciális végrehajtását.

26.2. táblázat: *A belső WF-tevékenységek áttekintése*

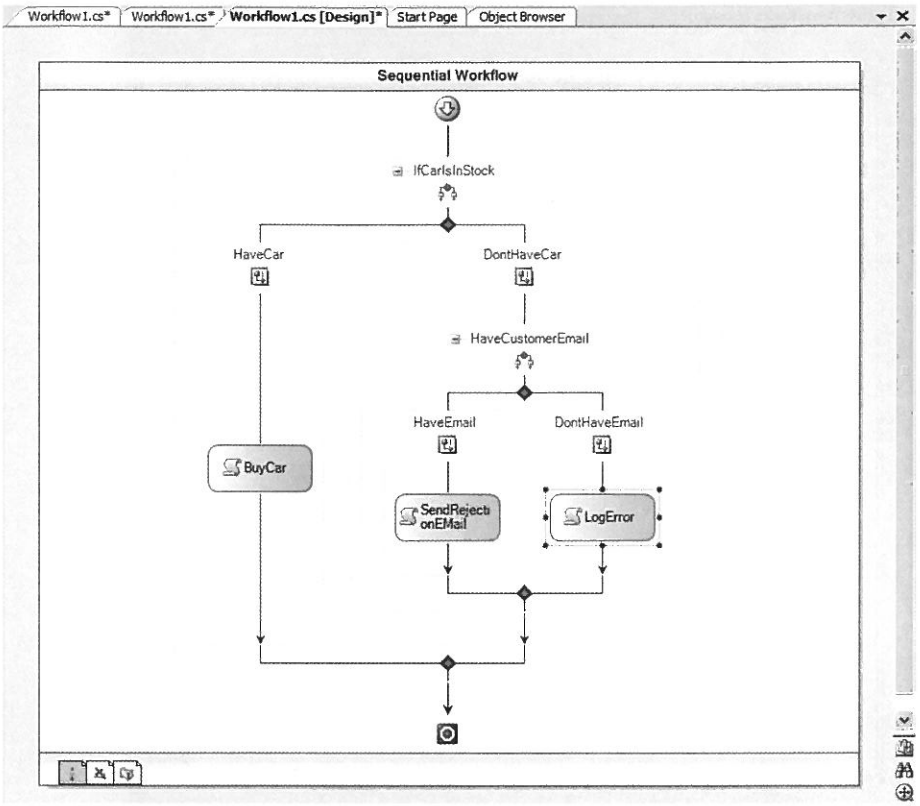
Bár nagyon sok, minden WF-alkalmazáshoz biztos alapot nyújtó beépített tevékenység létezik már most is, lehetőségünk van olyan újabb egyedi tevékenységek létrehozására is, amelyek zökkenőmentesen illeszkednek a Visual Studio integrált fejlesztői környezetébe és a WF-futtatómotorba.

Szekvenciális és állapotgép-munkafolyamatok

A WF-API az üzletifolyamat-munkafolyamatok két fajtájának modellezését támogatja: a szekvenciális munkafolyamatokat és az állapotgép-munkafolyamatokat. Végtére is mindkét kategória tetszőleges kapcsolódó tevékenységek összeillesztésével jön létre; így tulajdonképpen végrehajtásuk *módja* különbözteti meg őket.

A leglényegretörőbb munkafolyamat-típus a *szekvenciális*. Ahogy a neve sugallja, a szekvenciális munkafolyamat lehetővé teszi egy olyan üzleti folyamat modellezését, ahol az egyes tevékenységeket a rendszer sorban hajtja végre az utolsó tevékenység befejezéséig. Ez nem azt jelenti, hogy a szekvenciális munkafolyamat szükségszerűen lineáris vagy kiszámítható jellegű – létrehozhatunk olyan szekvenciális a munkafolyamatot is, amely különböző elágazási és ciklikus tevékenységeket tartalmaz, valamint olyan tevékenységeket, amelyek végrehajtása párhuzamosan külön szálakon történik.

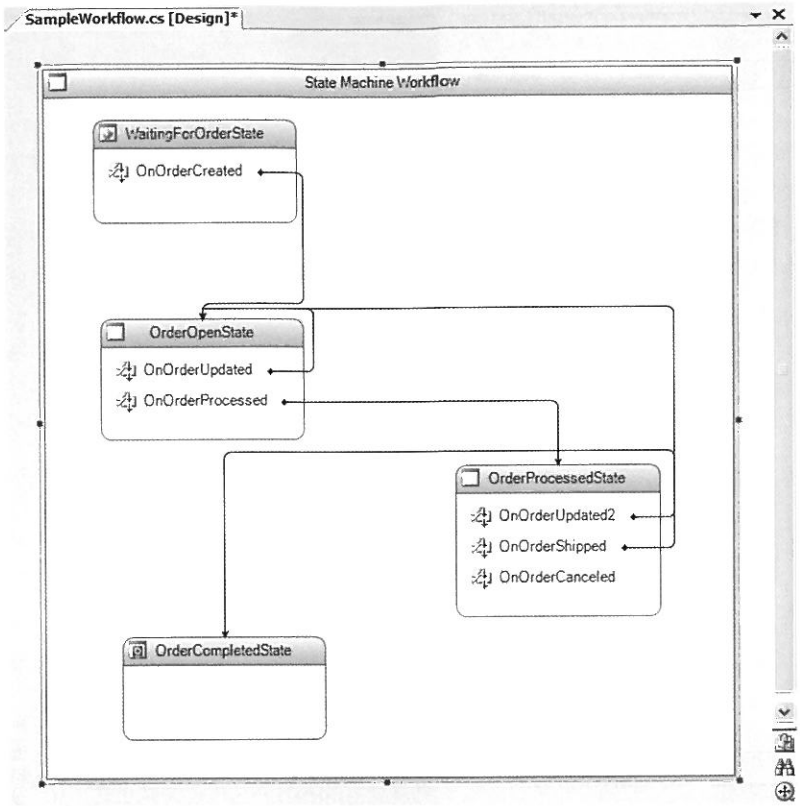
A szekvenciális munkafolyamat kulcseleme az, hogy kristálytiszta kezdő- és végponttal rendelkezik. A Visual Studio 2008 munkafolyamat-tervezőben a futási útvonal a WF-diagram tetején kezdődik, és lefelé halad a végpontig. A 26.3. ábrán egyszerű szekvenciális munkafolyamat látható. Egy olyan üzleti folyamat részletét mutatja be, amely azt ellenőrzi, hogy egy adott gépkocsi raktáron van-e.



26.3. ábra: *A szekvenciális munkafolyamatok egyértelmű kezdő- és végponttal rendelkeznek*

A szekvenciális munkafolyamatok akkor működnek, ha a munkafolyamat különböző rendszerszintű entitások közötti együttműködést modellez, és ha nincs szükség a folyamatban visszaléptetésre. A 26.3 ábrán modellezett üzleti folyamatnak például két lehetséges kimenete van: az autó vagy raktáron van, vagy nincs. Ha az autó valóban raktáron van, a rendelést a rendszer az egyedi kód valamely részletének használatával dolgozza fel (bármi legyen is az). Ha az autó nincs raktáron, értesítő e-mailt küldünk, ha az ügyfél e-mail címe a rendelkezésünkre áll.

A szekvenciális munkafolyamatokkal ellentétben az *állapotgép-munkafolyamatok* a tevékenységeket nem egyszerű lineáris útvonal segítségével modellezik, hanem több *kérésállapotot* és kapcsolódó *eseményeket* definiálnak, amelyek kiváltják az állapotok közötti átmeneteket. A 26.4. ábrán egyszerű állapotgép-WF-diagram látható, amely egy rendelés feldolgozását mutatja be. A munkafolyamatban bármely kérés néhány belső esemény hatására különféle állapotba kerülhet.



26.4. ábra: *Az állapotgép-munkafolyamatok nem követnek rögzített, lineáris útvonalat*

Az állapotgép-munkafolyamatok jól hasznosíthatók olyan üzleti folyamatok modellezésénél, amelyek különböző végrehajtási állapotban lehetnek, rendszerint annak köszönhetően, hogy az állapotok közötti átmenethez emberi beavatkozás szükséges. Adott egy kérésállapot, amely a megrendelés létrehozására vár. Ha ez megtörténik, egy esemény a tevékenység folyamatát a megrendelés nyitott állapotához irányítja, amely kiváltja a megrendelés feldolgozott állapotát (vagy visszatér az előző nyitott állapothoz), és így tovább.

Megjegyzés Ebben a fejezetben nem foglalkozunk az állapotgép-munkafolyamatok felépítésének részleteivel. További információkért forduljunk a .NET Framework 3.5 SDK dokumentációjához.

WF-szerelvények, -névterek és -projektek

Programozói szempontból a WF-et három alapvető szerelvény képviseli:

- `System.Workflow.Activities.dll`: A beépített tevékenységeket és az azokat irányító szabályokat definiálja.
- `System.Workflow.Runtime.dll`: Azokat a típusokat definiálja, amelyek a WF-futtatómotort és az egyedi munkafolyamat-példányokat képviselik.
- `System.Workflow.ComponentModel.dll`: Számos olyan típust definiál, amelyek lehetővé teszik a WF-alkalmazások tervezési idejű támogatását.

Ezek a szerelvények több .NET-névteret definiálnak, sokat közülük különböző WF vizuális tervezőeszközök a háttérben használnak. A 26.3. táblázat néhány figyelemreméltó WF-központú névteret mutat be.

Névtér	Jelentés
`System.Workflow.Activities`	Ez a fő tevékenység-központú névtér, amely típusdefiníciókat definiál a Windows Workflow eszköztárának minden egyes eleméhez. További alnévterek definiálják a szabályokat, amelyek irányítják, valamint a típusokat, amelyek konfigurálják ezeket a tevékenységeket.
`System.Workflow.Runtime`	Ez a névtér típusokat definiál, amelyek a WF-futtatómotort (mint a `WorkflowRuntime`) és (a `WorkflowInstance` révén) egy adott munkafolyamat egy példányát képviselik.
`System.Workflow.Runtime.Hosting`	Ez a névtér típusokat szolgáltat hoszt létrehozásában a WF futtatómotorjához, amely egyedi WF-szolgáltatásokat használ (nyomkövetés, naplózás stb.). Továbbá a névtér típusokat szolgáltat a beépített alapvető WF-szolgáltatások megjelenítéséhez.

26.3. táblázat: *Alapvető WF-névterek*

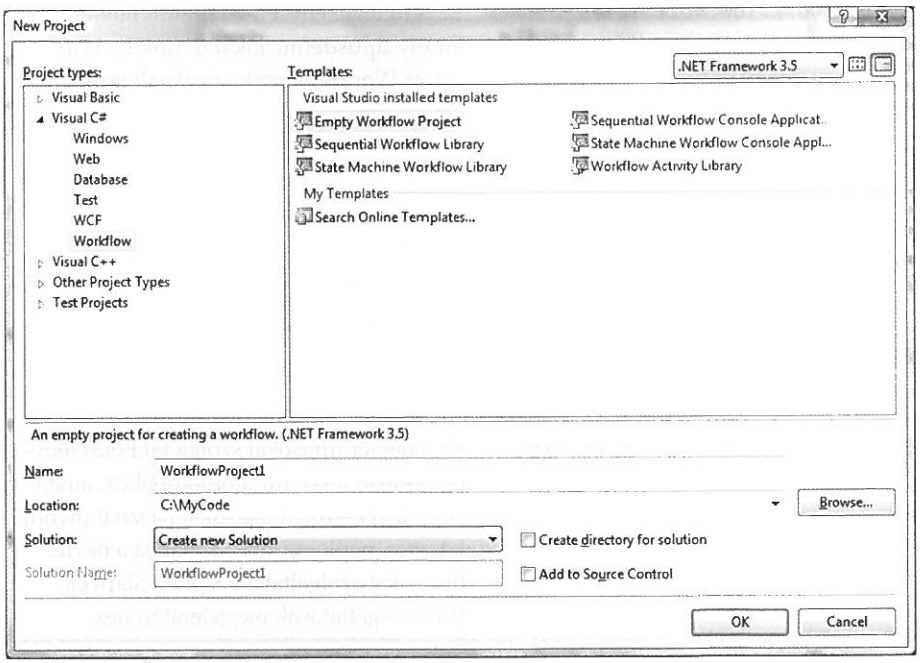
A .NET 3.5 WF-támogatása

A .NET 3.5 megjelenésével az alaposztálykönyvtárak egy negyedik WF-központú szerelvénnyel bővültek, ez a `System.WorkflowServices.dll`. Itt további olyan típusokat találhatunk, amelyek lehetővé teszik WF-alkalmazások létrehozását, ezek pedig integrálhatók a Windows Communications Foundation API-kkal. A szerelvény legfontosabb tulajdonsága az, hogy a `System.Workflow.Activities` névteret olyan új típusokkal bővíti, amelyek támogatják a WCF-integrációt.

Megjegyzés Amikor WF-képes Visual Studio 2008-as projektet készítünk, az IDE minden egyes Windows Workflow Foundation szerelvényhez automatikusan beállítja a referenciákat.

Visual Studio munkafolyamat-projektsablonok

A Visual Studio 2008 IDE rengeteg WF-projektsablont biztosít. Először is, amikor kiválasztjuk a File > New > Project párbeszédablakot, egy Workflow csomópontot találjuk a C# programozási kategóriája alatt (lásd a 26.5 ábrát).



26.5. ábra: *Az alapvető WF-projektsablonok*

Itt találjuk a projekteket, amelyek lehetővé teszik szekvenciális és állapotgép-munkafolyamat felépítését egyszerű konzolos vagy egyedi `*.dll` szerelvényen belül.

A New Project párbeszédablak Windows Communication Foundation (WCF-) csomópontja is rendelkezik WF-sablonokkal (lásd 26. fejezet). Itt találunk két projektsablont (Sequential Workflow Service Library és State Machine Workflow Service Library), amelyek lehetővé teszik olyan WCF-szolgáltatás létrehozását, amely belsőleg használja a munkafolyamatot. Ebben a fejezetben nem használjuk a WF-projektsablonoknak ezt a csoportját, ám emlékezzünk rá, amikor WCF-szolgáltatásokat építünk, új projekt létrehozásánál dönthetünk a WF-funkcionalitás integrálása mellett.

A munkafolyamat menete

Amikor WF-API segítségével programozunk, végeredményben *üzleti folyamatot* próbálunk modellezni, így az első lépés az üzleti folyamat vizsgálata lesz, beleértve minden állépést a folyamaton belül és minden lehetséges kimenetelt. Tételezzük fel például, hogy egy online tanfolyami regisztrációs folyamatot modellezünk. Amikor érkezik egy kérés, mit tehetünk, ha a kereskedők nincsenek az irodában? Mi történik, ha a tanfolyam betelt? Mi történik, ha a tanfolyamot törölték, vagy új időpontra helyezték át? Hogyan tudjuk meghatározni, hogy az oktató elérhető-e, nincs-e szabadságon, nem oktat-e egy másik tanfolyamon ugyanazon a héten, és így tovább?

Az aktuális hátterünktől függően a követelmények összegyűjtésének folyamata teljesen új feladatként jelenthet meg, hiszen az üzleti folyamat feltérképezése „valaki más problémája" lehet. Kisebb cégeknél a szükséges üzleti folyamat meghatározásának terhe a fejlesztők vállát nyomhatja. Nagyobb szervezetek jellemzően üzleti elemzőket alkalmaznak, akik elvállalják az üzleti folyamat feltárásának (és gyakran a modellezésének) a szerepét.

