Chapter 10

Custom Attributes

In this chapter:

In Chapter 9, "Reflection," I illustrate how you can use it to enumerate the Microsoft .NET Framework attributes associated with code entities, for example, to determine programmatically whether a class is serializable. In this chapter, I focus on how you can write your own custom attributes and use them to implement advanced programming techniques.

Applying .NET attributes is vaguely similar to assigning properties to a Microsoft Windows Forms or Web Form control. When you assign a value to the Location and BackColor properties, you know that a piece of code in the .NET Framework will eventually read those properties and change the position and the background color of that control. Properties promote *declarative programming*, by which you state what you want to achieve and let another piece of code process the property and run the actual instructions that carry out the assignment. Similarly, when you label a type or a class member with an attribute, you declare how that type or that member should be processed and let another piece of code perform the actual action.

This description holds true with all the .NET attributes I illustrate in earlier (and later) chapters, and it's also true with the custom attributes that you write. The ability to define and apply custom attributes is among the most underestimated of .NET features. That's why I devote an entire chapter to this topic.

Note To avoid long lines, code samples in this chapter assume that the following using statements are used at the top of each source file:

```
using System;
using System.CodeDom.Compiler;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Reflection;
using System.Runtime.Serialization.Formatters.Binary;
using System.Text;
using System.Text.RegularExpressions;
using System.Windows.Forms;
using System.Xml.Serialization;
```

Introducing Custom Attributes

As you learned in previous chapters, attributes are pieces of metadata that you attach to code entities—assemblies, classes, methods, or individual fields—to affect the behavior of the Microsoft C# compiler, the JIT compiler, or other portions of the .NET runtime.

Most of the time, you'll use only attributes that are defined in the .NET Framework and documented in the .NET software development kit (SDK). Attributes are simply .NET classes, though, and nothing prevents you from designing your own attribute types. The main difference between your custom attributes and predefined .NET attributes is that custom attributes require that you write the code that discovers and uses them.

Building a Custom Attribute Class

A custom attribute is a class that inherits from System.Attribute. Its name must end with *Attribute* and it is marked with an AttributeUsage attribute that tells the compiler to which program entities the attribute can be applied: classes, modules, methods, and so on. A custom attribute class can contain fields, properties, and methods that accept and return values only of the following types: bool, byte, short, int, long, char, float, double, string, object, System.Type, and public Enum. It can also receive and return one-dimensional arrays of one of the preceding types. A custom attribute class must have one or more public constructors, and it's customary to expose constructors that let developers specify the mandatory arguments to be passed to the attribute.

Note Custom attributes must be visible to at least two assemblies: the assembly where you apply the attribute and the assembly that reads and processes the applied attributes. For this reason, custom attribute types are usually defined in DLL assemblies. For simplicity's sake, all the sample attribute types I illustrate in this chapter are gathered in a demo DLL named CustomAttributes and are contained in a namespace also named CustomAttributes. You should add a reference to this DLL in all the projects where you apply or process these custom attributes.

The following example shows a custom attribute class that lets you annotate any class or class member with the name of the author, the source code version when the member was completed, and an optional property that specifies whether the code has been completely tested.

```
// The AttributeTargets.All value means that this attribute
// can be used with any program entity.
[AttributeUsage(AttributeTargets.All)]
public class VersionAttribute : System.Attribute
{
   // The Attribute constructor takes two required values.
   public VersionAttribute(string author, float version)
   {
      m_Author = author;
      m_Version = version;
   }

   // Private fields
   private string m_Author;
   private float m_Version;
   private bool m_Tested;

   public string Author
   {
      get { return m_Author; }
   }

   public float Version
   {
      get { return m_Version; }
   }

   public bool Tested
   {
      get { return m_Tested; }
      set { m_Tested = value; }
   }
}
```

Microsoft guidelines dictate that all the values accepted in the attribute constructor be implemented as read-only properties, whereas arguments that can't be set through the constructor must be implemented as read-write properties because they can be assigned only through named parameters, as in the following code:

```
[Version("John", 1.10F, Tested = true)]
public class TestVersionClass
{
   …
}
```

An attribute can expose fields, but in general encapsulating a value inside a property is preferable. Attribute classes can also include other kinds of members, but in practice this happens infrequently: an attribute is just a repository for metadata values that are read by other

programs; therefore, fields and properties are all you need most of the time. Attributes are meant to be discovered by a piece of code running in a different assembly, and therefore these classes typically have a public scope.

The argument passed to the AttributeUsage attribute specifies that the VersionAttribute attribute—or just Version, because the trailing Attribute portion of the name can be omitted—can be used with any program entity. The argument you pass to the AttributeUsage constructor is a bit-coded value formed by adding one or more elements in this list: Assembly (1), Module (2), Class (4), Struct (8), Enum (16), Constructor (32), Method (64), Property (128), Field (256), Event (512), Interface (1,024), Parameter (2,048), Delegate (4,096), ReturnValue (8,192), or All (16,383, the sum of all preceding values).

The AttributeUsage attribute supports two additional properties, which can be passed as named arguments in its constructor. The AllowMultiple property specifies whether the attribute being defined—VersionAttribute, in this case—can appear multiple times inside angle brackets. The Inherited attribute tells whether a derived class inherits the attribute. The default value for both properties is false.

The Conditional attribute is an example of an attribute that supports multiple instances and is also an example of an attribute that's inherited by derived classes. If the Conditional attribute were implemented in C#, its source code would be more or less as follows:

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = true, Inherited = true)]
public class ConditionalAttribute : System.Attribute
{
   private string m_ConditionString;

   // The constructor method
   public ConditionalAttribute(string conditionString)
   {
      this.ConditionString = conditionString;
   }

   // The only property of this attribute class
   public string ConditionString
   {
      get { return m_ConditionString; }
      set {m_ConditionString = value; }
   }
}
```

Notice that the ConditionalAttribute type violates Microsoft's own guidelines because the ConditionString writable property can also be set through the constructor.

Let's go back to the Version attribute. You can apply it to a class and its members:

```
[Version("John", 1.01F)]
public class TestVersionClass
{
   [Version("Robert", 1.01F, Tested = true)]
```

```
   public void MyProc()
   {
      …
   }

   [Version("Ann", 1.02F)]
   public long MyFunction()
   {
      …
   }
}
```

Compile the class in a Console project named TestApplication and read on to see how you can discover the attribute.

Reflecting on a Custom Attribute

As I emphasized previously, an attribute is a piece of information stored somewhere in its assembly's metadata tables, waiting for a program—let's call it the *agent*—to extract it and use it. When you apply .NET standard attributes, the agent program is the C# compiler, the JIT compiler, or the CLR; when you apply a custom attribute, you must write the agent code yourself. Such an agent code can be in the same DLL where the attribute is defined (if it's meant to be invoked from other assemblies) or in a separate EXE file (if it runs as a stand-alone program).

In our first example, the agent program can be as simple as a piece of code that scans an assembly and displays a report that lists which types have been authored by whom and which code members have been tested. Create another Console application, name it ShowVersion, add a reference to the CustomAttributes DLL, and type the following code:

```
using System;
using System.Reflection;
using AttributeLibrary;

namespace ShowVersion
{
   sealed public class App
   {
      public static void Main(string[] args)
      {
         // Read the assembly whose path is passed as an argument; error if not found.
         Assembly asm = Assembly.LoadFile(args[0]);
         // Display the header.
         Console.WriteLine("{0,-40}{1,-12}{2,-10}{3,-6}", "Member",
            "Author", "Version", "Tested");
         Console.WriteLine(new string('-', 68));

         // Iterate over all public and private types.
         foreach ( Type type in asm.GetTypes() )
         {
            // Extract the attribute associated with the type.
            VersionAttribute attr = (VersionAttribute)
               Attribute.GetCustomAttribute(type, typeof(VersionAttribute ) );
```

```
            if ( attr != null )
            {
               Console.WriteLine("{0,-40}{1,-12}{2,-10}{3,-6}",
                  type.FullName, attr.Author, attr.Version, attr.Tested);
            }

            // Iterate over all public and private members.
            foreach ( MemberInfo mi in type.GetMembers(BindingFlags.Public
               | BindingFlags.NonPublic
               | BindingFlags.Instance | BindingFlags.Static) )
            {
               // Extract the attribute associated with each member
               attr = (VersionAttribute) Attribute.GetCustomAttribute(mi,
                  typeof(VersionAttribute)) ;
               if ( attr != null )
               {
                  Console.WriteLine("    {0,-36}{1,-12}{2,-10}{3,-6}",
                     mi.Name, attr.Author, attr.Version, attr.Tested);
               }
            }
         }
      }
   }
}
```

(You might want to refer to Chapter 9 to read about the many techniques you can adopt to reflect on a custom attribute.) Compile the application and run it, passing the path of the TestApplication.exe assembly as an argument on the command line. You should see this report in the console window:

```
Member                                  Author      Version   Tested
------------------------------------------------------------------------
TestApplication.TestVersionClass        John        1.01      False
    MyProc                              Robert      1.01      True
    MyFunction                          Ann         1.02      False
```

Thanks to reflection and custom attributes you now have a report utility that lets you quickly display the author, the version, and the tested status of all the methods inside a compiled assembly. Sure, you can author a similar utility that reads special remarks in source code, but such a utility wouldn't work on compiled assemblies and, if your team works with other programming languages, you would be forced to write a different parser for each distinct language.

You can extend the ShowVersion utility to spot outdated or untested code quickly, and you can extend the VersionAttribute type with other properties, such as DateCreated and DateModified. You might automatically run ShowVersion at the end of your compilation process—for example, as a postbuild compilation step—to ensure that only fully tested code makes its way to your customers.

A Custom Attribute for CSV Serialization

The .NET Framework offers great support for serializing an object instance to and from XML, by means of the XmlSerializer type:

```
// Create a Person object.
Person pers = new Person();
pers.FirstName = "John";
pers.LastName = "Evans";

// Serialize it to a file.
XmlSerializer ser = new XmlSerializer(typeof(Person));
using ( FileStream fs = new FileStream(@"c:\person.xml", FileMode.Create) )
{
   ser.Serialize(fs, pers);
}

// Read it back. (Reuses the same serializer object.)
using ( FileStream fs = new FileStream(@"c:\person.xml", FileMode.Open) )
{
   Person p = (Person) ser.Deserialize(fs);
   Console.WriteLine("{0} {1}", p.FirstName, p.LastName);    // => John Evans
}
```

At the end of the serialization process, the person.xml file contains the following text:

```
<?xml version="1.0"?>
<Person xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <FirstName>John</FirstName>
  <LastName>Evans</LastName>
</Person>
```

As you can see, each property of the Person type is rendered as an XML element named after the property itself. When you import XML data produced by another program, however, you have no control over the XML schema adopted during the serialization process; to solve this potential problem, you can change the default behavior of the XmlSerializer type, for example, change the names of XML elements and decide that properties be rendered as XML attributes rather than as elements, as in this XML fragment:

```
<PersonalData xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      first="John" last="Evans">
</PersonalData>
```

As you might have guessed, you can exert this degree of control on the XML serialization process by applying attributes to the members of the Person class. For example, you can produce the previous XML file by defining the Person type as follows:

```
[XmlRoot("PersonalData")]
public class Person
{
   [XmlAttributeAttribute("first")]
```

```
   public string FirstName;
   [XmlAttributeAttribute("last")]
   public string LastName;
}
```

Unfortunately, not all the world out there speaks XML. This is especially true for legacy applications running on mainframes, which often exchange data in a simple comma-separated value (CSV) format. In this chapter, I show you how to implement CSV serialization by means of a powerful and elegant technique based on a CsvSerializer type and the CsvField custom attribute.

Let's start by defining the CsvFieldAttribute type, which can be applied to individual fields and properties of a class to affect how instances of that class are rendered in CSV format. This custom attribute has three properties: Index is the position of the field in the output; Quoted is a Boolean property that specifies whether the field or property's value must be enclosed in double quotation marks; Format is an optional format string, which gives you better control over how dates and numbers are written to the CSV file. The index value is mandatory, and therefore the Index property is marked as read-only and must be assigned from inside the attribute's constructor; the other two properties are optional and can be assigned by means of named arguments.