Mindenesetre a WF-fel dolgozni nem egyszerűen egy „ugorjunk neki, és kezdjünk kódolni" kísérlet. Ha a kódolás előtt nem szánunk időt arra, hogy tisztán elemezzük az üzleti problémát, amelyet megpróbálunk megoldani, akkor egészen bizonyosan problémáink lesznek. Ebben a fejezetben a hangsúlyt a munkafolyamat-tervezés alaptechnikáira, a tevékenységek használatára és a vizuális WF-tervezőkkel folytatott munka lehetőségeire fektetjük. Az életszerű munkafolyamatok jóval összetettebbé válhatnak.

Egyszerű munkafolyamat-alkalmazás létrehozása

Az első WF-példánk nagyon egyszerű szekvenciális folyamatot modellez. A cél az, hogy létrehozzunk egy olyan munkafolyamatot, amely bekéri a felhasználó nevét, és ellenőrzi az eredményt. Amíg az eredmény nem felel meg az üzleti szabályainknak, a nevet újra és újra bekérjük.

Először hozzunk létre a UserDataWFApp Sequential Workflow parancssorialkalmazás-projektet. Majd a Solution Explorer segítségével nevezzük át a kezdeti WF-tervező fájlját `Workflow1.cs`-ről a találóbb `ProcessUsernameWorkflow.cs` névre.

A kezdeti munkafolyamathoz tartozó kód vizsgálata

Mielőtt tevékenységeket használnánk az üzleti folyamatunk ábrázolásához, nézzük meg, belsőleg hogyan jelenik meg ez a kezdeti diagram. Ha a Solution Explorer segítségével vizsgáljuk a `ProcessUsernameWorkflow.cs` fájlt, látható, hogy más tervező által felügyelt fájlokhoz (űrlapok, ablakok, weboldalak) hasonlóan a WF-diagram részleges osztálydefiníciókból áll. Ha megnyitjuk a `*.cs` fájlt, olyan osztálytípust találunk, amely a `SequentialWorkflowActiviy` típusból származik, és egy alapértelmezett konstruktort tartalmaz, amit az `InitializeComponent()` metódust hívja meg:

```
public sealed partial class ProcessUsernameWorkflow :
                            SequentialWorkflowActivity
{
  public ProcessUsernameWorkflow()
  {
    InitializeComponent();
  }
}
```

Megjegyzés A WF-fejlesztés egyik elve, hogy a munkafolyamatok önálló elemi részek. Ennek a ténynek köszönhetően a munkafolyamat-osztálytípusok explicit módon lezártak, így megakadályozzuk, hogy a leszármazott típusok számára szülőosztályként működjenek.

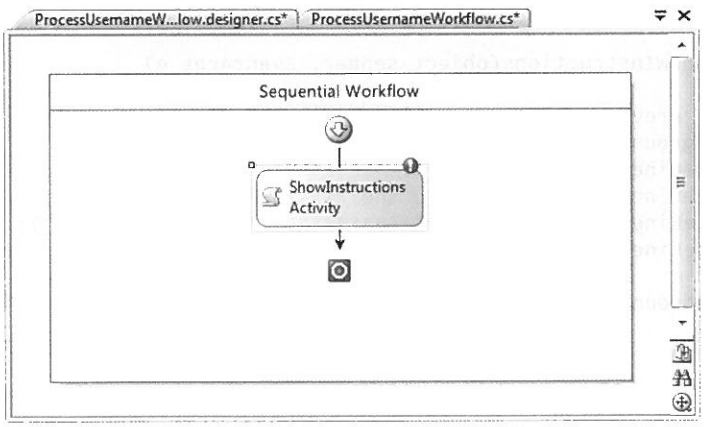
Ha megnyitjuk a kapcsolódó `*.Designer.cs` fájlt, látható, hogy az `InitializeComponent()` metódus beállította a `Name` tulajdonságot:

```
partial class ProcessUsernameWorkflow
{
  [System.Diagnostics.DebuggerNonUserCode]
  private void InitializeComponent()
  {
    this.Name = "ProcessUsernameWorkflow";
  }
}
```

Ha a Windows Workflow eszköztár segítségével különböző tevékenységeket húzunk a tervezőfelületre, és ezeket a Properties ablak (vagy az inline intelligens címkék) használatával konfiguráljuk, a rendszer a `*.Designer.cs` fájlt automatikusan frissíti. Más IDE-felügyelt fájlokhoz hasonlóan, teljesen figyelmen kívül hagyhatjuk a kódot ebben a fájlban, és az elsődleges `*.cs` fájlon belüli kód létrehozására összpontosíthatunk.

A Code tevékenység hozzáadása

Az első tevékenység, amelyet a sorozathoz hozzáadunk, egy Code tevékenység. Ehhez fogjunk meg a Code tevékenység elemet a Windows Workflow eszköztárról, és húzzuk a munkafolyamat kezdő és végpontját összekötő egyenesre. Ezután a Properties ablak segítségével nevezzük át ezt a tevékenységet ShowInstructionsActivityre. Ekkor a tervezőnk a 26.6. ábrához hasonlóan fog kinézni.



26.6. ábra: *Egy (nem egészen tökéletes) Code tevékenység*

A tervező éppen hibát jelez, amelyet a Code tevékenység tetején található piros felkiáltójelre kattintva nézhetünk meg. A hiba arról tájékoztat, hogy az ExecuteCode értéket nem adtuk meg, pedig ez egy kötelező lépés minden Code tevékenység típushoz. Az ExecuteCode adja meg annak a metódusnak a nevét, amelyet a rendszer végrehajt, amikor a feladat a WF-futtatómotorral találkozik.

A Properties ablakban az ExecuteCode értékét állítsuk a ShowInstructions metódusra. Amikor megnyomjuk az Enter billentyűt, az IDE a következő kódcsonkkal egészíti az elsődleges *.cs fájlt:

```
public sealed partial class ProcessUsernameWorkflow :
                            SequentialWorkflowActivity
{
  public ProcessUsernameWorkflow()
  {
    InitializeComponent();
  }

  private void ShowInstructions(object sender, EventArgs e)
  {
  }
}
```

Valójában az ExecuteCode a CodeActivity osztálytípus egy eseménye. Amikor a WF-motor a szekvenciális munkafolyamat ezen fázisával találkozik, az ExecuteCode esemény elsül, amit a ShowInstructions() metódus kezel. Ezt a metódust azokkal a Console.WriteLine() utasításokkal valósítsuk meg, amelyek alapvető utasításokat jelenítenek meg a végfelhasználó számára:

```
private void ShowInstructions(object sender, EventArgs e)
{
  ConsoleColor previousColor = Console.ForegroundColor;
  Console.ForegroundColor = ConsoleColor.Yellow;
  Console.WriteLine("****************************************");
  Console.WriteLine("**** Welcome to the first WF Example ****");
  Console.WriteLine("****************************************\n");
  Console.WriteLine("I will now ask for your name and validate
                     the data...\n");
  Console.ForegroundColor = previousColor;
}
```

While tevékenység hozzáadása

A szekvenciális folyamat a végfelhasználótól a nevét kéri mindaddig, amíg az érték egy egyedi üzleti szabály szerint (amelyet még definiálnunk kell) elfogadható nem lesz. Az ilyen ciklikus viselkedés a `While` tevékenységgel jeleníthető meg. A `While` tevékenységgel olyan kapcsolódó tevékenységeket definiálhatunk, amelyeket a rendszer folyamatosan végrehajt addig, amíg egy meghatározott feltétel nem teljesül.

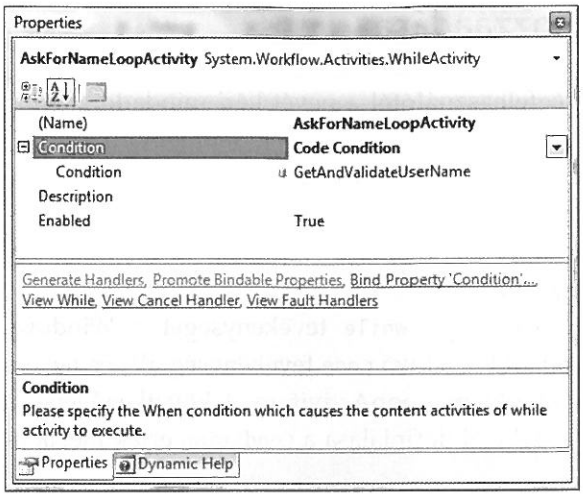
Ennek bemutatásához húzzunk egy `While` tevékenységet a Windows Workflow eszköztárból közvetlenül az előző `Code` tevékenység alá, és nevezzük át az új tevékenységet AskForNameLoopActivityre. A következő lépés a ciklus befejezéséhez szükséges feltétel definiálása a `Condition` érték megadásával a Properties ablakban.

A `Condition` érték (amely sok tevékenység számára közös tulajdonság) két alapvető módon adható meg. Először is létrehozhatunk egy *kódfeltételt*. Ahogy a neve sugallja, az opció egy olyan metódus meghatározását teszi lehetővé az osztályunkban, amelyet a tevékenység hív meg annak eldöntésére, hogy folytassa-e a munkát. Hogy erről a tényről tájékoztassa a tevékenységet, a megadott metódusnak szüksége van egy logikai visszatérési értékre (`true` az ismétléshez, `false` a befejezéshez).

A `Condition` érték megadásának másik módja egy *deklaratív szabályfeltétel* létrehozása. Ez az opció lehetővé teszi egyetlen olyan kódutasítás meghatározását, amely a feldolgozást követően `true` vagy `false` értéket kap; ezt az utasítást nem kódoljuk a szerelvényünkbe, hanem egy `*.rules` fájlban tároljuk. A szemlélet egyik előnye az, hogy a szabályok deklaratív módosítását teszi lehetővé.

A feltételünk azon az egyedi kódon alapul, amelyet még létre kell hoznunk. Ehhez az első lépés a Code Condition opció kiválasztása a Condition lehetőség értékeiből, majd annak a metódusnévnek a megadása, amely a tesztet végrehajtja. A Properties ablak segítségével adjuk a metódusnak a `GetAndValidateUserName` nevet (lásd a 26.7. ábrát).

Amint megadtuk a `While` tevékenység teszteléséhez a kódfeltétel nevét, az IDE létrehoz egy metódusrészletet, amelynek második paramétere `ConditionalEventArgs`-típusú. Ez a típus tartalmazza a `Result` tulajdonságot, amelyet `true` vagy `false` értékre állíthatunk az általunk modellezett feltétel teljesülésétől vagy sikertelenségétől függően.



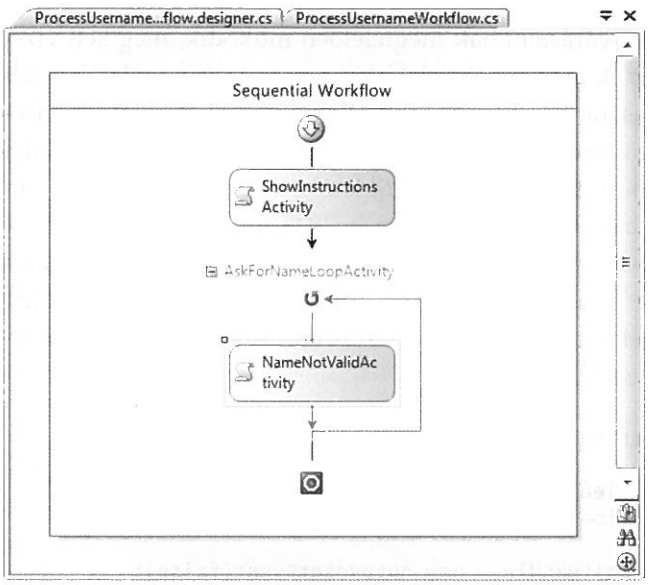
26.7. ábra: *A While tevékenység beállítása*

Adjuk az új `UserName` sztringtípusú automatikus tulajdonságot a `ProcessUsernameWorkflow` osztályunkhoz. A `GetAndValidateUserName()` metódus hatókörén belül kérjük a felhasználótól a neve megadását, és ha a név kevesebb, mint tíz karakterből áll, a `Result` tulajdonságot ennek megfelelően állítsuk `true` értékre. A kérdéses módosítás a következő:

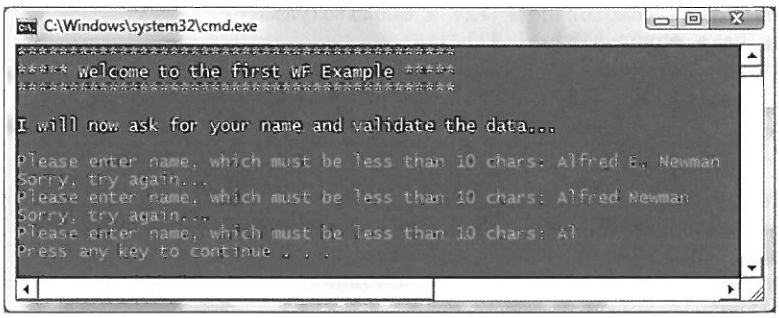
```
public sealed partial class ProcessUsernameWorkflow :
                            SequentialWorkflowActivity
{
  // A C# automatikus tulajdonság használata.
  public string UserName { get; set; }

  private void GetAndValidateUserName(object sender,
                                      ConditionalEventArgs e)
  {
    Console.Write("Please enter name, which must be less than
                  10 chars: ");
    UserName = Console.ReadLine();

    // Megnézzük, hogy a név hosszúsága megfelelő-e,
    // és beállítjuk az eredményt.
    e.Result = (UserName.Length >= 10);
  }
  ...
}
```

A `While` tevékenység elkészítésének utolsó lépéseként egy további tevékenységet adunk hozzá a `While` logika hatókörén belül. Az új `NameNotValidActivity` nevű `Code` tevékenységet rendeljük hozzá a `NameNotValid` metódushoz az `ExecuteCode` értéken keresztül. A 26.8. ábrán látható a végső munkafolyamat-diagram. A `NameNotValid()` megvalósítása szándékosan egyszerű:

```
private void NameNotValid(object sender, EventArgs e)
{
  Console.WriteLine("Sorry, try again...");
}
```



26.8. ábra: *A végső munkafolyamat-diagram*



26.9. ábra: *A munkafolyamat-alkalmazás működés közben*

Ekkor lefordíthatjuk és futtathatjuk a munkafolyamat-alkalmazást. Amikor futtatjuk a programot, néhányszor szándékosan adjunk meg tíznél több karaktert. Azt látjuk, hogy a futtatómotor arra kényszeríti a felhasználót, hogy újra adja meg az adatot egészen addig, amíg az üzleti szabály (kevesebb, mint tíz karakterből álló név) teljesül. A 26.9. ábrán láthatunk egy lehetséges kimenetet.

A WF-motor hosztolási kódja

Bár az első példánk az elvárásainknak megfelelően működik, meg kell vizsgálnunk azt a forráskódot, amely a WF-futtatómotort utasítja azoknak a feladatoknak a végrehajtására, amelyek az aktuális munkafolyamatot képviselik. Ehhez nyissuk meg a `Program.cs` fájlt, amely a kezdő projektünk definiálásakor jött létre. A `Main()` metódusban találjuk a kódot, amely két kulcsfontosságú típust használ, a `WorkflowRuntime` és a `WorkflowInstance` típusokat.

Ahogy a nevükből sejthetjük, a `WorkflowRuntime` típus képviseli magát a WF-futtatómotort, míg a `WorkflowInstance` adott munkafolyamat-példány egy példányának megjelenítésére szolgál. A kérdéses `Main()` metódus, megjegyzésekkel kiegészítve a következő:

```
static void Main(string[] args)
{
  // A futtatómotor leállításának biztosítása, ha elkészültünk.
  using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
  {
    AutoResetEvent waitHandle = new AutoResetEvent(false);

    // Események kezelése, amelyeket a rendszer észlel,
    // ha a futtatómotor befejezi a munkafolyamatot,
    // és ha a motor hibával áll le.
    workflowRuntime.WorkflowCompleted
        += delegate(object sender, WorkflowCompletedEventArgs e)
    {
      waitHandle.Set();
    };
    workflowRuntime.WorkflowTerminated
        += delegate(object sender, WorkflowTerminatedEventArgs e)
    {
      Console.WriteLine(e.Exception.Message);
      waitHandle.Set();
    };
```

```
        // Most létrehozzuk azt a WF-példányt,
        // amely a típusunkat képviseli.
        WorkflowInstance instance =
            workflowRuntime.CreateWorkflow(
                typeof(UserDataWFApp.ProcessUsernameWorkflow));
        instance.Start();

        waitHandle.WaitOne();
    }
}
```

A `WorkflowRuntime` típus `WorkflowCompleted` és `WorkflowTerminated` eseményeit a rendszer a C# névtelen metódusszintaxis segítségével kezeli. A `WorkflowCompleted` eseményt akkor hajtja végre, amikor a WF-motor befejezte a munkafolyamat-példány futtatását, míg a `WorkflowTerminated` eseményt akkor, ha a futtatómotor hibával fejezi be a működését.

Nem kötelező ezeknek az eseményeknek a kezelése, bár az IDE által generált kód ezt megteszi, ugyanis az `AutoResetEvent` típus segítségével így tájékoztatja a várakozó szálat arról, hogy ezek az események bekövetkeztek. Ez különösen fontos egy konzolalkalmazásnál, amikor a WF-motor egy második szálon fut. Ha a munkafolyamat-logika nem használ semmilyen várakozáskezelőt, a fő szál még azelőtt leállhat, hogy a WF-példány elvégezhetné a munkáját.

A következő érdekesség a `WorkflowInstance` típus létrehozása. A `WorkflowRuntime` típus a `CreateWorkflow()` metódussal rendelkezik, amely típusinformációkat vár a létrehozni kívánt munkafolyamatról. Egyszerűen meghívjuk a `Start()` metódust a visszakapott objektumreferenciából. Ez minden, amire szükségünk van a WF-futtatómotor elindításához és az egyedi munkafolyamat feldolgozásának megkezdéséhez.

Egyedi indítási paraméterek hozzáadása

Mielőtt folytatnák egy sokkal érdekesebb munkafolyamat-példával, vizsgáljuk meg, hogyan definiáljuk az alkalmazásszintű paramétereket. Ha megvizsgáljuk a `Code` tevékenységeink (különösen a `ShowInstructions()` és a `NameNotValid()`) által használt, tervező generálta metódusok szignatúráját, észrevehetjük, hogy ezeket a WF-eseményeken keresztül hívjuk, amelyek a `System.EventHandler` metódusreferenciát használják (`object` és `System.EventArgs` bejövő paramétertípussal).

Mivel ez a .NET-metódusreferencia megköveteli a regisztrált eseménykezelőt, és argumentumként használja a `System.Object` és a `System.EventArgs` elemeket, kérdés, hogy hogyan adhatunk át olyan *egyedi* paramétereket, amelyeket a `Code` tevékenység használhat. Tulajdonképpen arra vagyunk kíváncsiak, hogy hogyan definiálunk egyedi paramétereket, amelyeket bármilyen tevékenység használhat az aktuális munkafolyamat-példányban.

A WF-motor támogatja az egyedi paraméterek használatát a generikus `Dictionary<string, object>` típus segítségével. A `Dictionary` objektumhoz adott név/érték párokat társítani kell a munkafolyamat-példányunk (azonos névvel rendelkező) tulajdonságaival. Ha ezt megtettük, a paramétereket átadhatjuk a WF-futtatómotornak a munkafolyamat-példányunk elindításakor. Ennek a megközelítésnek a használatával beolvashatunk és beállíthatunk egyedi paramétereket egy meghatározott munkafolyamat-példány egészében.

Megjegyzés A `Dictionary` objektumban definiált neveket nyilvános tulajdonságokra kell képezni, *nem* nyilvános tagváltozókra. Ha ezt próbáljuk tenni, futásidejű kivételt generálunk.

A `Main()` kódjának módosításával definiáljunk egy `Dictionary<string, object>` elemet, amely két adatelemet tartalmaz. Az első elem sztring, amely a túl hosszú név esetében megjelenített hibaüzenetet mutatja, a második elem numerikus változó, amelyet a felhasználónév maximális hosszának meghatározására használunk. Regisztráljuk a paramétereket a WF-futtatómotorban, ehhez második paraméterként adjuk át a `Dictionary` objektumunkat a `CreateWorkflow()` metódusnak. A releváns módosítások a következők:

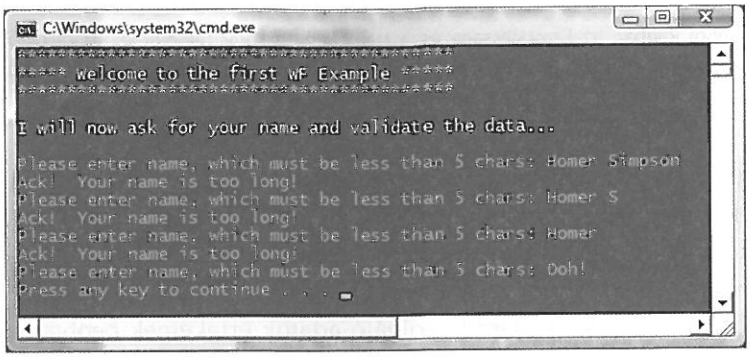
```
using (WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
  ...
  // Definiáljunk két paramétert, amelyet a munkafolyamat használ.
  // Ne feledjük, ezeket megegyező nevű tulajdonságokra
  // kell leképeznünk a munkafolyamat-osztálytípusunkban.
  Dictionary<string, object> parameters =
                            new Dictionary<string, object>();
  parameters.Add("ErrorMessage", "Ack! Your name is too long!");
  parameters.Add("NameLength", 5);

  // Most hozzunk létre egy WF-példányt, amely a típusunkat
  // képviseli, és adjuk át a paramétereket.
  WorkflowInstance instance =
     workflowRuntime.CreateWorkflow(
         typeof(UserDataWFApp.ProcessUsernameWorkflow), parameters);
  instance.Start();
  waitHandle.WaitOne();
}
```

Megjegyzés Az előbbi kódban az ErrorMessage és a NameLength könyvtár elemeihez tartozó értékek be vannak kódolva. Dinamikusabb szemléletet tükröz ezeknek az értékeknek a kapcsolódó `*.config` fájlból vagy az esetlegesen bejövő parancssori paraméterekből való beolvasása.

Ha megpróbáljuk futtatni a programunkat, futásidejű kivételt kapunk, hiszen a bejövő értékeket még nem kapcsoltuk nyilvános tulajdonságokhoz a munkafolyamat-típusunkban. A tulajdonságok összekapcsolása után a futtatórendszer a munkafolyamat létrehozásakor meghívja őket. Ezután ezeket a tulajdonságokat használhatjuk az alapul szolgáló adatok értékeinek beolvasására és beállítására. A releváns módosítások a `ProcessUsernameWorkflow` osztálytípushoz a következők:

```
public sealed partial class ProcessUsernameWorkflow :
                            SequentialWorkflowActivity
{
  ...
  // Ezek a tulajdonságok leképezhetők a Dictionary objektumon
  // belüli nevekre.
  public string ErrorMessage { get; set; }
  public int NameLength { get; set; }

  private void GetAndValidateUserName(object sender,
                                      ConditionalEventArgs e)
  {
    Console.Write("Please enter name, which much be less than {0}
                  chars: ", NameLength);
    UserName = Console.ReadLine();

    // Megnézzük, hogy a név megfelelő hosszúságú-e,
    // és beállítjuk az eredményt.
    e.Result = (UserName.Length >= NameLength);
  }

  private void NameNotValid(object sender, EventArgs e)
  {
    Console.WriteLine(ErrorMessage);
  }
  ...
}
```

Két új automatikus tulajdonsággal bővítettük a forráskódot, továbbá a `GetAndValidateUserName()` metódus ellenőrzi a `NameLength` tulajdonság segítségével megadott hosszúságot, valamint a hibaüzenet az `ErrorMessage` tulajdonságban talált értéket írja ki. Ezeket az értékeket mindkét esetben a munkafolyamat-példány létrehozásakor átadott `Dictionary` objektum határozza meg. A 26.10. ábrán láthatjuk a módosított példa lehetséges kimeneteit.



26.10. ábra: *A munkafolyamat működés közben, egyedi paraméterekkel*

Forráskód A UserDataWFApp kódfájlokat a forráskódkönyvtár 26. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

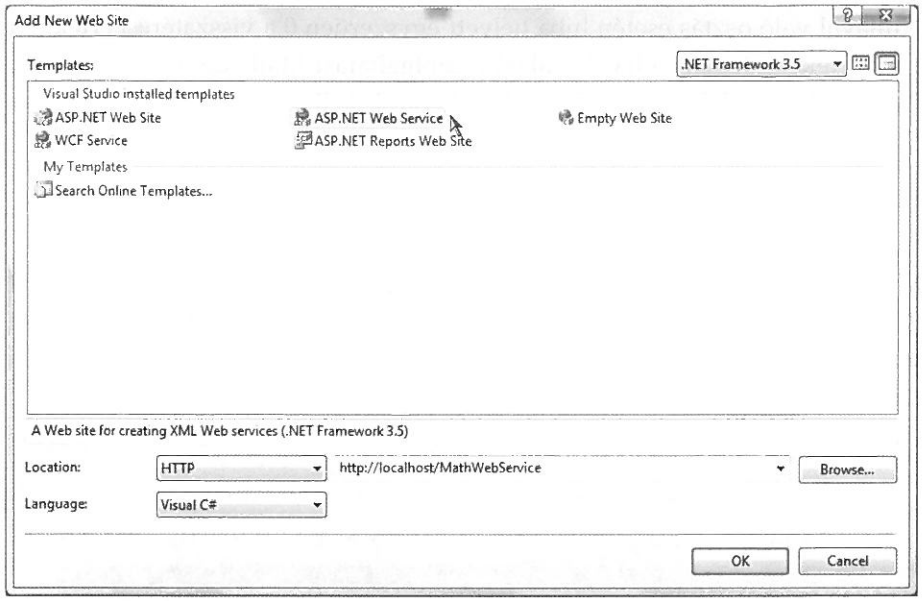
Webszolgáltatások hívása a munkafolyamatunkban

A WF számos olyan tevékenységet tartalmaz, amelyek biztosítják az XML-webszolgáltatásokkal való együttműködést a munkafolyamat-alkalmazásunk élettartama során. Ha egyszerűen csak egy létező webszolgáltatást szeretnénk meghívni, használhatjuk az `InvokeWebService` tevékenységet.

Megjegyzés Mivel a WCF a preferált API a szolgáltatások létrehozásánál, így nem részletezzük az XML-webszolgáltatások létrehozását (a 25. fejezet érinti a témát); ezért az általunk itt meghívott webszolgáltatások szándékosan egyszerűek.

A MathWeb szolgáltatás létrehozása

Az első feladat egy olyan XML-webszolgáltatás létrehozása, amelyet a munkafolyamat-alkalmazás felhasználhat. Ehhez hozzunk létre egy teljesen új XML-webszolgáltatás-projektet a File > New Web Site menüpont segítségével. Válasszuk az ASP.NET Web Service ikont, és állítsuk be a Locationt http-re, hogy a webszolgáltatást egy új IIS-beli virtuális könyvtárban hozzuk létre, amelyet a http://localhost/MathWebService címre képezhetünk le (lásd a 26.11. ábrát).



26.11. ábra: *XML-webszolgáltatás-projekt hozzáadása a WF-alkalmazáshoz*

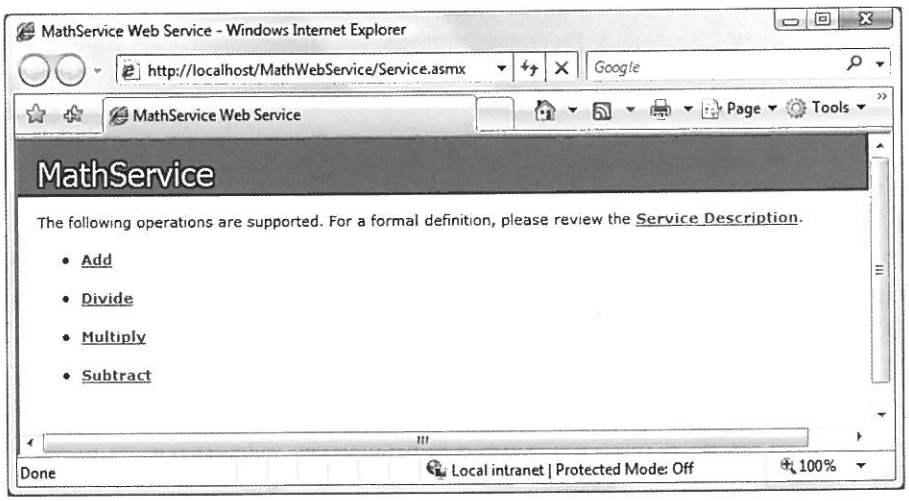
Az XML-webszolgáltatás lehetővé teszi külső hívók számára matematikai alapműveletek végrehajtását két egész számmal a következő `[WebMethod]` attribútummal ellátott nyilvános tagok segítségével:

```
[WebService(Namespace = "http://intertech.com/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class MathService : System.Web.Services.WebService
{
  [WebMethod]
  public int Add(int x, int y)
  { return x + y; }
  [WebMethod]
  public int Subtract(int x, int y)
  { return x - y; }
  [WebMethod]
  public int Multiply(int x, int y)
  { return x * y; }
  [WebMethod]
  public int Divide(int x, int y)
  {
    if (y == 0)
      return 0;
    else
      return x / y;
  }
}
```

Nullával való osztás esetén hiba helyett egyszerűen 0 a visszatérési érték, ha az y értéke valóban nulla. Továbbá a szolgáltatást MathService-re neveztük át, és ezért az alábbiak szerint módosítanunk kell a `Class` tulajdonságot az `*.asmx` fájlban:

```
<%@ WebService Language="C#" CodeBehind="~/App_Code/Service.cs"
    Class="MathService" %>
```

Most már tesztelhetjük az XML-webszolgáltatásunkat a projekt futtatásával (Ctrl + F5) vagy hibakereséssel (F5). Ennek során egy webalapú tesztelési front endet találunk, amely valamennyi webmetódus hívását lehetővé teszi (lásd a 26.12. ábrát).



26.12. ábra: *A MathWebService tesztelése*

Ezzel bezárhatjuk a webszolgáltatás-projektet.