```
[AttributeUsage(AttributeTargets.Field | AttributeTargets.Property)]
public class CsvFieldAttribute : Attribute, IComparable

    // These would be properties in a real-world implementation.
{
   public readonly int Index;
   public bool Quoted = false;
   public string Format = "";

   public CsvFieldAttribute(int index)
   {
      this.Index = index;
   }

   // Attributes are sorted on their Index property.
   public int CompareTo(object obj)
   {
      return this.Index.CompareTo((obj as CsvFieldAttribute).Index);
   }
}
```

Unlike most attribute types, the CsvFieldAttribute class exposes an interface, IComparable, and its Compare method. The reason for this design decision will be clear shortly. Let's now define an Employee class that uses the CsvField attribute:

```
using AttributeLibrary;

public class Employee
{
   [CsvField(1, Quoted = true)]
   public string FirstName;
```

```
   [CsvField(2, Quoted = true)]
   public string LastName;
   [CsvField(3, Format = "dd/M/yyyy")]
   public DateTime BirthDate;

   public decimal m_Salary;

   [CsvField(4, Format = "######.00")]
   public decimal Salary
   {
      get { return m_Salary; }
      set { m_Salary = value; }
   }

   // Constructors and other methods…
   public Employee()
   {}
}
```

The Employee class requires a default constructor for the CsvSerializer to work correctly. (The XmlSerializer has a similar requirement.) I have included an explicit parameterless constructor, in case you wish to add other constructors later.

The toughest part is writing the CsvSerializer class, which uses reflection to read the CsvField attribute associated with each field and property exposed by the type being serialized. By implementing the CsvSerializer class as a generic type you can write more concise and efficient code:

```
public class CsvSerializer<T> where T : new()
{
   private Type type;
   private string separator;
   private SortedDictionary<CsvFieldAttribute, MemberInfo> attrList =
      new SortedDictionary<CsvFieldAttribute,MemberInfo>();
   private string rePattern;

   // Constructors
   public CsvSerializer() : this(",")
   { }

   public CsvSerializer(string separator)
   {
      this.type = typeof(T);
      this.separator = separator;
      BuildAttrList();
   }
   …
}
```

The CsvSerializer class assumes that the field separator is a comma, but it offers an alternate constructor that enables you to specify a different separator, for example, the semicolon. While inside the constructor, the CsvSerializer class parses all the members of the T type and creates a list of (CsvFieldAttribute, MemberInfo) pairs to speed up the actual serialization and

deserialization process. Such a list is stored in a SortedDictionary object and is sorted on the attribute's Index property. (Here's why the CsvFieldAttribute type implements the IComparable interface.) In addition to creating the sorted dictionary, the BuildAttrList procedure creates a regular expression that can parse individual lines of a data file in CSV format. I won't describe this regular expression in detail because I explain a similar technique in the section titled "Parsing Data Files" in Chapter 6, "Regular Expressions."

```
// Build the sorted list of (attribute, MemberInfo) pairs.
private void BuildAttrList()
{
   // Create the list of public members that are flagged with the
   // CsvFieldAttribute, sorted on the attribute's Index property.
   foreach ( MemberInfo mi in type.GetMembers() )
   {
      // Get the attribute associated with this member.
      CsvFieldAttribute attr = (CsvFieldAttribute)
         Attribute.GetCustomAttribute(mi, typeof(CsvFieldAttribute));
      if ( attr != null )
      {
         // Add to the list of attributes found so far, sorted on Index property.
         attrList.Add(attr, mi);
      }
   }

   // Create the Regex pattern and format string output pattern.
   StringBuilder sb = new StringBuilder();

   foreach ( KeyValuePair<CsvFieldAttribute, MemberInfo> de in attrList )
   {
      // Add a separator to the pattern, but only from the second iteration onward.
      if ( sb.Length > 0 )
      {
         sb.Append(separator);
      }
      sb.Append(" *");

      // Get attribute and MemberInfo for this item.
      CsvFieldAttribute attr = de.Key;
      MemberInfo mi = de.Value;

      // Append to the Regex for this element.
      if ( !attr.Quoted )
      {
         sb.AppendFormat("(?<{0}>[^{1}]+)", mi.Name, separator);
      }
      else
      {
         sb.AppendFormat("\"(?<{0}>[^\"]+)\"", mi.Name);
      }
      sb.Append(" *");
   }
   // Set the pattern.
   rePattern = sb.ToString();
}
```

The CsvSerializer type exposes two Serialize methods, one that works with files and the other that works with any StreamWriter object. (The former method delegates its job to the latter.) Both methods take an ICollection<T> generic collection in their second argument, so you can serialize entire arrays and collections of T instances.

Serializing an individual object is easy. The list of serializable members is already in the attrList dictionary and is sorted in the correct sequence, so it's just a matter of reading the corresponding field or property, outputting its value to the stream, and applying the correct format string if one has been specified by means of the CsvField attribute:

```
// Serialize to text file.
public void Serialize(string fileName, ICollection<T> col)
{
   using ( StreamWriter writer = new StreamWriter(fileName) )
   {
      Serialize(writer, col);
   }
}

// Serialize to a stream.
public void Serialize(StreamWriter writer, ICollection<T> col)
{
   foreach ( T obj in col )
   {
      // This is the result string.
      StringBuilder sb = new StringBuilder();

      foreach ( KeyValuePair<CsvFieldAttribute, MemberInfo> kvp in attrList )
      {
         // Append the separator (but not at the first element in the line).
         if ( sb.Length > 0 )
         {
            sb.Append(separator);
         }
         // Get attribute and MemberInfo.
         CsvFieldAttribute attr = kvp.Key;
         MemberInfo mi = kvp.Value;

         // Get the value of the field or property, as an object.
         object fldValue = null;
         if ( mi is FieldInfo )
         {
            fldValue = ( mi as FieldInfo ).GetValue(obj);
         }
         else if ( mi is PropertyInfo )
         {
            fldValue = ( mi as PropertyInfo ).GetValue(obj, null);
         }

         // Get the format to be used with this field/property value.
         string format = "{0}";
         if ( attr.Format != "" )
         {
            format = "{0:" + attr.Format + "}";
         }
```

```
            if ( attr.Quoted )
            {
               format = "\"" + format + "\"";
            }
            // Call the ToString method, with a format argument if specified.
            sb.AppendFormat(format, fldValue);
         }
         // Output to the stream.
         writer.WriteLine(sb.ToString());
      }
   }
```

Deserializing from a file or a stream is only marginally more difficult. The Deserialize method uses the regular expression that was built when the CsvSerializer object was instantiated, and it extracts the value of each regular expression's Group. The Deserialize methods take an optional argument that specifies whether the first line should be ignored: this feature enables you to process data files in which the first line contains a header that lists the names of fields:

```
// Deserialize a text file.
public T[] Deserialize(string fileName, bool ignoreFieldHeader)
{
   using ( StreamReader reader = new StreamReader(fileName) )
   {
      return Deserialize(reader, ignoreFieldHeader);
   }
}

// Deserialize from a stream.
public T[] Deserialize(StreamReader reader, bool ignoreFieldHeader)
{
   // The result array
   List<T> list = new List<T>();
   // Create a compiled Regex for best performance.
   Regex re = new Regex("^" + rePattern + "$", RegexOptions.Compiled);

   // Skip the field header, if necessary.
   if ( ignoreFieldHeader )
   {
      reader.ReadLine();
   }

   while ( !reader.EndOfStream )
   {
      string text = reader.ReadLine();

      Match m = re.Match(text);
      if ( m.Success )
      {
         // Create an instance of the target type.
         T obj = new T();
         // Set individual properties.
         foreach ( KeyValuePair<CsvFieldAttribute, MemberInfo> de in attrList )
         {
            // Get attribute and MemberInfo.
            CsvFieldAttribute attr = de.Key;
            MemberInfo mi = de.Value;
```

```
            // Retrieve the string value.
            string strValue = m.Groups[mi.Name].Value;

            if ( mi is FieldInfo )
            {
               FieldInfo fi = (FieldInfo) mi;
               object fldValue = Convert.ChangeType(strValue, fi.FieldType);
               fi.SetValue(obj, fldValue);
            }
            else if ( mi is PropertyInfo )
            {
               PropertyInfo pi = (PropertyInfo) mi;
               object propValue = Convert.ChangeType(strValue, pi.PropertyType);
               pi.SetValue(obj, propValue, null);
            }
         }
         // Add this object to result.
         list.Add(obj);
      }
   }
   // Return the array of instances.
   return list.ToArray();
}
```

Using the CsvSerializer is simple:

```
// Create and fill a sample array of Employee objects.
Employee[] arr;
…
// Serialize all the objects to a file.
CsvSerializer<Employee> ser = new CsvSerializer<Employee>();
ser.Serialize("employees.txt", arr);
…
// Deserialize them from the file back to a different array.
Employee[] arr2 = ser.Deserialize("employees.txt", false);
```

The CsvSerializer type and its CsvField companion attribute enable you to serialize an object to the CSV format by writing very little code. You might achieve the same result by adopting a technique that isn't based on attributes, but you can hardly reach the same degree of code reusability and ease of maintenance. For example, if the order of fields in the CSV file changes, you simply need to edit one or more attributes in the Employee class without changing code elsewhere in the application and without having to worry about side effects.

Note Often you can replace, or complement, custom attributes burnt in source code with an external file that holds the same kind of information. In this particular case, you might use an XML file that contains the order of fields and the field format so that you simply need to provide a different XML file when the file format changes, without having to recompile the application. Both solutions have their pros and cons. XML files often give you more flexibility—for example, they can store hierarchical information—whereas custom attributes ensure that the metadata always travels with the code it refers to. Also, attributes are a more natural choice when the metadata doesn't change often and when, if the metadata changes, you need to recompile the code anyway. In this particular case, for example, if a new version of the CSV file has additional fields, you need to recompile the project anyway, and therefore custom attributes can be the preferred approach to store the metadata.

Building a Benchmark Tool

Microsoft Visual Studio 2005 offers many tools for testing your applications. However, when it's time to write benchmarks, the .NET Framework gives you the Stopwatch type, and that's it. If you need to benchmark multiple routines, compare and sort their results, and write a little report, you have to write all the plumbing code. You *had* to, at least.

Writing a tool that automates the production of your benchmarks requires very little effort. First, you need an attribute to mark the methods that you want to benchmark. For simplicity's sake, this version uses public fields instead of properties:

```
[AttributeUsage(AttributeTargets.Method)]
public class BenchmarkAttribute : Attribute, IComparable<BenchmarkAttribute>
{
   public BenchmarkAttribute(string group)
   {
      if ( group == null )
      {
         group = "";
      }
      this.Group = group;
   }

   // In the companion code, these are properties.
   public readonly string Group = "";
   public string Name = "";
   public double NormalizationFactor = 1;

   public int CompareTo(BenchmarkAttribute other)
   {
      int res = this.Group.CompareTo(other.Group);
      if ( res == 0 )
      {
         res = this.Name.CompareTo(other.Name);
      }
      return res;
   }
}
```

This class implements the IComparable<T> generic interface because instances of the attribute will be used as keys in a SortedList collection, as you'll see shortly. You can apply the Benchmark attribute only to a void or nonvoid static method that takes no arguments, for example, methods in a static class:

```
public static class TestBenchmarkModule
{
   [Benchmark("Concatenation", NormalizationFactor = 100)]
   public static void TestStringBuilder()
   {
      StringBuilder sb = new StringBuilder();
      for ( int i = 1; i <= 1000000; i++ )
      {
         sb.Append(i);
```

```
      }
   }

   [Benchmark("Concatenation")]
   public static void TestString()
   {
      string s = "";
      for ( int i = 1; i <= 10000; i++ )
      {
         s += i.ToString();
      }
   }

   [Benchmark("Division", Name = "Integer division")]
   public static int TestIntegerDivision()
   {
      int res = 0;
      for ( int i = 1; i <= 10000000; i++ )
      {
         res = 1000000 / i;
      }
      return res;
   }

   [Benchmark("Division", Name = "Long division")]
   public static long TestLongDivision()
   {
      long res = 0;
      for ( int i = 1; i <= 10000000; i++ )
      {
         res += 1000000 / i;
      }
      return res;
   }

   [Benchmark("Division", Name = "Double division")]
   public static double TestDoubleDivision()
   {
      double res = 0;
      for ( int i = 1; i <= 10000000; i++ )
      {
         res += 1000000 / i;
      }
      return res;
   }
}
```

The only mandatory argument of the Benchmark attribute is its Group property: benchmark methods that must be compared with each other must have the same Group name, and the tool you're going to build sorts results by their Group name. In the previous example, two benchmark groups, Concatenation and Division, contain two and three benchmarks, respectively. A benchmark also has a name, which defaults to the method name: this name identifies the individual benchmark in the report, and thus you should select a descriptive text for this property.

To see when the NormalizeFactor property can be useful, consider the TestString method, which appends 10,000 characters to a regular string. You'd like to compare this method with TestStringBuilder, but adding just 10,000 characters to a StringBuilder takes too little time to be measured by means of a Stopwatch object. The solution is to have the TestStringBuilder method perform one million iterations and to set the NormalizeFactor property to 100 so that the benchmark code knows that the result time must be divided by 100.

The structure of the benchmark tool is also simple. It scans the assembly passed to it on the command line, looking for methods flagged with the Benchmark attribute. (Only static methods with zero arguments are considered.) It sorts all benchmarks by their Group property, and then invokes each method in each group, sorts the results, and displays a report.

```
sealed public class App
{
   public static void Main(string[] args)
   {
      // Parse the assembly whose path is passed as an argument.
      Assembly asm = Assembly.LoadFile(args[0]);
      // Search all methods marked with the Benchmark attribute, sorted by their Group.
      SortedDictionary<BenchmarkAttribute, MethodInfo> attrList =
         new SortedDictionary<BenchmarkAttribute, MethodInfo>();

      // Iterate over all public and private static methods of all types.
      foreach ( Type type in asm.GetTypes() )
      {
         foreach ( MethodInfo mi in type.GetMethods(BindingFlags.Public |
            BindingFlags.NonPublic | BindingFlags.Static) )
         {
            // Extract the attribute associated with each member.
            BenchmarkAttribute attr = (BenchmarkAttribute)
               Attribute.GetCustomAttribute(mi, typeof(BenchmarkAttribute));
            // This must be a void method that takes no arguments.
            if ( attr != null && mi.GetParameters().Length == 0 )
            {
               // Benchmark name defaults to method name.
               if ( attr.Name.Length == 0 )
               {
                  attr.Name = mi.Name;
               }
               attrList.Add(attr, mi);
            }
         }
      }

      string lastGroup = null;
      SortedDictionary<long, BenchmarkAttribute> timingList =
         new SortedDictionary<long, BenchmarkAttribute>();
      // Display the report header.
      Console.WriteLine("{0,-20}{1,-30}{2,14}{3,12}", "Group", "Test", "Seconds", "Rate");
      Console.Write(new string('-', 78));
```

```
      // Run all tests, sorted by their group.
      foreach ( KeyValuePair<BenchmarkAttribute, MethodInfo> kvp in attrList )
      {
         BenchmarkAttribute attr = kvp.Key;
         MethodInfo mi = kvp.Value;

         // Show a blank line if this is a new group.
         if ( attr.Group != lastGroup )
         {
            DisplayGroupResult(timingList);
            Console.WriteLine();
            lastGroup = attr.Group;
            timingList.Clear();
         }

         // Invoke the method.
         Stopwatch sw = Stopwatch.StartNew();
         mi.Invoke(null, null);
         sw.Stop();
         // Remember total timing, taking normalization factor into account.
         timingList.Add(Convert.ToInt64(sw.ElapsedTicks / attr.NormalizationFactor), attr);
      }
      // Display result of the last group.
      DisplayGroupResult(timingList);
   }

   // Helper routine that displays all the timings in a group
   static void DisplayGroupResult(SortedDictionary<long, BenchmarkAttribute> timingList)
   {
      if ( timingList.Count == 0 )
      {
         return;
      }

      long bestTime = -1;
      foreach ( KeyValuePair<long, BenchmarkAttribute> kvp in timingList )
      {
         // The first timing in the sorted collection is also the best timing.
         if ( bestTime < 0 )
         {
            bestTime = kvp.Key;
         }
         double rate = System.Convert.ToDouble(kvp.Key) / bestTime;
         Console.WriteLine("{0,-20}{1,-30}{2,14:N4}{3,12:N2}", kvp.Value.Group,
            kvp.Value.Name, System.Convert.ToDouble(kvp.Key) / 100000000, rate);
      }
   }
}
```

Groups in the final report are sorted alphabetically, whereas benchmarks in each group are sorted by their timings, after normalizing them if necessary. (Faster benchmarks come first.) The rightmost column, Rate, compares each timing with the best time in its group. For example,

this is the report produced on my computer by running the tool against the five benchmarks listed previously:

```
Group              Test                              Seconds        Rate
-------------------------------------------------------------------------
Concatenation      TestStringBuilder                  0.1435        1.00
Concatenation      TestString                        17.8863      124.66

Division           Double division                    3.8228        1.00
Division           Integer division                   5.8769        1.54
Division           Long division                      6.1644        1.61
```

You can improve this first version of the benchmark tool in many ways. For example, you might save benchmark reports and automatically compare timings against a previous run of the tool to check whether some key methods of your application are performing slower than they did previously.

Writing Plug-ins for Windows Forms Applications

Most business applications must be extensible and customizable to meet specific requirements of different customers. For example, one customer might require additional fields on a data entry form; another customer might want to delete or add menu commands, and so forth. Typically, developers respond to these requirements by interspersing tons of if and switch statements in code, but this approach is clearly unsatisfactory and can quickly lead to maintenance insanity.

Even if customization isn't a requirement, you might want to build your applications from the ground up with extensibility in mind so that you can later release new modules that fit into the main application without forcing users to reinstall a completely new version. If your application can be extended and customized without having to recompile its source code, expert users might create their own modules, without putting your support team under pressure.

The first and most delicate step in building extensible and customizable applications is designing a plug-in infrastructure. In this section, I show you how to implement a powerful and flexible mechanism that enables you (or your users) to create plug-ins that are notified when a form in the main application is created so that each plug-in can add its own controls and menu commands or even replace the original form with a completely different form. Not surprisingly, the technique I am about to illustrate is based on custom attributes. It is a simplified version of the extension mechanism we use for Code Architects' Windows Forms applications that need to be extensible.

The PluginLibrary Project

Create a new blank solution and add a Class Library project named PluginLibrary. This project contains only two classes: FormExtenderAttribute and FormExtenderManager.

The FormExtenderAttribute Type

For simplicity's sake, the listing for the FormExtender custom attribute class uses fields instead of properties. (The companion code for this book uses property procedures that validate incoming values, though.)

```
[AttributeUsage(AttributeTargets.Class)]
public class FormExtenderAttribute : Attribute
{
   public readonly string FormName;
   public readonly bool Replace;
   public bool IncludeInherited;

   // Constructors
   public FormExtenderAttribute(string formName) : this(formName, false)
   {}

   public FormExtenderAttribute(string formName, bool replace)
   {
      this.FormName = formName;
      this.Replace = replace;
   }
}
```

You can use the FormExtender attribute with two different kinds of classes. You can use it either with a plug-in (nonvisual) class that must be notified when a form in the main application has been created or with a plug-in form class that replaces a form in the main application. In the former case, you set the Replace argument to false (or omit it); in the latter case, you assign it the true value. In both cases, *FormName* is the complete name of a form in the main application that is being instantiated; if IncludeInherited is true, the plug-in works with both the specified form class and all the forms that inherit from that form class. These three properties enable you to create several plug-in flavors:

```
[FormExtender("MainApplication.MainForm")]
public class MainForm_Extender
{
   public MainForm_Extender(MainForm frm)
   {
      // This plug-in class is instantiated when the MainForm is loaded.
      …
   }
}

[FormExtender("System.Windows.Forms.Form", IncludeInherited = true)]
public class GenericForm_Extender
{
   public GenericForm_Extender(Form frm)
   {
      // This plug-in class is instantiated when any form in the main application is
      // loaded because it specifies a generic form and sets IncludeInherited to true.
      …
   }
}
```

```
[FormExtender("MainApplication.MainForm", true)]
public class OptionForm_Replacement :
System.Windows.Forms.Form
{
   // This form replaces the main application's OptionsForm class.
   …
}
```

Plug-in classes that don't replace a form (Replace argument is false) are instantiated immediately after the form in the main application and a reference to the form is passed to their constructor so that the plug-in has an opportunity to add one or more user interface elements. (Read on for examples.)

The FormExtenderManager Type

The FormExtenderManager class exposes three important static methods. The InitializePlugins method parses all the DLL assemblies in a directory that you specify, looks for types marked with the FormExtender attribute, and stores information about these types in a generic Dictionary for later retrieval. If the main application doesn't call this method explicitly, it will be executed anyway when any of the next two methods is invoked. (In this case, you can't specify a path and InitializePlugins automatically looks for plug-ins in the main application's folder.)