Forráskód A MathWebService kódfájlokat a forráskódkönyvtár 26. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A WF-webszolgáltatás-fogyasztó létrehozása

Hozzuk létre a WFMathClient új Sequential Workflow konzolalkalmazás-projektet, és nevezzük át a kezdeti C#-fájlunkat `MathWF.cs`-re. Az alkalmazás a feldolgozandó adatokat a felhasználótól kéri, és megkérdezi, hogy milyen műveletet szeretne végrehajtani (összeadás, kivonás stb.). Először nyissuk meg a kódfájlunkat, és definiáljunk egy új `enum` típust `MathOperation` névvel:

```
public enum MathOperation
{
  Add, Subtract, Multiply, Divide
}
```

Ezután definiáljunk négy automatikus tulajdonságot az osztályunkban, kettőt, amelyek a feldolgozni kívánt numerikus adatokat képviselik, egyet, amely a művelet eredményét jeleníti meg, és egyet, amely magát a matematikai műveletet képviseli (a `MathWF` alapértelmezett konstruktora az `Operation` tulajdonságot `MathOperation.Add` értékre állítja):

```
public sealed partial class MathWF : SequentialWorkflowActivity
{
  // Tulajdonságok.
  public int FirstNumber { get; set; }
  public int SecondNumber { get; set; }
  public int Result { get; set; }
  public MathOperation Operation { get; set; }

  public MathWF()
  {
    InitializeComponent();

    // Állítsuk az alapértelmezett Operation értékét összeadásra.
    Operation = MathOperation.Add;
  }
  ...
}
```

A WF-tervezővel a Properties ablakban az `ExecuteCode` értékének beállításával adjunk hozzá egy új `GetNumericalInput` nevű `Code` tevékenységet, amely a `GetNumbInput()` nevű metódusra képzünk le. Ebben a metódusban a felhasználótól két numerikus érték meghatározását kérjük, amelyeket a `FirstNumber` és `SecondNumber` tulajdonságokhoz rendeljük:

```
public sealed partial class MathWF : SequentialWorkflowActivity
{
  ...
  private void GetNumbInput(object sender, EventArgs e)
  {
    // Az egyszerűség kedvéért nem foglalkozunk annak
    // ellenőrzésével, hogy a bejövő értékek valóban számok-e.
    Console.Write("Enter first number: ");
    FirstNumber = int.Parse(Console.ReadLine());

    Console.Write("Enter second number: ");
    SecondNumber = int.Parse(Console.ReadLine());
  }
}
```

Adjunk hozzá egy második Code tevékenységet GetMathOpInput névvel, amely a `GetOpInput()` metódusra képeződik le, amelyben megkérdezzük a felhasználótól, hogyan szeretné a numerikus adatokat feldolgozni. Feltételezzük, hogy a felhasználó az A, S, M vagy D betűket adja meg, és ennek a karakternek a függvényében beállítjuk a `MathOperation` tulajdonságot a felsorolt típus megfelelő értékre. Ennek egy lehetséges megvalósítása a következő:

```
private void GetOpInput(object sender, EventArgs e)
{
  Console.WriteLine("Do you wish to A[dd], S[ubtract],");
  Console.Write("Do you wish to M[ultiply] or D[ivide]: ");
  string option = Console.ReadLine();

  switch (option.ToUpper())
  {
    case "A":
      Operation = MathOperation.Add;
      break;
    case "S":
      Operation = MathOperation.Subtract;
      break;
    case "M":
      Operation = MathOperation.Multiply;
      break;
    case "D":
      Operation = MathOperation.Divide;
      break;
    default:
      numericalOp = MathOperation.Add;
      break;
  }
}
```

Ekkor már rendelkezünk a szükséges adatokkal. Próbáljuk ki, hogyan adhatjuk át feldolgozásra az adatokat az XML-webszolgáltatásunknak.

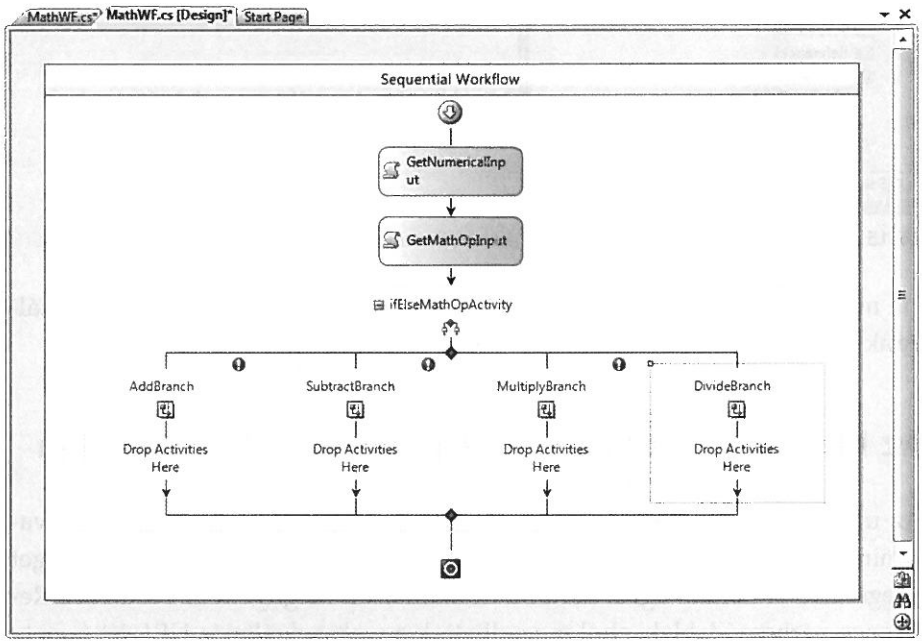
Az IfElse tevékenység konfigurálása

Mivel a numerikus adatainkat a rendszer négy egyedi módon dolgozhatja fel, egy IfElse tevékenységgel határozzuk meg, hogy a szolgáltatás melyik webmetódusát hívjuk meg. Ha IfElse tevékenységet húzunk a tervezőnkbe, automatikusan két elágazást kapunk. Az IfElse tevékenység további elágazásainak hozzáadásához kattintsunk jobb gombbal az IfElse tevékenység ikonjára, és válasszuk az AddBranch menüpontot. A 26.13. ábrán láthatjuk az aktuális WF-tervezőt (az egyes elágazások és a teljes IfElse tevékenység megfelelő nevet kapott).

Az IfElse tevékenység minden elágazása tartalmazhat tetszőleges számú belső tevékenységet, amelyek meghatározzák mi történik, ha a döntési logika az alkalmazás működését ezen az útvonalon viszi végig. Mielőtt hozzáadnánk ezeket az altevékenységeket, először hozzá kell adnunk a logikát, amely lehetővé teszi a WF-motor számára, hogy eldöntse, melyik ágon haladjon tovább; ennek megvalósulásához állítsuk be minden egyes IfElseBranch tevékenységhez a `Condition` értéket.

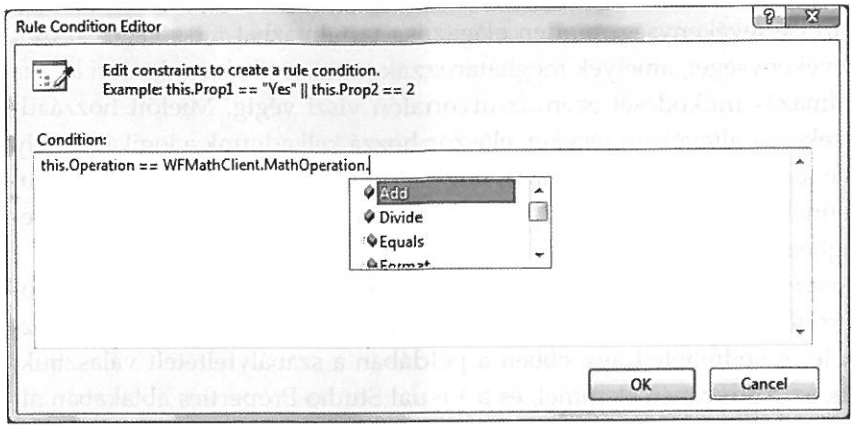
A `Condition` eseményt kódfeltétel vagy deklaratív szabályfeltétel létrehozására konfigurálhatjuk. A fejezet első példaprojektje bemutatta, hogyan hozhatunk létre kódfeltételt, így ebben a példában a szabályfeltételt választjuk. Kezdjük az AddBranch elemmel, és a Visual Studio Properties ablakában állítsuk a `Condition` tulajdonságot Declarative Rule Condition értékre. Ezután kattintsunk a ConditionName alcsomópont gombjára, és a megjelenő párbeszédablakban kattintsunk a New gombra. Ekkor létrehozhatjuk azt a kifejezést, amely meghatározza az aktuális ág igaz vagy hamis értékét. Az első ág esetén a kifejezés a következő:

```
this.Operation == MathOperation.Add
```



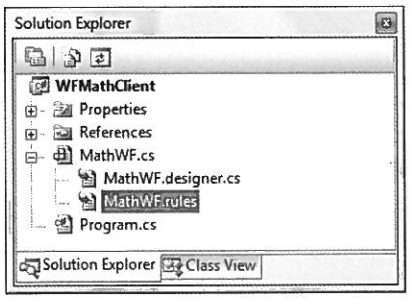
26.13. ábra: *Egy többelágazásos IfElse tevékenység*

Ez a párbeszédablak támogatja az IntelliSense-t, amely mindig szívesen látott szolgáltatás (lásd a 26.14. ábrát).



26.14. ábra: *Deklaratív szabályfeltétel definiálása*

Az összes szabály meghatározása után azt látjuk, hogy a projektünkben egy új fájl jött létre `*.rules` kiterjesztéssel (lásd a 26.15. ábrát).

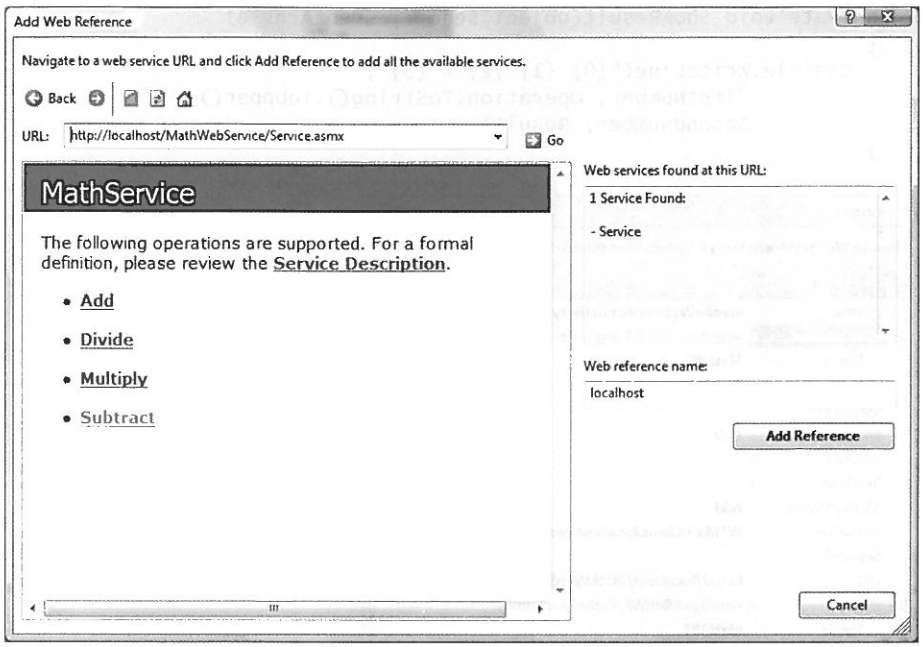


26.15. ábra: *A *.rules fájl tartalmaz minden deklaratív szabályt, amelyet a WF számára létrehoztunk*

Ha megnyitnánk ezt a fájlt, rengeteg `<RuleExpressionCondition>` elemet találnánk benne, amelyek az általunk létrehozott feltételeket írják le.

Az InvokeWebService tevékenységek konfigurálása

Az utolsó feladat a bejövő adatok átadása a megfelelő webmetódusnak, valamint az eredmény kiírása. Húzzunk egy InvokeWebService tevékenységet a legszélső bal oldali ágba. Ekkor automatikusan megnyílik az Add Web Reference párbeszédablak, ahol megadhatjuk a webszolgáltatás URL-jét (amely ebben a példában a http://localhost/MathWebService/Service.asmx cím), majd kattintsunk az Add Reference gombra (lásd a 26.16. ábrát).



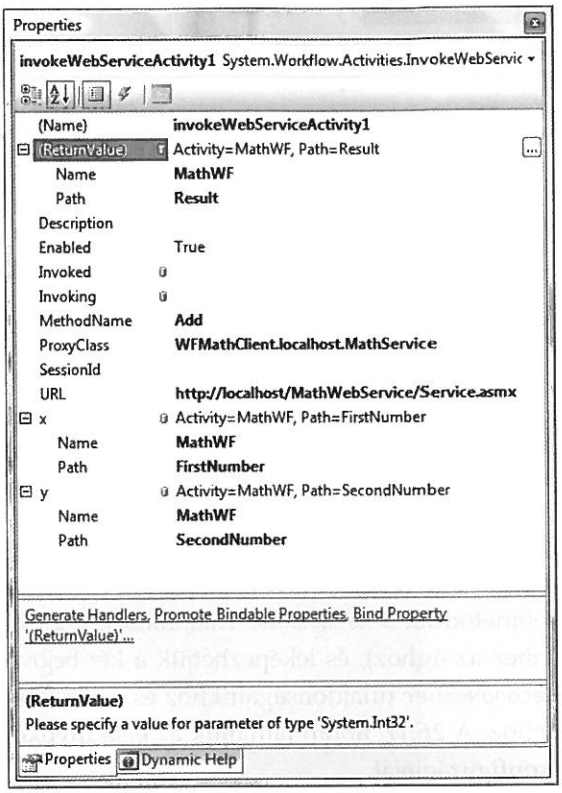
26.16. ábra: *Az XML-webszolgáltatásunk hivatkozása*

Ekkor az IDE a webszolgáltatásunkhoz létrehoz egy proxyt, és az InvokeWebService `ProxyClass` tulajdonságát erre állítja. Ekkor a Properties ablakban megadhatjuk a meghívandó webmetódust a `MethodName` tulajdonság segítségével (amely az `Add` metódus ehhez azághoz), és leképezhetjük a két bejövő paramétert a `FirstNumber` és a `SecondNumber` tulajdonságainkhoz és a visszatérési értéket a `Result` tulajdonsághoz. A 26.17. ábrán láthatjuk az első InvokeWebService tevékenység teljes konfigurációját.

A fennmaradó webmetódusok tevékenységhez rendelésével ismételhetjük meg a folyamatot a maradék három IfElseághoz. Bár az Add Web Reference párbeszédablak minden egyes InvokeWebService tevékenységhez megjelenik, az IDE elég intelligens ahhoz, hogy újra felhasználja a már meglévő proxyt, ugyanis minden tevékenység ugyanazzal a végponttal kommunikál.

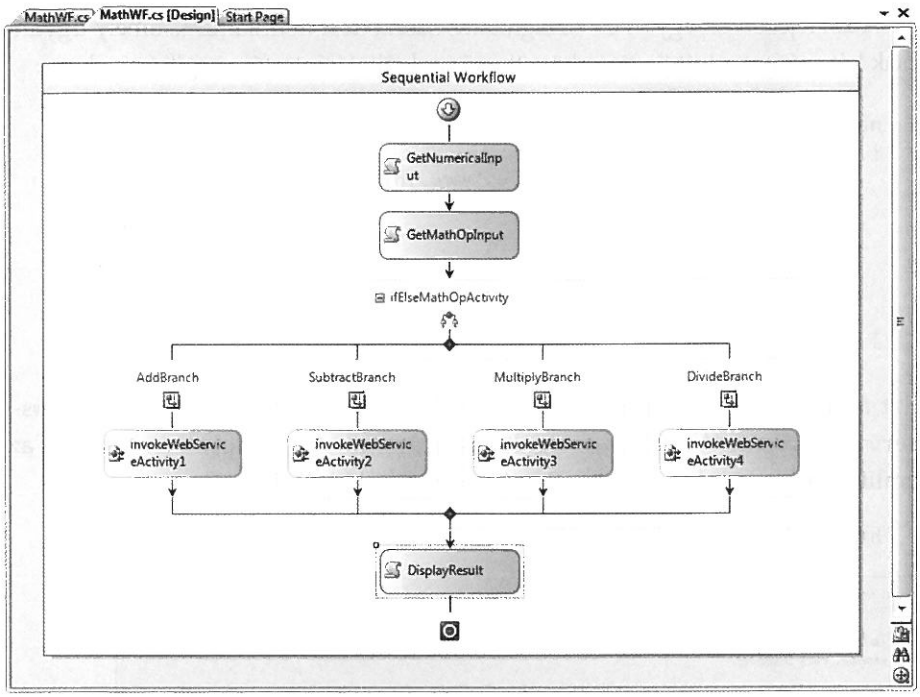
Végül, de nem utolsósorban, egy utolsó Code tevékenységet adunk az `IfElse` kódhoz, amely megjeleníti a felhasználó által választott művelet eredményét. A tevékenységnek adjuk a `DisplayResult` nevet, és állítsuk az alábbiak szerint megvalósított `ShowResult()` metódust `ExecuteCode` értékre:

```
private void ShowResult(object sender, EventArgs e)
{
  Console.WriteLine("{0} {1} {2} = {3}",
          FirstNumber, Operation.ToString().ToUpper(),
          SecondNumber, Result);
}
```

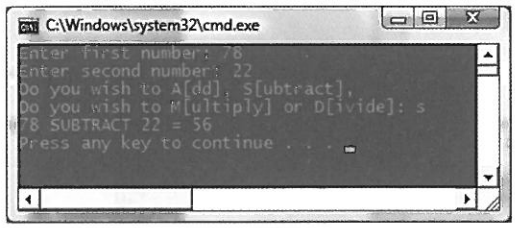


26.17. ábra: *Teljesen konfigurált InvokeWebService tevékenység*

Az egyszerűség kedvéért az `Operation` tulajdonság szöveges értékét használjuk a kiválasztott matematikai operátor ábrázolására, ahelyett, hogy további kóddal képeznénk le a `MathOperation.Add` elemet a + jelhez, a `MathOperation.Subtract` elemet a - jelhez és így tovább. A 26.18. ábrán láthatjuk a munkafolyamat végleges tervét; a 26.19 ábra pedig egy lehetséges kimenetet mutat.



26.18. ábra: *A kész webszolgáltatás-centrikus munkafolyamat*



26.19. ábra: *Kommunikáció XML-webszolgáltatással a WF-alkalmazásból*

Kommunikáció WCF-szolgáltatással a SendActivity segítségével

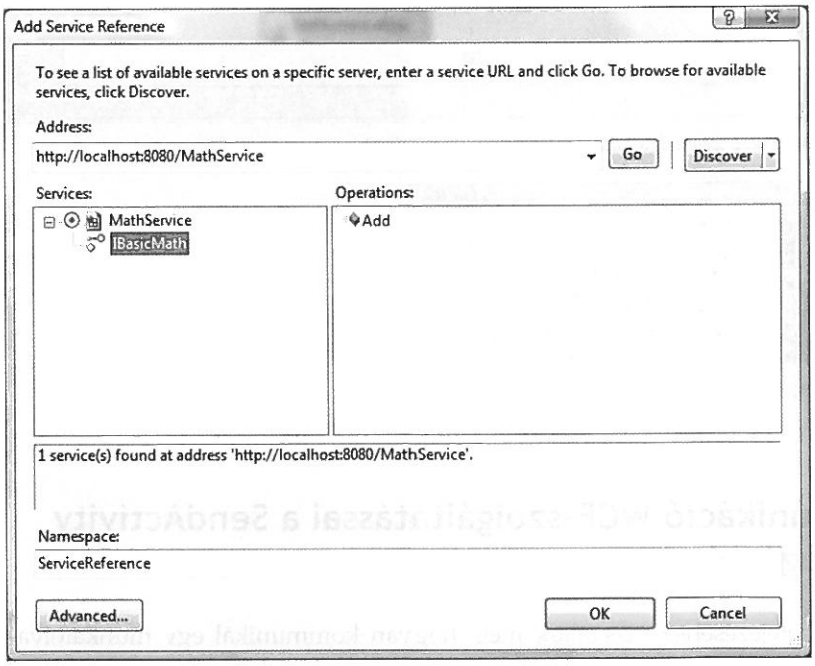
A példa befejezéséhez vizsgáljuk meg, hogyan kommunikál egy munkafolyamat-alkalmazás a WCF-szolgáltatással. A .NET 3.5-ös központú `SendActivity` és `ReceiveActivity` típusok ugyanis lehetővé teszik olyan munkafolyamat-alkalmazások létrehozását, amelyek WCF-szolgáltatásokkal kommunikálnak. Ahogyan a neve sugallja, a `SendActivity` típust WCF-szolgáltatás műveleteinek hívására használhatjuk, míg a `ReceiveActivity` a WCF-szolgáltatás számára a WF visszahívását teszi lehetővé (egy duplex hívási szerződés esetén).

A 25. fejezetben egy WCF-szolgáltatás-szerződést definiáltunk, amely ugyancsak két számot adott össze, a következő szolgáltatásinterfész segítségével:

```
namespace MathServiceLibrary
{
  [ServiceContract(Namespace = "www.intertech.com")]
  public interface IBasicMath
  {
    [OperationContract]
    int Add(int x, int y);
  }
}
```

Ezt az interfészt a `MathService` típussal valósítottuk meg, és a `MathWindowsServiceHost.exe` Windows-szolgáltatás hosztolta. A Windows-szolgáltatás az említett funkcionalitást a következő végpontból biztosítja:

```
http://localhost:8080/MathService
```

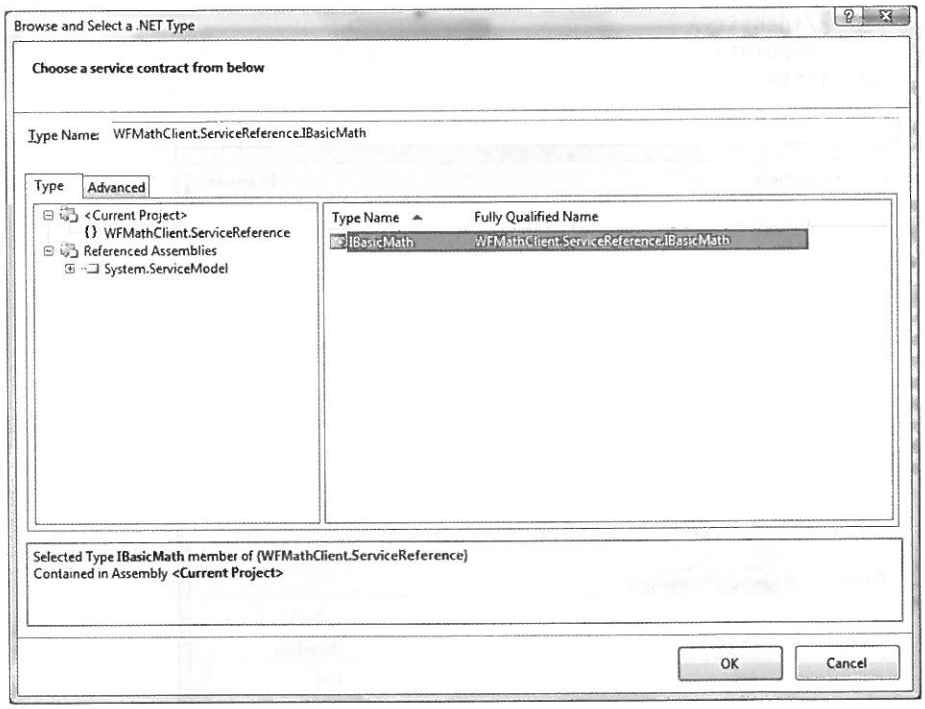


26.20. ábra: *A WCF MathService referenciájának létrehozása*

Ha létrehoztuk és telepítettük ezt a szolgáltatást (részleteket lásd a 25. fejezetben), módosíthatjuk az aktuális WFMathClient projektünket úgy, hogy a `SendActivity` típus segítségével kommunikálhasson a szolgáltatással. Az első lépés egy referencia hozzáadása a szolgáltatáshoz az Add Service Reference

párbeszédablak segítségével (lásd a 26.20 ábrát). Ezzel a lépéssel létrehozunk egy ügyféloldali proxyt, és módosítjuk az `App.config` fájlunkat a WCF-specifikus beállításokkal.

Húzzunk egy `SendActivity` típust a WF-tervezőnk felületére (a neve `WCFSendAddActivity`), közvetlenül az utolsó Code tevékenység után. A Properties ablakban kattintsunk a `ServiceOperationInfo` tulajdonság gombjára, és a megjelenő párbeszédablakban kattintsunk az Import elemre. Ekkor egy második párbeszédablak jelenik meg, ahol összekapcsolhatjuk a `SendActivity` típust egy metaadat-leírással az adott WCF-szolgáltatás-szerződéshez. Válasszuk ennél a végpontnál az egyetlen lehetséges szerződést: az `IBasicMathot` (lásd a 26.21. ábrát).



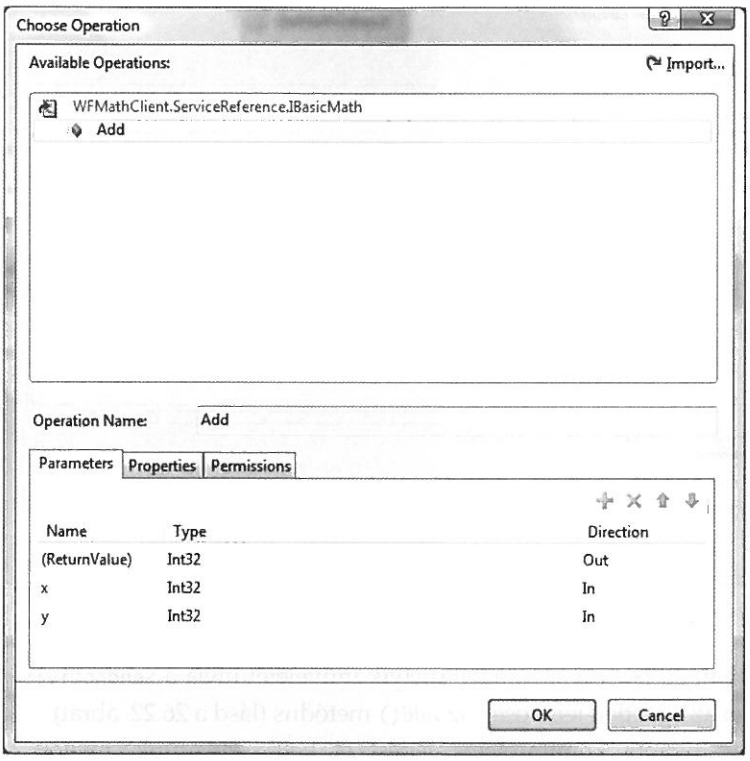
26.21. ábra: *Szolgáltatásmetódus kapcsolása a SendActivity típushoz*

Ekkor kiválaszthatjuk, hogy a szerződés melyik műveletét hívja a `SendActivity`. A mi esetünkben az egyetlen lehetőség az `Add()` metódus (lásd a 26.22. ábrát).

Még néhány további konfigurációs lépést el kell végeznünk, mielőtt a `SendActivity` készen állna a `FirstNumber` és a `SecondNumber` tulajdonságok értékeinek átadására, hogy a `MathService` feldolgozhassa őket. Különösen fontos a `ChannelToken` tulajdonság értékének beállításával tájékoztatni a `SendActivity` te-

vékenységet, hogy melyik kötést használjuk a hívás alatt (egyetlen WCF-szolgáltatást beállíthatunk úgy, hogy különböző kötésekhez álljon rendelkezésre). Ha megnyitjuk a módosított `App.config` fájlt, és megkeressük a `<client>` részt, azt találjuk, hogy a létrehozott kötés neve WSHttpBinding_IBasic_Math:

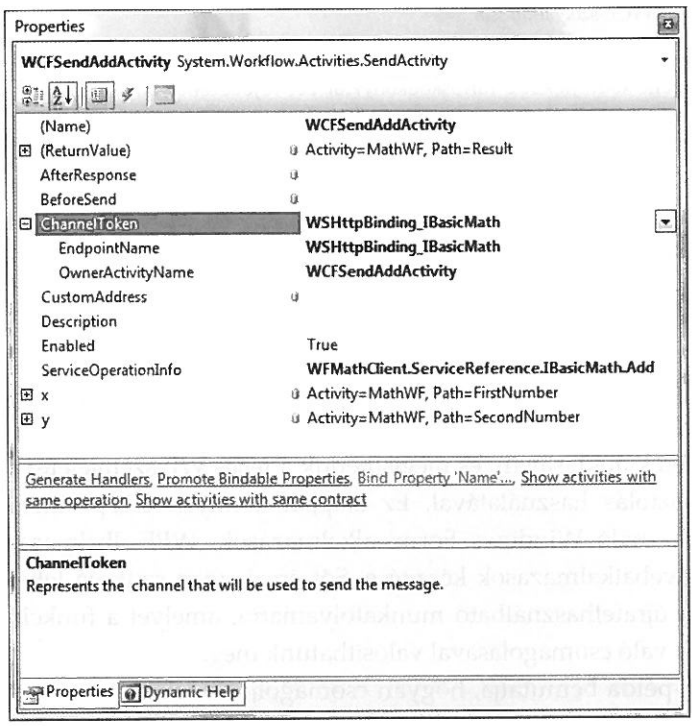
```
<client>
  <endpoint address="http://localhost:8080/MathServiceLibrary"
            binding="wsHttpBinding"
            bindingConfiguration="WSHttpBinding_IBasicMath"
            contract="WFMathClient.ServiceReference.IBasicMath"
            name="WSHttpBinding_IBasicMath"
  >
    <identity>
      <servicePrincipalName value=
                          "host/InterUber.intertech-inc.com" />
    </identity>
  </endpoint>
</client>
```



26.22. ábra: *Az Add() művelet kiválasztása*

Másoljuk ezt az értéket a vágólapra, és illesszük be a `ChannelToken` tulajdonságba az IDE Properties ablakának segítségével. A `ChannelToken` tulajdonság két alcsomópontot tartalmaz: az `EndpointName` és az `OwnerActivityName` csomópontokat. Mivel a `MathService` csak egyetlen végpontot tár fel, másoljuk át ugyanezt az értéksort a `ChannelTokenbe` (`WSHttpBinding_IBasicMath`) az `EndpointName`-hez, és jelöljük ki a munkafolyamat-példányunk nevét (`WCFSendAddActivity`) tulajdonosként.

Végül, de nem utolsósorban, az `Add()` metódus `x` és `y` paramétereit kapcsoljuk a `FirstNumber` és `Secondnumber` tulajdonságainkhoz, és a visszatérési értéket a `Result` tulajdonságunkhoz. Az eljárás megegyezik az InvokeWebService tevékenység konfigurálásával (a tulajdonságnév megadásához kattintsunk a kerek gombokra). A 26.23. ábrán látható a teljesen konfigurált SendActivity tevékenység.