```
public static class FormExtenderManager
{
   // All the FormExtenders known to this manager
   private static Dictionary<string, FormExtenderInfo> extenders;

   // Initialize the list of form extenders when the type is referenced.
   public static void InitializePlugIns()
   {
      string dirName = Path.GetDirectoryName(Application.ExecutablePath);
      InitializePlugIns(dirName);
   }

   // Create the list of form extenders.
   public static void InitializePlugIns(string dirName)
   {
      extenders = new Dictionary<string, FormExtenderInfo>();
      // Visit all the DLLs in this directory.
      foreach ( string dllName in Directory.GetFiles(dirName, "*.dll") )
      {
         try
         {
            // Attempt to load this assembly.
            Assembly asm = Assembly.LoadFile(dllName);
            ParseAssembly(asm);
         }
         catch ( Exception ex )
         {
            // Ignore DLLs that aren't assemblies.
         }
      }
   }
```

The extenders Dictionary contains instances of the FormExtenderInfo nested type. This dictionary is built inside the ParseAssembly private method:

```
private static void ParseAssembly(Assembly asm)
{
   Type attrType = typeof(FormExtenderAttribute);
   // Check all the types in the assembly.
   foreach (Type type in asm.GetTypes() )
   {
      // Retrieve the FormExtenderAttribute.
      FormExtenderAttribute attr = (FormExtenderAttribute)
         Attribute.GetCustomAttribute(type, attrType, false) ;
      if ( attr != null )
      {
         // Add to the dictionary.
         FormExtenderInfo info = new FormExtenderInfo();
         info.FormName = attr.FormName;
         info.Replace = attr.Replace;
         info.IncludeInherited = attr.IncludeInherited;
         info.Type = type;
         extenders.Add(info.FormName, info);
      }
   }
}

// A nested class used to hold information on form extenders
private class FormExtenderInfo
{
   public string FormName;
   public bool Replace;
   public bool IncludeInherited;
   public Type Type;
}
```

The Create method takes a form type as a generic parameter, creates an instance of that form type, and then notifies all plug-ins that the form has been created. However, if a plug-in class has a FormExtender attribute whose FormName property matches the form's name and whose Replace property is set to true, the Create method creates an instance of the plug-in form rather than the original form.

```
public static T Create<T>() where T : Form
{
   // The private CreateForm method does the real job.
   return (T) CreateForm<T>() ;
}

private static Form CreateForm<T>() where T : Form
{
   // Initialize plug-ins if necessary.
   if ( extenders == null )
   {
      InitializePlugIns();
   }
```

```
    Type formType = typeof(T);
    string formName = formType.FullName;
    bool mustNotify = true;

    // Check whether this form appears in the dictionary.
    if ( extenders.ContainsKey(formName) )
    {
       FormExtenderInfo info = extenders[formName];
       // If form must be replaced, instantiate the corresponding type instead.
       if ( info.Replace )
       {
          formType = info.Type;
          mustNotify = false;
       }
    }

    // Create the form.
    Form frm = (Form) Activator.CreateInstance(formType, true) ;
    // Notify all extenders, if necessary, and then return form to the caller.
    if ( mustNotify )
    {
       NotifyFormCreation(frm);
    }
    return frm;
}
```

NotifyFormCreation takes a form type as an argument and instantiates all the plug-in classes that have been declared as extensions for the specified form.

```
public static void NotifyFormCreation(Form frm)
{
   // Initialize plug-ins if necessary.
   if ( extenders == null )
   {
      InitializePlugIns();
   }

   string formName = frm.GetType().FullName;
   Type extenderType = null;

   // Check whether this form appears in the dictionary.
   if ( extenders.ContainsKey(formName) )
   {
      // Don't notify forms that would replace the original one.
      if ( ! extenders[formName].Replace )
      {
         extenderType = extenders[formName].Type;
      }
   }
   else
   {
      // Check whether there is an extender that applies to one of the base classes.
      Type type = frm.GetType();
```

```
            do
            {
               type = type.BaseType;
               if ( extenders.ContainsKey(type.FullName) )
               {
                  FormExtenderInfo info = extenders[type.FullName];
                  // We can use this extender only if the IncludeInherited property is true.
                  if ( info.IncludeInherited && !info.Replace )
                  {
                     extenderType = info.Type;
                     break;
                  }
               }
               // Continue until we get to the System.Windows.Forms.Form base class.
            }
            while ( type != typeof(Form) );
         }

         if ( extenderType != null )
         {
            // Invoke the extender's constructor, passing the form instance as an argument.
            // (This statement fails if such a constructor is missing.)
            try
            {
               object[] args = { frm };
               Activator.CreateInstance(extenderType, args);
            }
            catch ( Exception ex )
            {
               // This should never happen.
               Debug.WriteLine("Constructor not found for type " + extenderType.FullName);
            }
         }
      }
}   // End of FormExtenderManager class
```

Notice that this version of the PluginLibrary supports only one plug-in for each form. A more complete implementation would manage a list of all the plug-ins that want to be notified when a given form is created. (Clearly, only one plug-in class can replace a form, though.)

The MainApplication and MainApplicationStartup Projects

Applications that are extensible through plug-ins must be built by following a couple of criteria. First, the application is actually split into two assemblies: a DLL that contains the bulk of the application, including all its forms, and a simple EXE that displays the application's main form (and therefore indirectly bootstraps the DLL). It is necessary to have all the forms in a separate DLL because plug-ins might need to inherit from one of the forms in the main application. (You can reference a type in another EXE assembly, but you can't inherit a form from a form defined in an EXE.) The second criterion requires that you instantiate a form by means the FormExtenderManager.Create method rather than by using the New keyword, as I explain shortly.

The Startup Project

Let's create the startup assembly first. Create a Windows Forms application containing this code:

```
// (This code assumes that you've imported the PluginLibrary namespace.)
public static void Main()
{
   Application.EnableVisualStyles();
   Application.Run(PluginLibrary.FormExtenderManager.Create<MainApplication.MainForm>());
}
```

The current project requires a reference to both the PluginLibrary project and the MainApplication project, which I illustrate in the following section.

The MainApplication Project

Next, create a Class Library project named MainApplication, and add a reference to the PluginLibrary project and to any other .NET assembly you use, including the System.Windows.Forms.dll assembly. Also, the code that follows assumes that a using statement for the PluginLibrary namespace has been added at the top of each source file.

Start adding forms to this project and the code that uses these forms as you'd do normally, with one exception: use the FormExtenderManager.Create method rather than a plain New keyword to instantiate a form. For example, the MainForm class can have a menu, as shown in Figure 10-1. The Sample Form command on the View menu should display a form named SampleForm, and thus you should display such a form using the following code:

```
Form frm = FormExtenderManager.Create<SampleForm>();
frm.Show();
```

Figure 10-1 The application's main form

Alternatively, if you are sure that a form is never replaced by a plug-in form, you can create the form normally and then use the NotifyFormCreation method to let plug-ins know that the form has been created. You can call this method from inside the overridden OnLoad method:

```
protected override void OnLoad(EventArgs e)
{
   base.OnLoad(e);
   FormExtenderManager.NotifyFormCreation(this);
}
```

Creating Inheritable Forms

If the main application accesses one or more public members of a form, you can't simply replace the original form with another form defined in the plug-in DLL. For example, notice how the following code accesses the Total public property of the CalculatorForm:

```
CalculatorForm frmCalc = PluginLibrary.FormExtenderManager.Create<CalculatorForm>();
if ( frmCalc.ShowDialog() == System.Windows.Forms.DialogResult.OK )
{
   // Read the Total public property of the CalculatorForm type.
   MessageBox.Show(frmCalc.Total.ToString(), "Total", MessageBoxButtons.OK,
      MessageBoxIcon.Information);
}
```

The only way for a plug-in to replace the CalculatorForm type with a different form without making the previous code fail is by having the plug-in form inherit from CalculatorForm. (Because of this requirement you had to define all the application's forms in a DLL rather than in a plain EXE.) If you want to allow plug-ins to do the replacement in a simple way, you should create the CalculatorForm class with inheritance in mind. Basically, this means that you have to adhere to the following guidelines:

- In the Properties window, set the Modifiers property to Protected for all the controls on the form. (Tip: select all controls by pressing Ctrl+A and change the Modifiers property for all of them in one operation.)
- Don't put any "interesting" code inside event handlers; instead, place this code inside public or protected methods marked as virtual and call these methods from inside event handlers.

For example, the CalculatorForm must update all TextBox controls on the right when the contents of any field on the left changes. (See Figure 10-2.) Here's the correct way to update these calculated fields:

```
public partial class CalculatorForm
{
   public decimal Total;

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

```
   private void ValueChanged(System.Object sender, System.EventArgs e)
   {
      CalculateTotal();
   }

   protected virtual void CalculateTotal()
   {
      try
      {
         int units = int.Parse(txtUnits.Text);
         decimal unitPrice = decimal.Parse(txtUnitPrice.Text);
         decimal percentTax = decimal.Parse(txtPercentTax.Text);
         decimal total = units * unitPrice;
         decimal tax = total * percentTax / 100;
         decimal grandTotal = total + tax;

         txtTotal.Text = total.ToString("N2");
         txtTax.Text = tax.ToString("N2");
         txtGrandTotal.Text = grandTotal.ToString("N2");
      }
      catch ( Exception ex )
      {
         // Clear result fields in case of exceptions.
         txtTotal.Text = "";
         txtTax.Text = "";
         txtGrandTotal.Text = "";
      }
   }
}
```

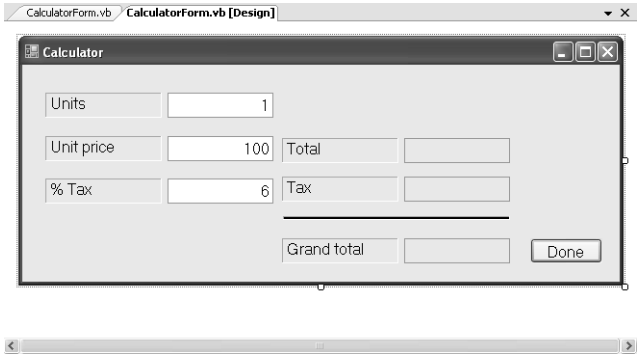

Figure 10-2 The CalculatorForm, which lets you perform simple calculations

Before proceeding, make MainApplicationStartup the startup project in the solution and run the application. You haven't defined any plug-in so far, and thus the PluginLibrary should do absolutely nothing.

The SamplePlugin Project

You're now ready to create your first plug-in. Add a new Class Library project, name it Sample-Plugin, and add a reference to the PluginLibrary and the MainApplication projects. The sample

plug-in project contains three types: SampleForm_Replacement is a form that replaces the SampleForm form; MainForm_Extender adds a menu command to the application's main form; CalculatorForm is a form that replaces the CalculatorForm class.

Replacing a Form with a Different Form

The first class is very simple indeed. Just create a form named SampleForm_Replacement, mark it with the FormExtender attribute, and add controls as you'd do in a regular form:

```
[PluginLibrary.FormExtender("MainApplication.SampleForm", true)]
public class SampleForm_Replacement
{
   …
}
```

In this case, the replacement form doesn't need to have any relation to the original form because the main application never references a method, a field, or control from outside the form class itself.