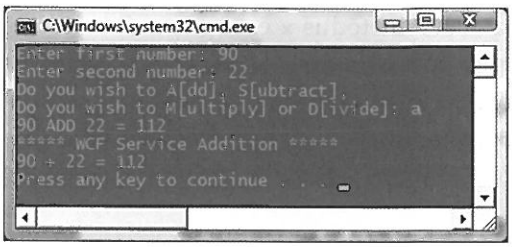


26.23. ábra: *A teljesen konfigurált SendActivity*

Az eredmény megtekintéséhez helyezzünk el egy utolsó Code tevékenységet a munkafolyamat-tervezőnkön, és rendeljük az `ExecuteCode` értéket a `WCFResult()` metódushoz, amelyet a következőképpen valósítunk meg:

```
private void WCFResult(object sender, EventArgs e)
{
  Console.WriteLine("***** WCF Service Addition *****");
  Console.WriteLine("{0} + {1} = {2}",
                    FirstNumber, SecondNumber, Result);
}
```

A 26.24. ábrán láthatjuk a végső kimenetet.



26.24. ábra: *Kommunikáció WCF-szolgáltatással*

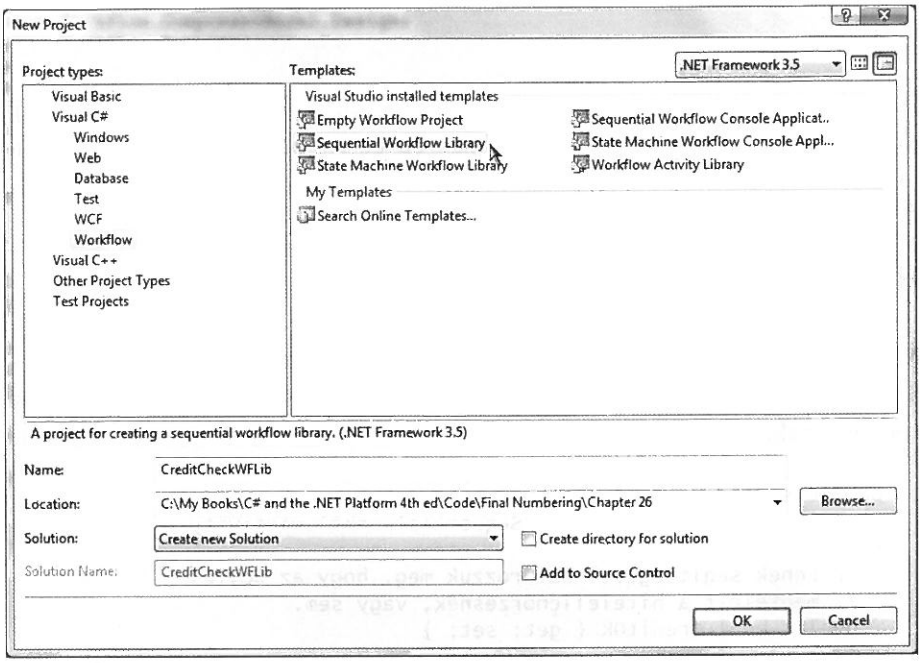
Forráskód A WFMathClient kódfájlokat a forráskódkönyvtár 26. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Újrafelhasználható WF-kódkönytár létrehozása

Az első példák lehetővé teszik, hogy körbejárjuk a különböző WF-tevékenységeket a tervezés során, kapcsolatba lépjünk a munkafolyamat futtatómotorjával (egyedi paraméterek átadásával), és megismerjük a teljes WF-szemléletet a konzolalapú WF-hosztolás használatával. Ez alapján könnyen elképzelhető munkafolyamatot használó Windows Forms-alkalmazások, WPF-alkalmazások vagy ASP.NET-webalkalmazások készítése. Sőt érezhetően szükség lehet alkalmazások között újrafelhasználható munkafolyamatra, amelyet a funkció .NET-kódkönyvtárba való csomagolásával valósíthatunk meg.

A következő WF-példa bemutatja, hogyan csomagoljunk munkafolyamatokat `*.dll` szerelvényekbe, és hogyan használjuk azokat hosztoló Windows Forms-alkalmazásból (amely mellesleg ugyanaz a folyamat, mint amelyik külső munkafolyamatot hosztolhat bármely végrehajtható fájlban, pl. egy WPF-alkalmazásban). Olyan munkafolyamatot tervezünk, amely egy személyautó megvásárlásához szükséges rendelés hitelellenőrzésének alapfolyamatát modellezi a 22. fejezetben létrehozott AutoLot adatbázisból.

Kezdjük a CreditCheckWFLib nevű Sequential Workflow Library projekt kiválasztásával (lásd a 26.25. ábrát), és nevezzük át a kezdeti fájlunkat `CreditCheckWF.cs`-re.



26.25. ábra: *Sequential Workflow Library projekt létrehozása*

Ekkor egy kezdő munkafolyamat-tervezőt kapunk. Egyetlen munkafolyamat-kódkönyvtár több olyan munkafolyamatot tartalmazhat, amelyek mindegyike beilleszthető a Project > Add New Item párbeszédablakkal. Adjunk hozzá egy referenciát a 22. fejezetben létrehozott `AutoLotDAL.dll` szerelvényünkre, és egészítsük ki a kezdeti kódfájlunkat az `AutoLotConnectedLayer` névtér importálásával:

```
// Adjuk hozzá az alábbi importutasítást.
using AutoLotConnectedLayer;

namespace CreditCheckWFLib
{
  ...
}
```

Ezután adjunk hozzá egy automatikus tulajdonságot, amely az ügyfél azonosítóját képviseli:

```
public sealed partial class CreditCheckWF :
                            SequentialWorkflowActivity
{
  // Az ügyfélazonosító a hitelellenőrzéshez.
  public int ID { get; set; }
  ...
}
```

Amikor egy későbbi lépésben létrehozzuk a kliensalkalmazást, ezt a tulajdonságot egy bejövő `Dictionary<string, object>` objektum segítségével állítjuk be, amelyet a munkafolyamat futtatómotorjának adunk át.

Hitelellenőrzés végrehajtása

Módosítsuk az osztályunkat egy további automatikus tulajdonsággal (`CreditOk`), amely azt jelenti, hogy az ügyfél megfelelt a „szigorú" hitelellenőrzési folyamatnak:

```
public sealed partial class CreditCheckWF :
                            SequentialWorkflowActivity
{
  // Ennek segítségével határozzuk meg, hogy az ügyfél
  // megfelelt a hitelellenőrzésnek, vagy sem.
  public bool CreditOK { get; set; }
  ...
}
```

Helyezzünk egy ValidateCreditActivity nevű Code tevékenységet a WF-tervezőnkre, és állítsuk az `ExecuteCode` értéket az új `ValidateCredit` metódusra. Nyilvánvaló, hogy egy termékszintű hitelellenőrzés rengeteg altevékenységeket, adatbázis-keresést és egyebeket foglalna magában. Ebben a példában egy véletlen számot fogunk generálni annak a lehetőségnek a megjelenítésére, hogy a hívó megfelel-e a hitelellenőrzésünknek:

```
private void ValidateCredit(object sender, EventArgs e)
{
  // Tegyünk úgy, mintha végrehajtottunk volna
  // néhány egzotikus hitelellenőrzést...
  Random r = new Random();
  int value = r.Next(500);
  if (value > 300)
    CreditOK = true;
  else
    CreditOK = false;
}
```

Ezután adjunk hozzá egy CreditCheckPassedActivity nevű IfElse tevékenységet két elágazással, amelyek a CreditCheckOk és a CreditCheckFailed névvel rendelkeznek. Állítsuk be a bal oldali elágazást úgy, hogy a rendszer új deklaratív szabályfeltételének a segítségével az alábbi kifejezés alkalmazásával értékelje ki:

```
this.CreditOK == true
```

Ha a felhasználó *nem felel meg* a hitelellenőrzésnek, akkor el kell távolítani a Customers táblából és hozzáadni a CreditRisks táblához. Mivel a 22. fejezet már gondoskodott erről a lehetőségről, az `InventoryDAL` típus `ProcessCreditRisk()` metódusának segítségével, adjunk új `CodeActivity` típust a `ProcessCreditRiskActivity` nevű `CreditCheckFailed` ágban. A típus a `ProcessCreditRisk()` metódusra képezhető le. Hozzuk létre a metódust így:

Megjegyzés Logikai paramétert adtunk az `InventoryDAL.ProcessCreditRisk()` metódushoz, hogy meghiúsítsuk a tranzakciót a tesztelés érdekében. Mindenképpen `false` értéket adjunk át első paraméterként.

```
private void ProcessCreditRisk(object sender, EventArgs e)
{
  // Ideális esetben a kapcsolatsztringet külsőleg tárolnánk.
  InventoryDAL dal = new InventoryDAL();
  dal.OpenConnection(@"Data Source=
                    (local)\SQLEXPRESS;Integrated Security=SSPI;" +
                    "Initial Catalog=AutoLot");
  try
  {
    dal.ProcessCreditRisk(false, ID);
  }
  finally
  {
    dal.CloseConnection();
  }
}
```

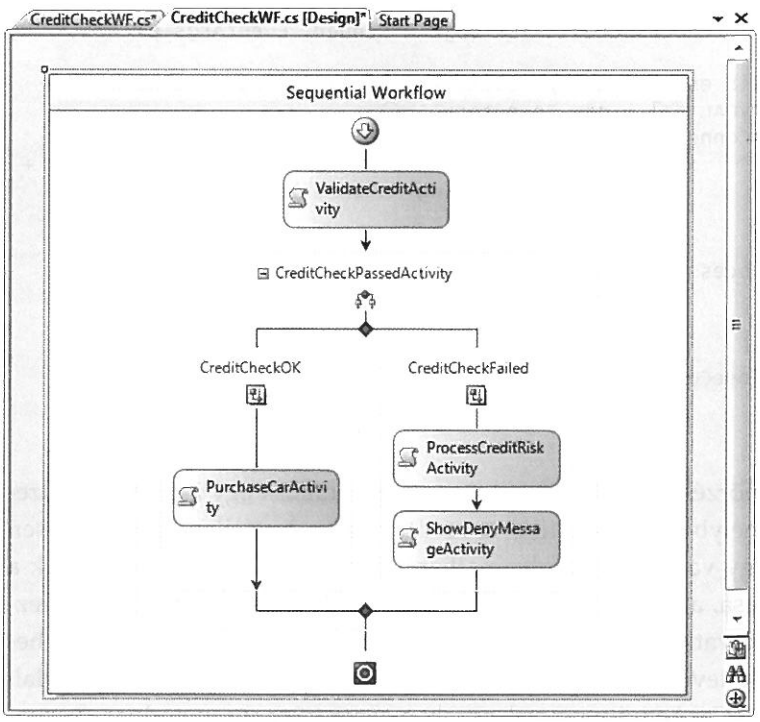
Ha a hitelellenőrzés sikerült, egyszerűen megjelenítünk egy információs üzenetdobozt, amelyben tájékoztatjuk a hívót, hogy a hitelellenőrzés sikeresen megtörtént. Egy valós munkafolyamatban a következő lépések lehetnének a rendelés leadása, a rendelésellenőrző e-mail küldése és így tovább. Tételezzük fel, hogy hivatkoztuk a `System.Windows.Forms.dll` szerelvényre, ezért helyezzünk Code tevékenységet az IfElse tevékenységünk legszélső, bal oldali ágába PurchaseCarActivity névvel, amely a `PurchaseCar()` metódusra képezhető le, és megvalósítása a következő:

```
private void PurchaseCar(object sender, EventArgs e)
{
  // Az egyszerűségre törekszünk. Így könnyen módosíthatjuk
  // az AutoLotDAL.dll szerelvényt új metódussal, hogy új rendelést
  // helyezzünk az Orders táblába.
  System.Windows.Forms.MessageBox.Show("Your credit has been
                                       approved!");
}
```

A munkafolyamat befejezéséhez adjunk egy utolsó CodeActivity tevékenységet a legszélső, jobb oldaliághoz közvetlenül a ProcessCreditRiskActivity után. Legyen az új tevékenység neve ShowDenyMessageActivity, amely a következő metódusra képezhető le:

```
private void CreditDenied(object sender, EventArgs e)
{
  System.Windows.Forms.MessageBox.Show("You are a CREDIT RISK!",
                                       "Order Denied!");
}
```

A munkafolyamat megjelenése most nagyjából a 26.26. ábrához hasonlít.

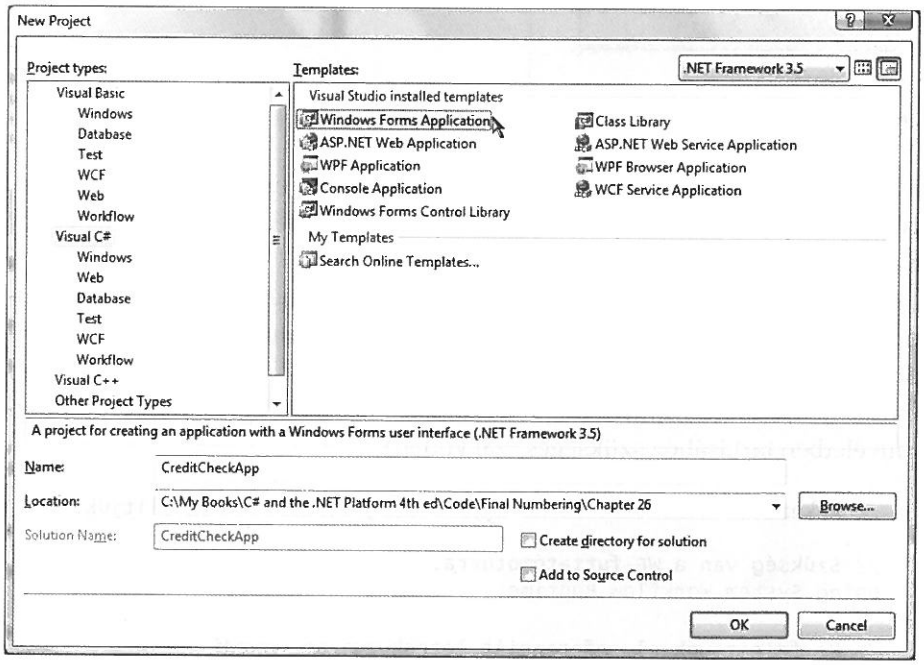


26.26. ábra: *A kész Sequential Workflow Library projekt*

Forráskód A CreditCheckWFLib kódfájlokat a forráskódkönyvtár 26. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Windows Forms-kliensalkalmazás létrehozása

Miután létrehoztunk egy újrafelhasználható .NET-kódkönyvtárat, amely egyedi munkafolyamatot tartalmaz, most már bármilyen .NET-alkalmazást létrehozhatunk, amely használhatja ezt a munkafolyamatot. Jóllehet a grafikus felületek készítésének a részleteit még nem ismerjük, a Windows Forms API használatával egy kezdetleges felhasználói felületet hozunk létre a munkafolyamat logikájának tesztelésére (a GUI-alapú .NET-alkalmazások vizsgálatát lásd a 27. fejezetben). Kezdjük egy új CreditCheckApp nevű Windows Forms-projekt létrehozásával (lásd a 26.27. ábrát).