Extending a Form with User Interface Elements

The MainForm_Extender class extends the MainForm class without replacing it; thus, you must flag the MainForm_Extender class with a FormExtender attribute whose Replace property is false. The plug-in infrastructure will pass a MainForm instance to this class's constructor when the main form is loaded so that code in MainForm_Extender can add new elements to the form's Controls collection or, as in this case, to the DropDownItems collection of a StripMenu component:

```
[PluginLibrary.FormExtender("MainApplication.MainForm")]
public class MainForm_Extender
{
   public MainForm_Extender(MainApplication.MainForm frm)
   {
      // Add an entry to the Plugins menu.
      ToolStripMenuItem item = new ToolStripMenuItem("Show Date", null, MenuClick);
      frm.mnuPlugins.DropDownItems.Add(item);
   }

   private void MenuClick(object sender, EventArgs e)
   {
      MessageBox.Show(DateTime.Now.ToString(), "Current Date/Time",
         MessageBoxButtons.OK, MessageBoxIcon.Information);
   }
}
```

Notice that the previous code assumes that the mnuPlugins ToolStripMenu object on the application's MainForm is declared as public; otherwise, the code in the plug-in DLL can't add new elements to it.

Replacing a Form with an Inherited Form

The CalculatorForm_Replacement form both inherits from and replaces the main application's CalculatorForm type. To create this form, select the Add New Item command from the Project menu, select the Inherited Form template, click the Add button, select the CalculatorForm element inside the Inheritance Picker dialog box, and then click OK.

Next, make the form taller and move the two bottommost rows of fields down to make room for new controls that let you define a discount percentage. (See Figure 10-3.) Finally, add this code to make the form perform as intended:

```
[PluginLibrary.FormExtender("MainApplication.CalculatorForm", true)]
public partial class CalculatorForm_Replacement
{
   public CalculatorForm_Replacement()
   {
      InitializeComponent();
   }

   private void txtPercentDiscount_TextChanged(System.Object sender, System.EventArgs e)
   {
      CalculateTotal();
   }


   protected override void CalculateTotal()
   {
      if ( !this.Visible )
      {
         return;
      }

      try
      {
         int units = int.Parse(txtUnits.Text);
         decimal unitPrice = decimal.Parse(txtUnitPrice.Text);
         decimal percentTax = decimal.Parse(txtPercentTax.Text);
         decimal percentDiscount = decimal.Parse(txtPercentDiscount.Text);

         decimal total = units * unitPrice;
         decimal discount = total * percentDiscount / 100;
         decimal discountedTotal = total - discount;
         decimal tax = discountedTotal * percentTax / 100;
         decimal grandTotal = discountedTotal + tax;

         txtTotal.Text = total.ToString("N2");
         txtDiscount.Text = discount.ToString("N2");
         txtDiscountedTotal.Text = discountedTotal.ToString("N2");
         txtTax.Text = tax.ToString("N2");
         txtGrandTotal.Text = grandTotal.ToString("N2");
      }
      catch ( Exception ex )
      {
         // Clear result fields in case of exceptions.
         txtTotal.Text = "";
```

```
            txtDiscount.Text = "";
            txtDiscountedTotal.Text = "";
            txtTax.Text = "";
            txtGrandTotal.Text = "";
        }
    }
}
```

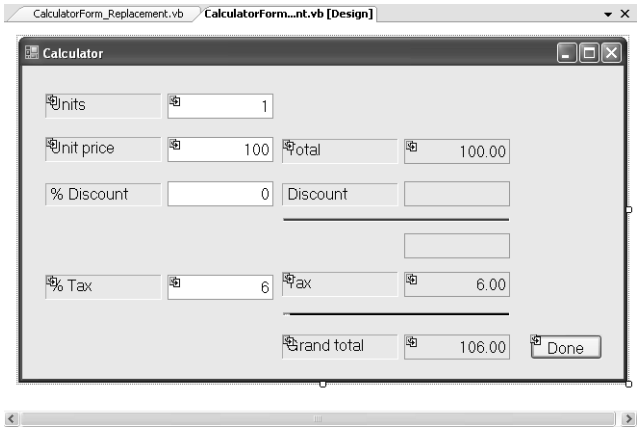

Figure 10-3 The new SamplePlugin project, which extends the CalculatorForm to account for a discount percentage

Notice that you need to handle the TextChange event only for the txtPercentDiscount control because the original CalculatorForm class in the main application takes care of the other input controls. When the original form invokes the CalculateTotal method, the overridden version in CalculatorForm_Replacement runs and you have an opportunity to take a discount into account.

Compiling and Testing the SamplePlugin Project

By default, The FormExtenderManager.InitializePlugins method looks into the main application's folder for plug-in DLLs. In our example, this folder is the folder where the MainApplicationStartup EXE file is located; therefore, you should manually copy the SamplePlugin.dll file into this folder whenever you recompile the SamplePlugin project. Alternatively, you can define a postcompilation build step that automates the copy operation or, even better, modify the Output Path setting (on the Build page of the project's designer) so that the SamplePlugin.dll file is compiled right in the main application's folder.

Another simple solution to this minor issue is to have the MainApplicationStartup project include a reference to the SamplePlugin project. The code in the main application's project never references the plug-in project, but this reference forces Visual Studio to copy the plug-in DLL into the application's folder whenever you recompile the solution.

Once all the executables are compiled and stored in the correct folders, you can run the application, check that the new user interface elements have been created, and check that they react as intended.

One final note: the PluginLibrary enables you to extend or replace any form defined in the main application, but the concepts it relies on can be applied to any kind of class, not just forms. In the last section of this chapter, you'll learn more about using custom attributes to affect the way objects are instantiated and used in your applications.

A Framework for *n*-Tiered Applications

The last example in this chapter is also the most complex code sample in the entire book. It is a completely functional, though simplified, framework that promotes the creation of data-centric applications that can be expanded, modified, and customized at will. In this section, I refer to this framework as the CAP framework because it's a (very) stripped-down version of Code Architects Platform, a product that my company has built and refined over the years and that is the heart of many real-world data-centric applications used in Italian government agencies that serve tens of thousands of clients simultaneously. (Contact me if you need information about the real thing.)

An *n-tiered application* is an application that makes a clear distinction between objects that generate the user interface, objects that represent real-world entities (the *business objects*), and objects that take care of reading and writing data on a database or another persistent medium (the *data objects*). All the data objects used in an application make up the Data Access Layer, or DAL. Keeping the three kinds of objects completely distinct makes programming more complex, but it offers an unparalleled degree of flexibility. For example, if your *n*-tiered application has a DAL that works with Oracle, you might replace it with a DAL that is tailored for Microsoft SQL Server. If the user interface, business, and data layers are completely distinct, this change doesn't affect either the user interface or the business tier.

Having distinct data and business tiers offers additional advantages. For example, in some cases you can change the structure of the database with minimal or no impact on the user interface. Even better, you can deploy the data objects on a remote computer, where they can interact with the database in a more efficient way. (In this case, the data objects must have a way of communicating with the business tier or the user interface tier by means of a remoting technique supported by .NET, such as Web Services, serviced components, or .NET remoting.) If the *n*-tiered application is designed correctly, you can change the deployment configuration of individual data and business objects to match a specific network configuration, again with minimal impact on other tiers.

The example I illustrate in this section is a simplified *n*-tiered application, which has only the user interface tier (a Windows Forms client) and the data tier. For simplicity's sake, no business tier is included. Also, in an attempt to make the code as concise as possible, DAL objects move data from the database to the application and back by means of typed DataSet objects. Some Service Oriented Architecture (SOA) purists might dislike this design choice, but—again—I wanted to simplify the code to make the important details stand out. If you don't like using DataSets in this fashion, you can modify the CAP framework to use custom collections. This replacement is relatively simple, thanks to generic collections. (See Chapter 4, "Generics.")

Despite its simplicity, the CAP framework supports a couple of advanced features. First, the client application never instantiates a data object directly; instead, it uses a factory class named DataObjectFactory, which returns a data object that is specific for a given configuration and the specified database table. This approach enables you to use different data objects for different configurations transparently—such as "SqlServer," "Oracle," "Access," "Demo," and so forth—without changing the code in the client application. Different configurations can correspond to different databases, different application versions, different network topologies, and so on.

> **Note** In this chapter, I use the term *database table* to indicate the entity that a data object reads from and writes to. A data object can interact with sources other than database tables—for example, text or XML files, or a Web Service—therefore, you shouldn't take this term literally in this context. Think of it more as an abstract "data entity" than a physical database table.

Second, and more interesting, you can associate one or more companion objects with each data object. A *data object companion* is an object that attaches itself to a data object and can take part in commands that read or write data to the database. The job of a companion object can be as simple as logging all database operations to a trace file or as complex as filtering the data being read from or written to the database. Near the end of this chapter, I show you how to build a companion object that extends a data object with caching abilities.

The DataObjectLibrary Project

All the interfaces, custom attributes, and helper types that make up the CAP framework are gathered in the DataObjectLibrary.dll assembly. (See Figure 10-4.) This assembly must be referenced both by the client application and the DLLs that contain data objects and companion objects. In the remainder of this chapter I assume that a using statement for the DataObjectLibrary namespace has been added at the file or project level where necessary.

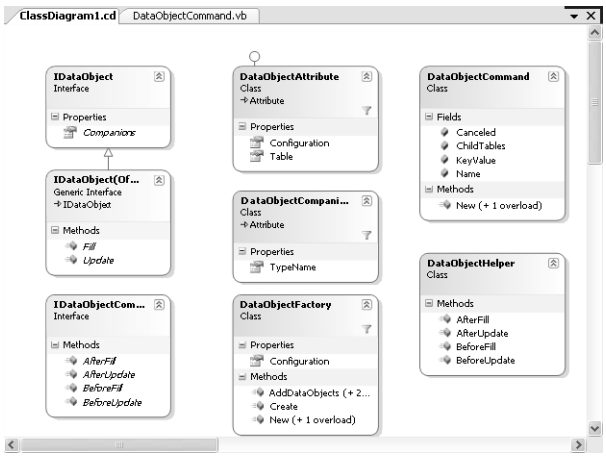

Figure 10-4 All the types in the DataObjectLibrary project, the core of the CAP framework

Interfaces

The CAP framework defines three interfaces. The IDataObject and IDataObject<T> interfaces define the methods that a data object must expose, whereas the IDataObjectCompanion interface defines the data commands a companion object can take part in:

```
public interface IDataObject
{
   List<IDataObjectCompanion> Companions { get; }
}

public interface IDataObject<TDataSet> : IDataObject where TDataSet : DataSet
{
   TDataSet Fill(TDataSet ds, DataObjectCommand command);
   TDataSet Update(TDataSet ds, DataObjectCommand command);
}

public interface IDataObjectCompanion
{
   void BeforeFill(IDataObject obj, DataSet ds, DataObjectCommand command);
   void BeforeUpdate(IDataObject obj, DataSet ds, DataObjectCommand command);
   void AfterFill(IDataObject obj, DataSet ds, DataObjectCommand command);
   void AfterUpdate(IDataObject obj, DataSet ds, DataObjectCommand command);
}
```

Notice that the IDataObject<T> interface derives from the nongeneric IDataObject interface and defines data objects that work with a strong-typed DataSet object. The role of these interfaces will be clear shortly.

Custom Attributes

The DataObjectLibrary project contains two custom attributes: DataObjectAttribute is used to mark data objects, whereas DataObjectCompanionAttribute is used to mark companion objects. Here's an abridged definition of these attributes, where I have replaced properties with fields to keep the listing as concise as possible. (See the companion code for the complete definition of these classes.)