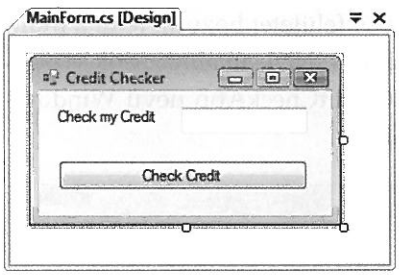


26.27. ábra: *Windows Forms-alkalmazásprojekt létrehozása a munkafolyamat-könyvtárunk tesztelésére*

Ezután nevezzük át a kezdeti `Form1.cs` fájlt a találó `MainForm.cs` névre, ezért jobb gombbal kattintsunk a Form1.cs ikonra a Solution Explorerben, és válasszuk a `Rename` opciót. Ezután adjunk referenciát az alábbi .NET-szerelvények mindegyikéhez:

- `CreditCheckWFLib.dll`
- `System.Workflow.Runtime.dll`
- `System.Workflow.Activities.dll`
- `System.Workflow.ComponentModel.dll`

Az alkalmazásunk felhasználói interfésze az kezdő űrlapon a következőkből áll: egy leíró `Label`, egy `TextBox` (`txtCustomerID` néven) és egy egyszerű `Button` típus (`btnExecuteWorkflow` néven). A 26.28 ábra egy lehetséges tervet ábrázol.



26.28. ábra: *Egyszerű felhasználói felület a munkafolyamat-könyvtárunk teszteléséhez*

Miután elhelyeztük ezeket a felhasználóifelület-elemeket a tervezőnkön, kezeljük a *Button* típus *Click* eseményét. A tervezői felületen kattintsunk kétszer a gomb ikonjára.

A kódfájlunkban valósítsuk meg a kattintásiesemény-kezelőt, hogy elindítsuk a WF-futtatómotort, és létrehozzuk az egyedi munkafolyamat egy példányát. A következő kód megegyezik a konzolalapú munkafolyamat-alkalmazás kódjával (leszámítva a munkafolyamat befejezéséig a parancssori program életben tartásához szükséges szál kódját):

```
// A kezdeti utasításokat az egyszerűség kedvéért eltávolítjuk.
...
// Szükség van a WF-futtatómotorra.
using System.Workflow.Runtime;

// Ne felejtsünk el referenciát létrehozni az egyedi
// WF-könyvtárunkhoz.
using CreditCheckWFLib;

namespace WinFormsWFClient
{
  public partial class MainForm : Form
  {
```

```
    public MainForm()
    {
      InitializeComponent();
    }

    private void btnCheckCustomerCredit_Click(object sender,
                                              EventArgs e)
    {
      // Hozzuk létre a WF-futtatómotort.
      WorkflowRuntime wfRuntime = new WorkflowRuntime();

      // Kérjük be az azonosítót a TextBox-ba, hogy az értéket
      // a munkafolyamatnak továbbíthassuk.
      Dictionary<string, object> args = new
                          Dictionary<string, object>();
      args.Add("ID", int.Parse(txtCustomerID.Text));

      // Megkapjuk a WF egy példányát.
      WorkflowInstance myWorkflow =
              wfRuntime.CreateWorkflow(typeof(CreditCheckWF), args);

      // Indítsuk el.
      myWorkflow.Start();
    }
  }
}
```

Az alkalmazás futtatásakor adjunk meg egy ügyfélazonosító értéket, és győződjünk meg róla, hogy a megadott ügyfélazonosító nem rendelkezik referenciával az Orders táblában (ezzel gondoskodunk arról, hogy az elemet sikeresen töröljük a Customers táblából).

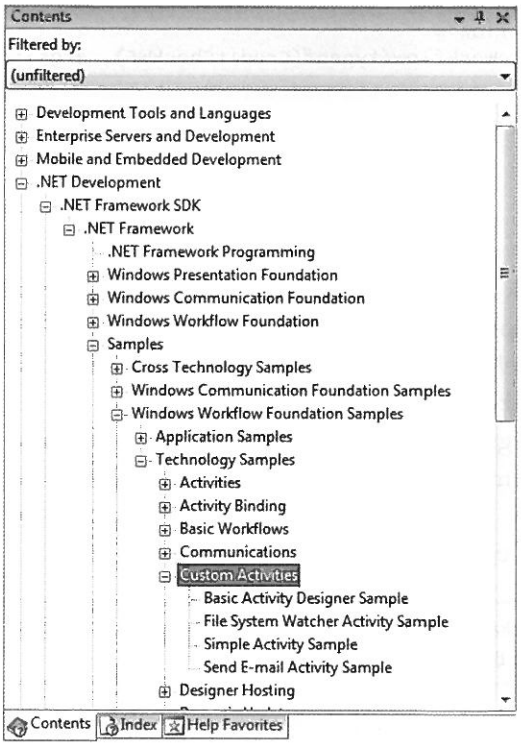
A hitelképesség tesztelésekor végül is azzal szembesülünk, hogy a kockázatos ügyfelet a rendszer törli a Customers táblából, és beírja a CreditRisk táblába. Tesztelési céllal hozzáadhatunk egy mesterséges bejegyzést a Customers táblában, és megpróbálhatjuk ellenőrizni egy rögzített vásárló hitelképességét is.

Forráskód A WinFormsWFClient kódfájlokat a forráskódkönyvtár 26. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Az egyedi tevékenységek

Megvizsgáltuk, hogy hogyan kell konfigurálni általános WF-tevékenységeket különböző típusú projektekben. Bár ezek a beépített tevékenységek sok WF-alkalmazás számára jelentenek biztos kezdőpontot, nem gondoskodnak minden lehetséges körülményről. Szerencsére a WF-közösség új egyedi tevékenységeket hoz létre, sok közülük ingyen letölthető, más tevényekségekhez pedig harmadik félen keresztül férhetünk hozzá különböző árakon.

Megjegyzés Ha szeretnénk további munkafolyamat-tevékenységeket vizsgálni, jó kiindulási pont a http://wf.netfx3.com címen található webhely. Itt szép számmal tölthetünk le további tevékenységeket, amelyek bővítik a termékkel szállított választékot.



26.29. ábra: *A .NET Framework 3.5 SDK dokumentáció több munkafolyamat-példát biztosít*

Az online WF-közösségektől beszerezhető kiegészítő tevékenységek nagy száma ellenére lehetséges (és néha szükséges is) egyedi tevékenységeket létrehozni. A Visual Studio 2008 rendelkezik egy Workflow Activity Library projektsablonnal pontosan erre a célra. Ha ezt a projekttípust választjuk, egy ter-

vezői felületet kapunk, amellyel létrehozhatjuk egyedi tevékenységünket hasonló megközelítéssel, mint amellyel magát a munkafolyamatot hozzuk létre (új tevékenységek hozzáadása, a tevékenységek hozzákapcsolása kódhoz stb.).

Hasonlóan egy egyedi Windows Forms-vezérlőelem készítésének folyamatához, az egyedi tevékenységhez is hozzárendelhető több .NET-attribútum, amelyek azt felügyelik, hogyan kell az összetevőt integrálni az integrált fejlesztői környezetben – például melyik bittérkép képe jelenjen meg az eszköztáron, melyik konfiguráció-párbeszéd jelenjen meg (ha megjelenik egyáltalán), amikor egy tulajdonságot konfigurálunk a Properties ablakban és így tovább.

Ha szeretnénk többet megtudni az egyedi tevékenységek létrehozásáról, a .NET Framework 3.5 SDK dokumentáció nagyon sok érdekes példával szolgál, beleértve egy „Send E-mail Activity" tevékenység létrehozását is. További részletekért egyszerűen keressük a megadott dokumentáció WF Samples csomópontjában található Custom Activities példákat (lásd a 26.29. ábrát).

Összefoglalás

A Windows Workflow Foundation (WF) egy olyan API, amely a .NET 3.0 verzióval jelent meg. Lényegében a WF teszi lehetővé egy alkalmazás belső üzleti folyamatainak közvetlen modellezését magában az alkalmazásban. Az általános munkafolyamat puszta modellezésén túl a WF teljes futtatómotort tartalmaz, és több szolgáltatást biztosít, amelyek kerek egésszé teszik az API funkcionalitását (tranzakciószolgáltatások, rögzítési szolgáltatás és nyomozási szolgáltatás stb.). Bár ez a bevezető fejezet nem vizsgálja részletesen ezeket a szolgáltatásokat egy termékszintű WF-alkalmazás egészen biztosan használja ezeket a lehetőségeket.

Munkafolyamat-alkalmazás létrehozásához a Visual Studio 2008 számos tervezői eszközt nyújt, beleértve egy munkafolyamat-tervezőt, konfigurálást a Properties ablak segítségével és (a legfontosabbat) a Windows Workflow eszköztárát. Több beépített *tevékenységet* találunk, amelyek hozzájárulnak egy meghatározott munkafolyamat teljes felépítéséhez. Miután modelleztük a munkafolyamatot, futtathatjuk a munkafolyamat-példányt a `WorkflowRuntime` típus segítségével, a választott hoszt felhasználásával.

6. rész

Felhasználói felületek

HUSZONHETEDIK FEJEZET

Windows Forms-programozás

A .NET platform 2001-es megjelenése óta az alaposztálykönyvtárak egy sajátos, Windows Forms nevű API-t is tartalmaznak (amelyet a `System.Windows.Forms.dll` szerelvény képvisel). A Windows Forms eszközrendszere biztosítja azokat a típusokat, amelyek a grafikus felhasználói felület (GUI) készítéséhez, az egyedi vezérlőelemek létrehozásához, az erőforrások (sztringtáblázatok, ikonok stb.) kezeléséhez, valamint egyéb, GUI-központú programozási feladatok elvégzéséhez szükségesek. Emellett egy GDI+ nevű, önálló (a `System.Drawing.dll` szerelvényben található) API további típusokat biztosít a programozóknak a 2D-s grafikák létrehozására, a hálózati nyomtatókkal történő kommunikációra, valamint a képadatok kezelésére.

A Windows Forms (és a GDI+) API-k a mai napig, a .NET 3.5 megjelenésekor is léteznek, és az alaposztálykönyvtárban még jó ideig (valószínűsíthetően örökre) jelen lesznek. A .NET 3.0 megjelenése óta azonban a Microsoft egy vadonatúj GUI-eszközrendszert is rendelkezésre bocsátott, Windows Presentation Foundation (WPF) névvel. A WPF óriási hajtóerőt biztosít a fejlesztés alatt álló felhasználói interfészek számára (lásd a következő fejezetet).

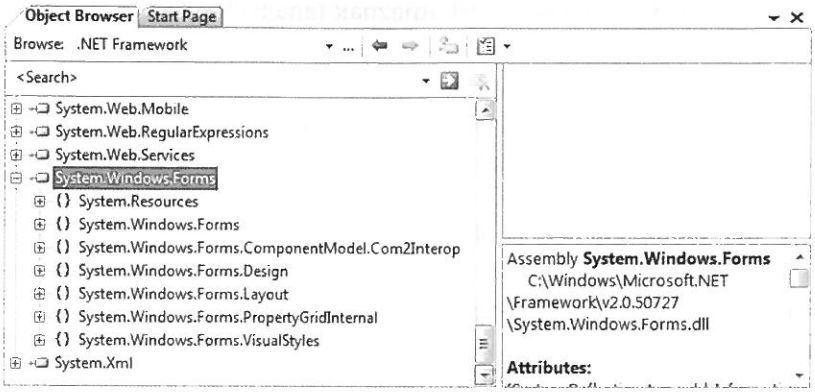
Most a középpontban azonban a hagyományos Windows Forms API-k állnak, ugyanis számos GUI-alkalmazás egyszerűen nem igényli azt a hajtóerőt, amelyet a WPF kínál. A WPF igazság szerint jó néhány grafikus felhasználói felülettel rendelkező alkalmazás számára túlzás volna. Továbbá a .NET univerzumban számos fenntartandó Windows Forms-alkalmazás létezik.

A jelen fejezetben megismerkedünk a Windows Forms programozási modellel, dolgozunk a Visual Studio 2008 beépített tervezőalkalmazásaival, kísérletezünk számos Windows Forms-vezérlőelemmel, valamint áttekintjük a GDI+-t alkalmazó grafikus programozást. Az információk egységes megjelenítéséhez létrehozunk egy (korlátozott funkcionalitással bíró) rajzolóalkalmazást.

Megjegyzés A könyv korábbi kiadásai három (meglehetősen hosszú) fejezetet szenteltek a Windows Forms API-nak. Mivel a WPF a .NET-es grafikusfelület-fejlesztés preferált eszközrendszere, ez a kiadvány a Windows Forms/GDI+ témának csupán ezt a fejezetet szenteli. A könyv vásárlói, az Apress webhelyéről ingyenesen letölthetik a korábbi, PDF-formátumú Windows Forms/GDI+ fejezeteket.

A Windows Forms-névterek

A Windows Forms API több száz típust (osztályt, interfészt, struktúrát, felsorolt típust és metódusreferenciát) tartalmaz, amelyek a `System.Windows.Forms.dll` szerelvény különböző névtereibe rendeződnek. A 27.1. ábra ezen névtereket ábrázolja a Visual Studio 2008 objektumböngészőn keresztül.



27.1. ábra: *A System.Windows.Forms.dll Windows Forms névterei*

A legfontosabb névtér mindenképpen a `System.Windows.Forms`. A `System.Windows.Forms` névtérben található típusokat az alábbi tágabb kategóriákba sorolhatjuk:

- *Alapvető infrastruktúra:* Ezek a típusok adják a Windows Forms-program (`Form`, `Application` stb.) alapvető működését, valamint azokat a különböző típusokat, amelyek megkönnyítik az együttműködést az örökölt ActiveX vezérlőelemekkel.
- *Vezérlőelemek:* Ezek a típusok gazdag felhasználói felületeket hoznak létre (`Button`, `MenuStrip`, `ProgressBar`, `DataGridView` stb.), amelyek mindegyike a `Control` ősosztályból származik. A vezérlőelemek tervezési időben konfigurálhatók, és láthatók (alapértelmezés szerint) futási időben.
- *Komponensek:* Ezek a típusok nem a `Control` ősosztályból származnak, de ugyanúgy a Windows Forms vizuális funkcióiért felelősek (`ToolTip`, `ErrorProvider` stb.). Számos komponens (például a `Timer` és a `BackgroundWorker`) futási időben nem látható, de tervezési időben vizuálisan konfigurálható.

- *Általános párbeszédablakok:* A Windows Forms jó néhány előre gyártott párbeszédablakot kínál az általános műveletekhez (`OpenFileDialog`, `PrintDialog`, `ColorDialog` stb.). Igény szerint, ha a standard párbeszédablakok nem felelnek meg, természetesen egyedi párbeszédablakokat is készíthetünk.

Mivel a `System.Windows.Forms` által kínált típusok teljes száma jóval meghaladja a százat, felesleges volna a Windows Forms-család listájához tartozó valamennyi tagot felsorolni. A fejezet áttanulmányozása után olyan szilárd alapot kapunk, amelyre később nyugodtan építhetünk. További információkért lapozzuk fel a .NET Framework 3.5 SDK dokumentációját.

Egyszerű Windows Forms-alkalmazás (IDE-mentes) létrehozása

Elvárásainknak megfelelően a modern .NET IDE-k (mint pl. a Visual Studio 2008, a C# 2008 Express vagy a SharpDevelop) számos űrlaptervezőt, vizuális szerkesztőt és integrált kódgeneráló eszközt (azaz varázslót) kínál a Windows Forms-alkalmazások könnyebb szerkesztése érdekében. Hasznosságuk ellenére ezek az eszközök meg is nehezíthetik a Windows Forms-elsajátítását, mert nehezebb felfedezni a lényeget az általuk generált nagy mennyiségű sablonkód között.

Az első Windows Forms-példát egy egyszerű szövegszerkesztővel és a C# parancssoros fordítóval hozzuk létre (a `csc.exe` alkalmazással történő munka leírása az előző kötet 2. fejezetében található).

Először hozzunk létre egy SimpleWinFormsApp mappát (közvetlenül a C meghajtón), nyissuk meg a Visual Studio 2008 parancssort, majd a kívánt szövegszerkesztővel hozzunk létre egy `SimpleWFApp.cs` nevű fájlt. Az új fájlban állítsuk össze az alábbi forráskódot, majd mentsük el a SimpleWinFormsApp mappába.

```
// Minimálisan szükséges névterek.
using System;
using System.Windows.Forms;
namespace SimpleWFApp
{
  // Ez az alkalmazásobjektum.
  class Program
```

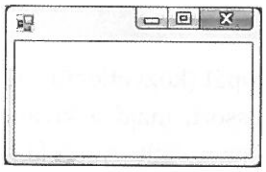
```
  {
    static void Main()
    {
      Application.Run(new MainWindow());
    }
  }

  // Ez a főablak.
  class MainWindow : Form {}
}
```

Ez a lehető legegyszerűbb Windows Forms-alkalmazás forráskódja. Minimálisan is szükségünk van egy olyan osztálytípusra, amely kibővíti a `Form` ősosztályt a `Main()` metódusra, amely a statikus `Application.Run()` metódus hívása (a `Form` és az `Application` részleteit lásd később). Az alkalmazást az alábbi utasítással fordíthatjuk le (az alapértelmezett reakciófájl `[csc.rsp]` automatikusan hivatkozik számos .NET-szerelvényre, köztük a `System.Windows.Forms.dll` és a `System.Drawing.dll` szerelvényre; lásd az előző kötet 2. fejezetében):

```
csc /target:winexe *.cs
```

Megjegyzés Technikai értelemben a parancssornál a `/target:exe` lehetőség segítségével készíthetünk Windows-alkalmazást; ekkor azonban azt tapasztaljuk, hogy a háttérben levő parancsablak láthatóvá válik (és az is marad, amíg be nem zárjuk a főablakot). A `/target:winexe` megadásával a végrehajtható fájl mint eredeti Windows Forms-alkalmazás futtatható (a látható parancsablak nélkül).

Ha futtatni szeretnénk az alkalmazást, egy átméretezhető, minimalizálható, maximalizálható és bezárható főablakot kapunk (lásd a 27.2. ábrát).



27.2. ábra: *Nagyon egyszerű Windows Forms-alkalmazás*

Az aktuális alkalmazás jól illusztrálja, milyen egyszerű lehet egy Windows Forms-alkalmazás. Ezután adjunk a `MainWindow` típushoz olyan egyedi konstruktort, amely lehetővé teszi, hogy a hívó különféle tulajdonságokat állítson be a megjelenítendő ablakban. Például:

```
// Ez a főablak.
class MainWindow : Form
{
  public MainWindow(string title, int height, int width)
  {
    // Határozzunk meg néhány tulajdonságot a szülőosztályból.
    Text = title;
    Width = width;
   Height = height;

    // Származtatott metódus az űrlapnak a képernyő
    // közepére helyezéséhez.
    CenterToScreen();
  }
}
```

Ezután módosítsuk az `Application.Run()` hívását a következőre:

```
static void Main()
{
  Application.Run(new MainWindow("My Window", 200, 300));
}
```

Minden jól működő ablak különböző felhasználói interfészelemeket (menürendszereket, állapotsávokat, gombokat stb.) igényel a bevitelhez. Annak megértéséhez, hogy egy `Formból` származó típus hogyan tartalmazhat ilyen elemeket, meg kell értenünk a `Controls` tulajdonság, valamint a mögötte álló vezérlőelemek szerepét.

A vezérlőelemek gyűjteményének feltöltése

A `System.Windows.Forms.Control` alaposztály (amely a `Form` típus származtatási lánca) határozza meg a *Controls* tulajdonságot. Ez a tulajdonság egyedi gyűjteményt burkol be a `ControlsCollection` nevű `Control` osztályba beágyazva. A gyűjtemény (ahogy a neve is sugallja) hivatkozik minden egyes, a származtatott típus által fenntartott felhasználóifelület-elemre. Ahogy a többi tároló, ez a típus is támogat számos olyan metódust, mellyel felhasználóifelület-elemeket tudunk beszúrni, eltávolítani, valamint megkeresni (lásd a 27.1. táblázatot).

Tag	Jelentés
`Add()` `AddRange()`	Új, a `Control` osztályból származó típust (vagy típustömböt) szúr be a gyűjteménybe.
`Clear ()`	Eltávolítja a gyűjtemény valamennyi bejegyzését.
`Count`	Visszaadja a gyűjtemény elemeinek a számát.
`GetEnumerator()`	Visszaadja a gyűjtemény `IEnumerator` interfészét.
`Remove()` `RemoveAt()`	Vezérlőelemet távolít el a gyűjteményből.

27.1. táblázat: *ControlCollection tagok*

Ha `Form`-leszármazott típusokat szeretnénk hozzáadni a felhasználói felülethez, az alábbi, jól meghatározott lépések sorát kell követnünk:

- Határozzuk meg a felhasználói felület elemének egy, a `Formból` származó osztályon belüli tagváltozóját.
- Konfiguráljuk a felhasználói felület elemének megjelenését és működését.
- A `Controls.Add()` metódus segítségével adjuk a felhasználói felület elemét az űrlap `ControlsCollection` tárolójához.

Tételezzük fel, hogy a File > Exit menürendszer támogatásához módosítani szeretnénk a `MainWindow` osztályt. A releváns módosítások, valamint a hozzá tartozó magyarázatok a következők:

```
class MainWindow : Form
{
  // Egyszerű menürendszer tagjai.
  private MenuStrip mnuMainMenu = new MenuStrip();
  private ToolStripMenuItem mnuFile = new ToolStripMenuItem();
  private ToolStripMenuItem mnuFileExit = new ToolStripMenuItem();

  public MainWindow(string title, int height, int width)
  {
    ...
    // A menürendszer létrehozásának metódusa.
    BuildMenuSystem();
  }
```

```
  private void BuildMenuSystem()
  {
    // Adjuk a File menüelemet a főmenühöz.
    mnuFile.Text = "&File";
    mnuMainMenu.Items.Add(mnuFile);

    // Most adjuk az Exit menüt a File menühöz.
    mnuFileExit.Text = "E&xit";
    mnuFile.DropDownItems.Add(mnuFileExit);
    mnuFileExit.Click +=
          new System.EventHandler(this.mnuFileExit_Click);

    // Végül állítsuk be az űrlap menüjét.
    Controls.Add(this.mnuMainMenu);
    MainMenuStrip = this.mnuMainMenu;
  }

  // A File | Exit esemény kezelője.
  private void mnuFileExit_Click(object sender, EventArgs e)
  {
    Application.Exit();
  }
}
```

A `MainWindow` típus ez alkalommal három tagváltozót tartalmaz. A `MenuStrip` típus a menürendszer teljességét ábrázolja, ahol egy adott `ToolStripMenuItem` jeleníti meg a hoszt által támogatott főmenü- (pl. File) vagy almenüelemet (pl. Exit).

Megjegyzés Korábban a `MainMenu` típus tetszőleges számú `MenuItem` objektum tartására szolgált. A `MenuStrip` vezérlőelem (amelyet a .NET 2.0-ben ismerhettünk meg) hasonló a `MainMenu` típushoz, de a `MenuStrip` „normál menüelemeken" kívüli vezérlőelemeket (legördülő listadobozokat, szövegdobozokat stb.) is tartalmazhat.

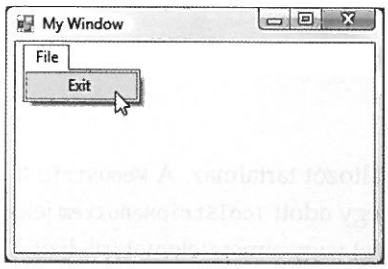
A menürendszer konfigurációja a `BuildMenuSystem()` segédfüggvényen belül történik. Minden `ToolStripMenuItem` szövegét a Text tulajdonságon keresztül adhatjuk meg, s ezek mindegyikéhez olyan sztringliterál tartozik, amely beágyazott & karaktert tartalmaz. Ez a szintaxis határozza meg az Alt billentyűhöz tartozó gyorsbillentyűt, azaz az Alt + F kombinációval aktiválhatjuk a File menüt, míg az Alt + X billentyűkombináció az Exit menüt aktiválja. A File `ToolStripMenuItem` objektumhoz (`mnuFile`) alelemeket a `DropDownItems` tulajdonság révén adtunk. A `MenuStrip` objektumhoz pedig az Items tulajdonságon keresztül adtuk hozzá a főmenüelemeket.

A kész menürendszer (a `Controls` tulajdonságon keresztül) egy vezérlőelemgyűjteményhez adódik; majd a `MenuStrip` objektumot az örökölt `MainMenuStrip` tulajdonsághoz rendeljük. Bár ez felesleges lépésnek tűnik, ám egy olyan megha-

tározott tulajdonság révén, mint a `MainMenuStrip`, lehetőség van dinamikusan meghatározni, hogy melyik menürendszer jelenjen meg a felhasználó számára, például a felhasználói vagy a biztonsági beállítások alapján.

A File > Exit menü `Click` eseményét azért kezeljük, hogy értesüljünk róla, hogy a felhasználó mikor választja ki ezt az almenüt. A `Click` esemény a szokásos, `System.EventHandler` nevű metódusreferencia-típussal működik együtt. A `Click` esemény csak azokat a metódusokat hívja, amelyeknek `System.Object` az első és `System.EventArgs` a második paramétere. Tehát implementáltuk a metódusreferencia célját (`mnuFileExit_Click`) annak érdekében, hogy a statikus `Application.Exit()` metódus felhasználásával kilépjünk az egész Windows-alkalmazásból.

Az alkalmazás újrafordítása és futtatása után, láthatjuk, hogy az egyszerű ablak egy egyedi menürendszert jelenít meg (lásd a 27.3. ábrát).



27.3. ábra: *Egyszerű ablak egyszerű menürendszerrel*

A System.EventArgs és a System.EventHandler szerepe

A `System.EventHandler` egyike annak a számos metódusreferencia-típusnak, amelyeket az eseménykezelő folyamat során a Windows Forms (és az ASP.NET) API-kon belül használunk. Ez a metódusreferencia csak olyan metódusokra mutathat, ahol az első argumentum az eseményt küldő típusra hivatkozó `System.Object` típus. Ha például módosítani szeretnénk a `mnuFileExit_Click()` metódus implementációját az alábbiak szerint:

```
private void mnuFileExit_Click(object sender, EventArgs e)
{
  MessageBox.Show(string.Format("{0} sent this event",
                  sender.ToString()));
  Application.Exit();
}
```

akkor megerősíthetnénk, hogy a `mnuFileExit` típus küldte az eseményt, ugyanis az alábbi sztring:

```
"E&xit sent this event"
```

jelenik meg az üzenetdobozban. Kérdés, mire jó a második, a `System.EventArgs` argumentum. A `System.EventArgs` típus szerepe valójában kicsi: csak az *objectet* terjeszti ki, de gyakorlatilag semmit nem ad hozzá magához a funkcionalitáshoz:

```
public class EventArgs
{
  public static readonly EventArgs Empty;
  static EventArgs();
  public EventArgs();
}
```

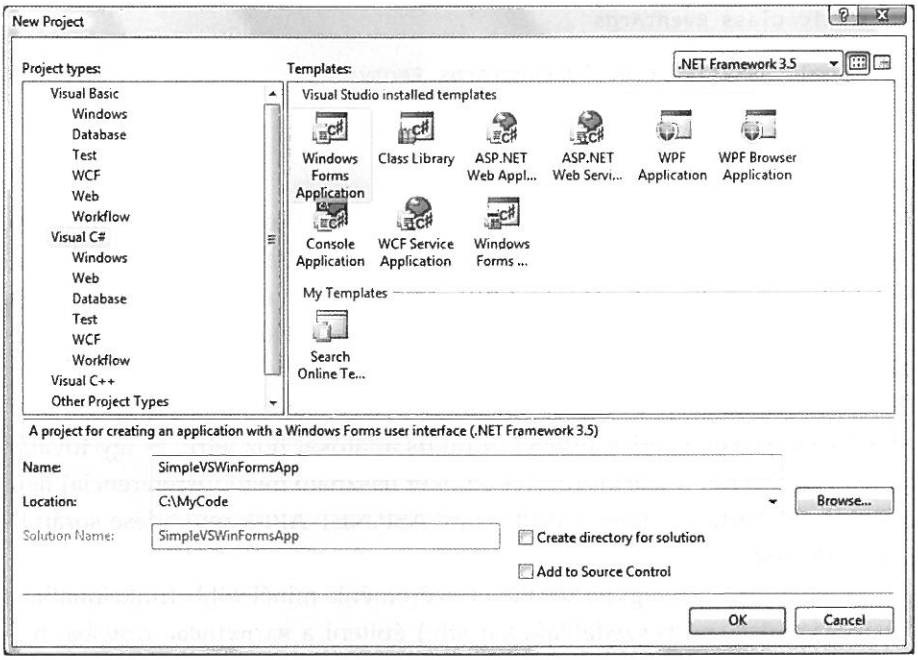
Ám ez a típus kifejezetten hasznos a .NET-eseménykezelés teljes sémájában, ugyanis a `System.EventArgs` a szülője sok más (igencsak hasznos) származtatott típusnak. A `MouseEventArgs` típus például kibővíti az `EventArgs` típust, hogy az egér aktuális állapotára vonatkozó részletekkel szolgáljon. A `KeyEventArgs` szintén az `EventArgs` típust terjeszti ki, és a billentyűzet állapotára (azaz a lenyomott billentyűkre) vonatkozó részletekkel szolgál; a `PaintEventArgs` az `EventArgs` kiterjesztésével grafikus adatokat hoz létre, és így tovább. Számos `EventArgs`-leszármazott (és az őket használó metódusreferencia) nem a Windows Forms, hanem a WPF és az ASP.NET API-k működése során figyelhető meg.

Ha egyszerű szövegszerkesztővel szeretnénk minél több funkcionalitást (állapotsávokat, párbeszédablakokat stb.) építeni a `MainWindow` típusba, manuálisan kellene összeállítanunk valamennyi, a vezérlőelemeket konfiguráló logikát. Szerencsére a Visual Studio 2008 számos integrált tervezővel rendelkezik, amelyek megoldják ezeket a részleteket.

Forráskód A SimpleWinFormsApp projektet a forráskódkönyvtár 27. alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A Visual Studio Windows Forms-projektsablona

Ha ki szeretnénk használni a Visual Studio 2008 Windows Forms-tervezőeszközeit, először válasszuk ki a Windows Application projektsablont a File > New Project menüelem segítségével. Hogy megismerjük az alapvető Windows Forms-tervezőeszközöket, hozzunk létre egy új, SimpleVSWinForms-App nevű alkalmazást (lásd a 27.4. ábrát).



27.4. ábra: *A Visual Studio 2008 Windows Forms-projektsablona*

A vizuális tervezőfelület

Készítsük el ismét az előző alkalmazást, de most már használjuk a tervezőeszköz nyújtotta lehetőségeket. Ha létrehoztuk a Windows Forms projektet, a Visual Studio 2008 olyan tervezőfelületet jelenít meg, amelyen számos vezérlőelemet elhelyezhetünk az áthúzás funkció segítségével. Ugyanezzel a tervezővel a készülő alkalmazásablak kezdeti méretét is beállíthatjuk a széleken található kapaszkodó megragadásával és arrébb húzásával (lásd a 27.5. ábrát).