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct, Inherited = true)]
public class DataObjectAttribute : Attribute, IComparable<DataObjectAttribute>
{
   public DataObjectAttribute(string configuration, string table)
   {
      this.Configuration = configuration;
      this.Table = table;
   }

   // The configuration in which the data object is valid
   public string Configuration;
   // The database table this data object applies to
   public string Table;
```

```
   // Support for the IComparable<DataObjectAttribute> interface
   public int CompareTo(DataObjectAttribute other)
   {
      int res = this.Configuration.CompareTo(other.Configuration);
      if ( res == 0 )
      {
         res = this.Table.CompareTo(other.Table);
      }

      return res;
   }
}

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class DataObjectCompanionAttribute : Attribute
{
   public DataObjectCompanionAttribute(string typeName)
   {
      if ( typeName == null )
      {
         typeName = "";
      }
      this.TypeName = typeName;
   }

   // The type of the data object. Use "" for all data objects.
   public string TypeName;
}
```

The DataObjectCompanionAttribute can be applied multiple times to the same class: in other words, a companion object can serve multiple data objects.

The DataObjectFactory Type

The DataObjectFactory type is a key component of the CAP framework. Client applications use this type to instantiate the data object that corresponds to a given configuration and a given database table.

```
public class DataObjectFactory
{
   // This dictionary holds all data objects found so far.
   public readonly Dictionary<string, Type> DataObjects = new Dictionary<string, Type>();
   // This dictionary holds all validator objects for a given data object.
   public readonly Dictionary<string, List<Type>> DataCompanions = new Dictionary<string,
      List<Type>>();

   // Constructors
   public DataObjectFactory(string configuration, Assembly assembly)
   {
      this.Configuration = configuration;
      AddDataObjects(assembly);
   }
```

```
public DataObjectFactory(string configuration, string dataObjectsDirectory)
{
   this.Configuration = configuration;
   AddDataObjects(dataObjectsDirectory);
}

// The Configuration string
public readonly string Configuration;
```

The AddDataObjects method is where the DataObjectFactory type parses an assembly and records all the data objects and the companion objects it finds:

```
// Add all the DataObjects of this assembly to the list.
public void AddDataObjects(Assembly assembly)
{
   foreach ( Type type in assembly.GetTypes() )
   {
      // Look for types marked with the DataObject attribute.
      DataObjectAttribute doAttr = (DataObjectAttribute)
         Attribute.GetCustomAttribute(type, typeof(DataObjectAttribute), false);
      // Ensure that type implements IDataObject and that it's suitable
      // for this configuration (or current configuration is null).
      if ( doAttr != null && typeof(IDataObject).IsAssignableFrom(type) &&
         ( string.Compare(this.Configuration, doAttr.Configuration, true) == 0
         || this.Configuration.Length == 0 ) )
      {
         // Add to the dictionary only if not there already.
         if ( !DataObjects.ContainsKey(doAttr.Table.ToLower()) )
         {
            DataObjects.Add(doAttr.Table.ToLower(), type);
         }
      }

      // Look for types marked with the DataObjectValidator attribute.
      // (This attribute allows multiple instances.)
      DataObjectCompanionAttribute[] coAttrs =
         (DataObjectCompanionAttribute[])Attribute.GetCustomAttributes(type,
         typeof(DataObjectCompanionAttribute), false);
      if ( coAttrs != null && coAttrs.Length > 0 )
      {
         // Iterate over each instance of the attribute.
         foreach ( DataObjectCompanionAttribute coAttr in coAttrs )
         {
            // Create an item in the DataValidators dictionary, if necessary.
            if ( !this.DataCompanions.ContainsKey(coAttr.TypeName) )
            {
               this.DataCompanions.Add(coAttr.TypeName, new List<Type>());
            }
            // Add this validator to the list.
            this.DataCompanions[coAttr.TypeName].Add(type);
         }
      }
   }
}
```

The Create method is invoked by client applications when they need a data object for a specific database table:

```
   // Create a data object for a given database/table.
   public IDataObject Create(string table)
   {
      if ( !this.DataObjects.ContainsKey(table.ToLower()) )
      {
         throw new ArgumentException("Table not found: " + table);
      }
      // Create an instance of the corresponding data object.
      Type type = this.DataObjects[table.ToLower()];
      IDataObject dataObj = (IDataObject) Activator.CreateInstance(type, true);

      // Add all data companions associated with this specific data object.
      if ( this.DataCompanions.ContainsKey(type.FullName) )
      {
         // Create an instance of each companion associated with this data object,
         // and add the instance to the Companions collection.
         foreach ( Type coType in this.DataCompanions[type.FullName] )
         {
            IDataObjectCompanion compObj = (IDataObjectCompanion)
               Activator.CreateInstance(coType, true);
            dataObj.Companions.Add(compObj);
         }
      }

      // Add all the generic companions (associated with all data objects).
      if ( this.DataCompanions.ContainsKey("") )
      {
         // Create an instance of each companion associated with this data object,
         // and add it to the Companions collection.
         foreach ( Type coType in this.DataCompanions[""] )
         {
            IDataObjectCompanion compObj = (IDataObjectCompanion)
               Activator.CreateInstance(coType, true);
            dataObj.Companions.Add(compObj);
         }
      }
      return dataObj;
   }
}       // End of DataObjectFactory type
```

In practice, a client application must create an instance of the DataObjectFactory type for a given configuration and then invoke the Create method when a new data object is needed. The Create method instantiates the data object associated with the configuration/table combination, creates all the data companions that should be associated with that data object, and finally returns the data object to the client:

```
// (In a client application…)
// Prepare a list of all data objects in the application's folder.
DataObjectFactory factory = new DataObjectFactory("Access", Application.StartupPath);
```

```
void PerformQuery()
{
   // Create a data object for the Customers database table that
   // knows how to deal with NWINDDataSet objects.
   IDataObject<NWINDDataSet> doCustomers =
      (IDataObject<NWINDDataSet>) factory.Create("Customers");
   …
}
```

If you have your data objects spread in more than one DLL, you can call the AddDataObjects method multiple times, once for each DLL.

The DataObjectCommand Type

The DataObjectCommand type is a repository for storing information about an operation that a data object must perform. It contains only fields and one constructor.

```
public class DataObjectCommand
{
   // The name of this command
   public readonly string Name;
   // The list of child tables to be included in the fill/update command
   public readonly List<string> ChildTables = new List<string>();
   // The value of key column. (This version supports only one key column.)
   public readonly object KeyValue;
   // True if this command has been canceled (by a data companion)
   public bool Canceled;

   public DataObjectCommand(string name, object keyValue, params string[] childTables)
   {
      this.Name = name;
      this.KeyValue = keyValue;
      if ( childTables != null )
      {
         this.ChildTables.AddRange(childTables);
      }
   }
}
```

The DataObjectCommand type adds a lot of flexibility to the CAP framework. Thanks to this type, data objects simply need to expose two methods, Fill and Update, which can execute virtually all the operations you typically perform on a database. Each method can behave differently, depending on the values you pass through the DataObjectCommand argument. For example, the Fill method can read an entire table or just one row, depending on whether you've assigned a value to the KeyValue property, and might return the rows in child tables as well, if you have passed the name of one or more child tables.

```
// (In a client application…)
// Create a data object for the Customers database table that
// knows how to deal with NWINDDataSet objects.
   IDataObject<NWINDDataSet> doCustomers =
      (IDataObject<NWINDDataSet>) factory.Create("Customers");
```

```
// Create a command that reads the entire Customers table (no key specified)
// and also retrieves the Orders child table.
DataObjectCommand command = new DataObjectCommand("GetCustomers", null, "Orders");
// Use the data object to fill a local DataSet.
doCustomers.Fill(this.NwindDataSet1, command);
```

A crucial feature of the CAP framework is that all companion objects can inspect the DataObjectCommand that has been passed to the Fill or Update method, both before and after the actual data operation is performed. In the "before" phase, a companion object can set the command's Cancel property to true and indirectly prevent the command from being carried out. Later in this chapter, I show you how you can use this feature to implement a caching mechanism: in that case, the companion object fills the DataSet with data taken from the cache, and then sets the Cancel property to false to notify the data object that it doesn't have to actually extract data from the database. A common operation you can perform either in the "before read" or in the "after read" phase is setting a filter on the data being read to ensure that each user can see only the data he or she is allowed to see. You often use the "before write" phase to validate all data one instant before it is written to the database.

The DataObjectHelper Type

The last type in the CAP framework is simply a container for static helper methods. You can use this helper object to reduce the amount of code inside individual data objects.

```
public static class DataObjectHelper
{
   public static bool BeforeFill(IDataObject obj, DataSet ds, DataObjectCommand command)
   {
      foreach ( IDataObjectCompanion companion in obj.Companions )
      {
         companion.BeforeFill(obj, ds, command);
      }
      return !command.Canceled;
   }

   public static bool BeforeUpdate(IDataObject obj, DataSet ds, DataObjectCommand command)
   {
      foreach ( IDataObjectCompanion companion in obj.Companions )
      {
         companion.BeforeUpdate(obj, ds, command);
      }
      return !command.Canceled;
   }

   public static bool AfterFill(IDataObject obj, DataSet ds, DataObjectCommand command)
   {
      foreach ( IDataObjectCompanion companion in obj.Companions )
      {
         companion.AfterFill(obj, ds, command);
      }
      return !command.Canceled;
   }
```

```
   public static bool AfterUpdate(IDataObject obj, DataSet ds, DataObjectCommand command)
   {
      foreach ( IDataObjectCompanion companion in obj.Companions )
      {
         companion.AfterUpdate(obj, ds, command);
      }
      return !command.Canceled;
   }
}
```

In practice, each method in this class notifies all the companions of a data object that a Before*Xxxx* or After*Xxxx* step has been reached, and it returns true if no companion object has canceled the command. Thanks to this helper class, the code in the data object can be simplified remarkably:

```
// (Inside a data object…)
public NWINDDataSet Fill(NWINDDataSet ds, DataObjectCommand command)
{
   // Notify all companion objects that the Fill command is about to be executed.
   if ( DataObjectHelper.BeforeFill(this, ds, command) )
   {
      // Execute the Fill command. (No companion object has canceled it.)
      …
   }
   // Notify all companion objects that the Fill command has been completed.
   DataObjectHelper.AfterFill(this, ds, command);
   return ds;
}

public NWINDDataSet Update(NWINDDataSet ds, DataObjectCommand command)
{
   // Notify all companion objects that the Update command is about to be executed.
   if ( DataObjectHelper.BeforeUpdate(this, ds, command) )
   {
      // Execute the Update command. (No companion object has canceled it.)
      …
   }
   // Notify all companion objects that the Update command has been completed.
   DataObjectHelper.AfterUpdate(this, ds, command);
   return ds;
}
```

The DataSets Project

The next step in building a data-centric application based on the CAP framework is creating a separate DLL project where you define one or more strong-typed DataSet types. Keeping your DataSets in a separate project is necessary because both the main application and the various DLLs that contain data objects and companion objects must have a reference to these shared DataSet types.

The demonstration solution contains an assembly named DataSets, which contains one DataSet type named NWINDDataSet, which is shaped after the NWIND.MDB Microsoft Access database. (I use this database because chances are that you already have it on your computer; if not, it comes with the companion code for this book.) I use Access only for the sake of simplicity: *n*-tiered applications typically work with enterprise-sized database engines such as SQL Server or Oracle.

Code in the companion project assumes that the NWIND.MDB file resides in the C:\ root directory, and thus you should move it there before running the samples. If you want to store the file in a different directory, you must change the value of the NWINDConnectionString setting in all the projects that make up the solution.

The whole point in using data objects is the ability to keep the code in the main application completely separate from the data access code so that you can easily migrate this sample application to use a different database engine. For this reason, you should look at the NWINDDataSet type only as a container for data, not as a sort of in-memory database that exactly mirrors the structure of a real database. As I noted previously, a data object doesn't necessarily interact with a database and can read from and write to other data stores, such as XML files.

You can create the NWINDDataSet type in Visual Studio 2005 very simply. Select the Add New Data Source command from the Data menu to open the Data Source Configuration Wizard. In the first step, select the Database option and click Next; in the second step, click the New Connection button to create a new OLE DB connection that points to the NWIND.MDB file. When you click Next, Visual Studio asks whether you want to include the .mdb file in the project folder: answer No because the NWINDDataSet type should have no direct relation to the database you're using, as I just emphasized. In the next step, accept that you want to save the connection string in the application's configuration file. (See Figure 10-5.) In the fourth and last steps, select which tables you want to include in the DataSet. In this demonstration, you can just select the Customers, Orders, and Order Details tables.

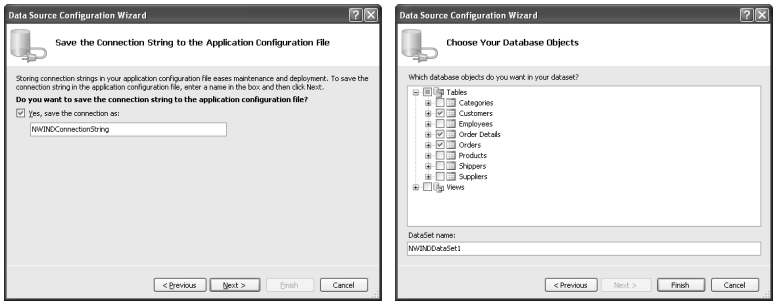

Figure 10-5 Two steps in the Data Source Configuration Wizard

When you click the Finish button, Visual Studio creates the file NWINDDataSet.xsd, which contains the schema of the new DataSet. You can double-click this file in the Solution Explorer to view the graphical representation of all the tables in the DataSet and their relations. (See Figure 10-6.) In Microsoft Visual Studio .NET 2003, you had to create table relationships by hand.

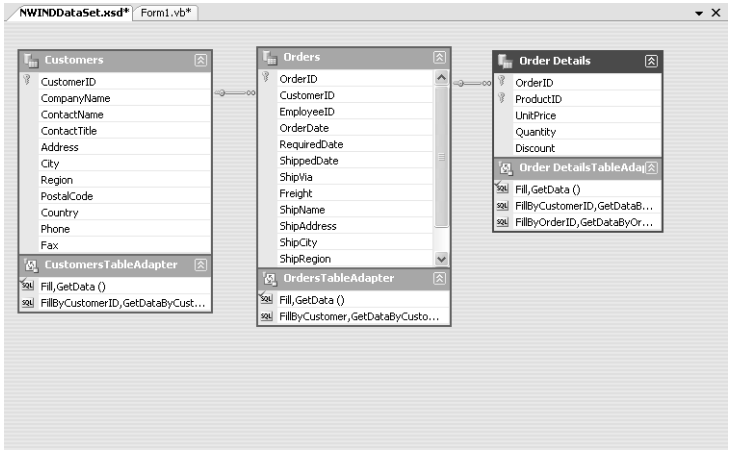

Figure 10-6 For each DataTable in the DataSet, you can define one or more custom commands, such as FillByCustomerID.

Each DataTable type in the DataSet has a companion TableAdapter object, which enables you to read data from and write data to that table by means of methods named Fill, GetData, and Update. In addition to these standard methods, you can define your own queries by right-clicking a table in the .xsd schema, pointing to Add, and selecting Query. For example, the sample code requires that you add a query named FillByCustomerID. In the first step of the TableAdapter Query Configuration Wizard, you decide whether you want to retrieve data using an SQL statement or a stored procedure; if you're using an Access database, only the first option is available. In the second step, you select which type of query you want to perform; for this demo, choose the Select Which Returns Rows option. In the next step, type the SQL statement that returns data, for example, a parameterized query that returns a single record:

```
SELECT CustomerID, CompanyName, ContactName, ContactTitle, Address, City,
   Region, PostalCode, Country, Phone, Fax FROM Customers WHERE CustomerID=?
```

(Use an argument name prefixed with the at sign (@) if you're accessing a SQL Server database.) In the fourth step, assign a name to the Fill*Xxxx* and GetData*Xxxx* commands you are creating (FillByCustomerID and GetDataByCustomerID in this specific example). Finally, click the Finish button to generate the custom TableAdapter commands. (See Figure 10-7.)

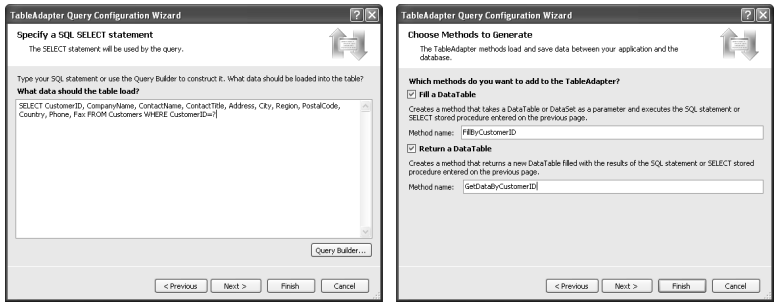

Figure 10-7 Two steps of the TableAdapter Query Configuration Wizard

Note Although I love the power of TableAdapters and their custom commands, I can't help but notice that the Microsoft decision to generate TableAdapter types in the same assembly as the DataSet type they refer to is a rather questionable design choice that makes TableAdapters less useful in enterprise-level *n*-tiered applications. In such applications, TableAdapter objects shouldn't be located in the same assembly where the DataSet is defined because this arrangement prevents each data object from using a TableAdapter object that is specific for a given database.

In this demonstration program, we leave the TableAdapters as nested classes of the NWINDDataSet type. In a real application, however, you should cut the TableAdapter code from that type and paste it into the same assembly where the data objects for that specific database are located. Of course, you'll also need to update all the using statements to ensure that all the TableAdapter commands continue to work correctly even if they now reference a DataSet in a different assembly.

The DemoClient Project

Even if you haven't defined any data objects so far, already you can create a client application that will use them. In fact, the client application doesn't need any direct reference to the assembly that contains your data objects. More precisely, the application should *not* have such a reference because clients must be completely independent of the actual data objects. If you don't create a dependency between the client and a specific data object, you can later replace a new set of DAL objects to match a different database or network configuration.

Create a Windows Forms project named DemoClient and ensure that it has a reference to the DataObjectLibrary and DataSets projects. Open the Data section of the Toolbox, drop a DataSet component onto the form's designer, specify DataSets.NWINDDataSet as the typed DataSet, and name it NwindDataSet1. Next, drop two BindingSource components onto the form, name them bsCustomers and bsCustomersOrders, and set their properties as follows:

```
bsCustomers.DataSource = NwindDataSet1
bsCustomers.DataMember = Customers
bsCustomersOrders.DataSource = bsCustomers
bsCustomersOrders.DataMember = CustomersOrders
```

Create a very simple user interface by dropping a couple of buttons and two DataGridView controls on the main form, as shown in Figure 10-8. Set the DataSource property of the topmost DataGridView equal to bsCustomers and the DataSource property of the bottom-most DataGridView equal to bsCustomersOrders. Finally, enter this code in the form's class:

```
// Look for all the data objects in the application's folder.
DataObjectFactory factory = new DataObjectFactory("Access", Application.StartupPath);

private void btnFill_Click(System.Object sender, EventArgs e)
{
   IDataObject<NWINDDataSet> doCustomers =
      (IDataObject<NWINDDataSet>) factory.Create("Customers");
   DataObjectCommand command = new DataObjectCommand("GetCustomers", null, "Orders");
   doCustomers.Fill(this.NwindDataSet1, command);
}

private void btnUpdate_Click(object sender, EventArgs e)
{
   IDataObject<NWINDDataSet> doCustomers =
      (IDataObject<NWINDDataSet>) factory.Create("Customers");
   DataObjectCommand command = new DataObjectCommand("UpdateCustomers", null, "Orders");
   doCustomers.Update(this.NwindDataSet1, command);
}
```

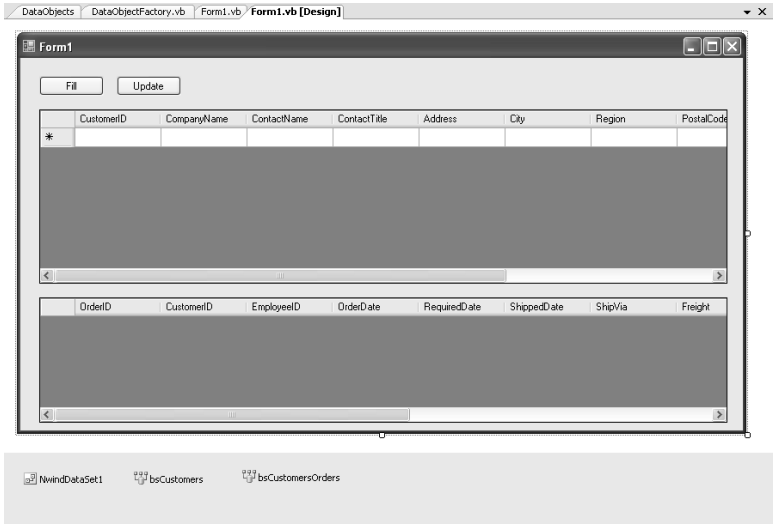

Figure 10-8 The main form of the DemoClient project

I commented on this code in the section titled "The DataObjectCommand Type" earlier in this chapter. Just notice that the client code has no direct reference to the actual data object: it uses a DataObjectFactory to get a data object that works with the specified configuration/table

combination, it specifies a data command in a generic way, and finally it interacts with the data object by means of the IDataObject<T> generic interface.

Compile the DemoClient project to ensure that everything is in place, but don't run it yet because it won't work until you define at least one data object for the "Access" configuration.

The DataObjects Project

Create a new Class Library project named DataObjects and ensure that the output directory for this project matches the output directory of the main application. This build configuration simulates what happens in a real-world application when you deploy a DLL holding one or more data objects in the main application's folder.

The DOCustomers Type

A data object is a class that implements the IDataObject(Of T) generic interface and that is marked with the DataObject custom attribute. For example, the following data object is used under the configuration named "Access" and can access the database table named "Customers":

```
[DataObject("Access", "Customers")]
public partial class DOCustomers : IDataObject<NWINDDataSet>
{
   // The IDataObject interface

   private List<IDataObjectCompanion> m_Companions = new List<IDataObjectCompanion>();

   public List<IDataObjectCompanion> Companions
   {
      get { return m_Companions; }
   }

   // The IDataObject<TDataSet> interface

   public NWINDDataSet Fill(NWINDDataSet ds, DataObjectCommand command)
   {
      …
   }

   public NWINDDataSet Update(NWINDDataSet ds, DataObjectCommand command)
   {
      …
   }
}
```

The main application invokes the Fill method to read one or more database tables into the DataSet. The DOCustomers object is expected to read data from the database and fill one or more tables of the DataSet passed in the first argument. The data object learns which child tables to read, if any, by looking at the ChildTable collection of the DataObjectCommand passed in the second argument. (Corresponding statements in the following listing are in

bold type.) Before actually reading any data, however, the data object notifies its companion objects that an operation is about to be performed:

```
public NWINDDataSet Fill(NWINDDataSet ds, DataObjectCommand command)
{
   if ( ds == null )
   {
      ds = new NWINDDataSet();
   }

   if ( DataObjectHelper.BeforeFill(this, ds, command) )
   {
      DataSets.NWINDDataSetTableAdapters.CustomersTableAdapter taCustomers =
         new DataSets.NWINDDataSetTableAdapters.CustomersTableAdapter();
      taCustomers.Fill(ds.Customers);
      if ( command.ChildTables.Contains("Orders") )
      {
         DataSets.NWINDDataSetTableAdapters.OrdersTableAdapter taOrders =
            new DataSets.NWINDDataSetTableAdapters.OrdersTableAdapter();
         taOrders.Fill(ds.Orders);
      }
   }
   DataObjectHelper.AfterFill(this, ds, command);
   return ds;
}
```

You can easily enhance the DOCustomers type to support additional commands. For example, this data object might be able to read data about individual customers and their orders:

```
// (Replace the bold type section of the previous listing with this code.)
string customerId = "";
if ( command.KeyValue != null )
{
   customerId = command.KeyValue.ToString();
}

DataSets.NWINDDataSetTableAdapters.CustomersTableAdapter taCustomers =
   new DataSets.NWINDDataSetTableAdapters.CustomersTableAdapter();
if ( customerId.Length == 0 )
{
   taCustomers.Fill(ds.Customers);
}
else
{
   taCustomers.FillByCustomerID(ds.Customers, customerId);
}

if ( command.ChildTables.Contains("Orders") )
{
   DataSets.NWINDDataSetTableAdapters.OrdersTableAdapter taOrders =
      new DataSets.NWINDDataSetTableAdapters.OrdersTableAdapter();
   if ( customerId.Length == 0 )
   {
      taOrders.Fill(ds.Orders);
   }
```

```
         else
         {
            taOrders.FillByCustomerID(ds.Orders, customerId);
         }
      }
```

Typically, the Update method is simpler and has fewer options. Here's an implementation of this method that writes the Customers table and, optionally, the Orders child table. This code splits in two parts the updates to the Customers table—first it inserts the new records and then it removes the deleted records—to not break the referential integrity of the records that take part in the Customers_Orders relation:

```
public NWINDDataSet Update(NWINDDataSet ds, DataObjectCommand command)
{
   if ( DataObjectHelper.BeforeUpdate(this, ds, command) )
   {
      DataSets.NWINDDataSetTableAdapters.CustomersTableAdapter taCustomers =
         new DataSets.NWINDDataSetTableAdapters.CustomersTableAdapter();
      // Send new and modified rows to the database.
      taCustomers.Update(ds.Customers.Select(null, null,
         DataViewRowState.Added | DataViewRowState.ModifiedCurrent));

      // Update the child table, if so requested.
      if ( command.ChildTables.Contains("Orders") )
      {
         DataSets.NWINDDataSetTableAdapters.OrdersTableAdapter taOrders =
            new DataSets.NWINDDataSetTableAdapters.OrdersTableAdapter();
         taOrders.Update(ds.Orders);
      }
      // Remove deleted records from the database.
      taCustomers.Update(ds.Customers.Select(null, null, DataViewRowState.Deleted));
   }
   DataObjectHelper.AfterUpdate(this, ds, command);
   return ds;
}
```

A more robust implementation should account for other relations that exist in the database. For example, you can't delete a row in the Orders table if you don't delete all the rows in the Order Details table that are related to that specific order. Also, a real-world data object should perform all the updates under a transaction so that all changes can be rolled back if an error occurs during the process.

At this point, you can compile the entire solution and run the DemoClient application. If you didn't make any mistakes, the application should be able to view and modify any record in the Customers and the Orders tables. Once you're sure that everything is working as expected, you can begin building a few companion objects for the DOCustomers type.

Data object companions can be located in the same assembly as the data objects they refer to or in a separate assembly. In the demonstration solution, I opted for the first approach to keep the number of projects as low as possible, but in most real-world cases you should put them in their own assembly.

The Tracer Companion Type

The simplest data object companion you can build is a tracer component, which displays details of all the operations being performed on data. Such a tracer component can work with any data object; thus, the first argument in the DataObjectCompanion attribute that marks it can be an empty string:

```
[DataObjectCompanion("")]
public class Tracer : IDataObjectCompanion
{
   public void BeforeFill(IDataObject obj, DataSet ds, DataObjectCommand command)
   {
      Console.WriteLine("[{0}] BeforeFill - Command:{1}",
         (obj as object).GetType().Name, command.Name);
   }

   public void BeforeUpdate(IDataObject obj, DataSet ds, DataObjectCommand command)
   {
      Console.WriteLine("[{0}] BeforeUpdate - Command:{1}",
         ( obj as object ).GetType().Name, command.Name);
   }

   public void AfterFill(IDataObject obj, DataSet ds, DataObjectCommand command)
   {
      Console.WriteLine("[{0}] AfterFill - Command:{1}",
         ( obj as object ).GetType().Name, command.Name);
   }

   public void AfterUpdate(IDataObject obj, DataSet ds, DataObjectCommand command)
   {
      Console.WriteLine("[{0}] AfterUpdate - Command:{1}",
         ( obj as object ).GetType().Name, command.Name);
   }
}
```

Recompile the DataObjects.dll, run the DemoClient again, and test the read and update commands: you should see the various diagnostic messages in the Visual Studio console window. (Needless to say, a real tracer object should send its output to a debugger or a file.)

The CustomerCache Companion Type

The second companion type in the DataObjects project is far more interesting in that it enables you to cache the result of a query on a local file transparently so that the actual database isn't accessed too often. Thanks to the CAP framework's infrastructure, such a caching feature can be achieved with very little code. First, define a user-level setting named CacheFile and assign it the path to the cache file you want to create, for example, c:\CustomersCache.xml. Next, enter this code:

```
[DataObjectCompanion("DataObjects.DOCustomers")]
public class CustomerCache : IDataObjectCompanion
{
```

```
// The location of the cache file is stored in the application's config file.
public string cacheFile = Properties.Settings.Default.CacheFile;

public void AfterFill(IDataObject obj, DataSet ds, DataObjectCommand command)
{
   if ( command.ChildTables.Contains("Orders") & !command.Canceled )
   {
      SaveToCache(ds);
   }
}

public void AfterUpdate(IDataObject obj, DataSet ds, DataObjectCommand command)
{
   if ( command.ChildTables.Contains("Orders") & !command.Canceled )
   {
      SaveToCache(ds);
   }
}

public void BeforeFill(IDataObject obj, DataSet ds, DataObjectCommand command)
{
   // If the cached file has already been saved today, use cached data.
   if ( File.Exists(cacheFile) && File.GetCreationTime(cacheFile).Date == DateTime.Today )
   {
      LoadFromCache(ds);
      // Let the data objects know that the command has been canceled.
      command.Canceled = true;
   }
}

public void BeforeUpdate(IDataObject obj, DataSet ds, DataObjectCommand command)
{
   // Nothing to do.
}

// Save and load from the cache.

private void SaveToCache(DataSet ds)
{
   Console.WriteLine("Dataset is being saved to {0}", cacheFile);
   ds.RemotingFormat = SerializationFormat.Binary;
   using ( FileStream fs = new FileStream(cacheFile, FileMode.Create) )
   {
      BinaryFormatter bf = new BinaryFormatter();
      bf.Serialize(fs, ds);
   }
}

private void LoadFromCache(DataSet ds)
{
   Console.WriteLine("Dataset is being loaded from {0}", cacheFile);
   ds.RemotingFormat = SerializationFormat.Binary;
   using ( FileStream fs = new FileStream(cacheFile, FileMode.Open) )
```

```
      {
         BinaryFormatter bf = new BinaryFormatter();
         DataSet ds2 = (DataSet) bf.Deserialize(fs);
         ds.Merge(ds2);
      }
   }
}
```

The companion object checks the creation date of the cache file in the BeforeFill method, and then, if the cache file has been created on the same day, the DataSet is filled with data taken from the local file and the command is canceled. Other companion objects can check the command's Canceled property and decide whether they should perform their intended action. For example, the Tracer companion might output a message that makes it clear that the default command has been canceled.

The CustomerFilter Companion Type

The last companion type I show is a filter that automatically removes from the DataSet all the records that the current client shouldn't see and that ensures that records being written to the database meet specific criteria. In this particular example, the CustomerFilter class displays only customers who are located in Germany (see Figure 10-9), but of course you can expand the code as you prefer:

```
[DataObjectCompanion("DataObjects.DOCustomers")]
public class CustomerFilter : IDataObjectCompanion
{
   public string Country = "Germany";

   public void AfterFill(IDataObject obj, DataSet ds, DataObjectCommand command)
   {
      NWINDDataSet ds2 = (NWINDDataSet) ds;
      // Remove all companies not in the right country.
      for ( int i = ds2.Customers.Rows.Count - 1; i >= 0; i-- )
      {
         if ( ds2.Customers[i].Country != Country )
         {
            // Remove this row from the result.
            ds2.Customers.Rows.RemoveAt(i);
         }
      }
   }

   public void AfterUpdate(IDataObject obj, DataSet ds, DataObjectCommand command)
   {}

   public void BeforeFill(IDataObject obj, DataSet ds, DataObjectCommand command)
   {}

   public void BeforeUpdate(IDataObject obj, DataSet ds, DataObjectCommand command)
   {
      NWINDDataSet ds2 = (NWINDDataSet) ds ;
      foreach ( NWINDDataSet.CustomersRow custRow in ds2.Customers.GetChanges().Rows )
```

```
      {
         if ( custRow.Country != Country )
         {
            throw new Exception("Records must have Country = " + Country);
         }
      }
   }
}
```

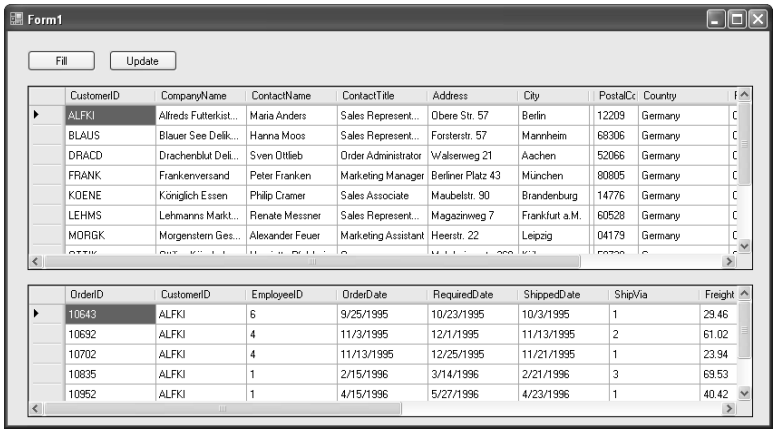

Figure 10-9 The demonstration application filtering customers by their country

You can improve the CustomerFilter type to support a filter based on an SQL clause that the type would use to modify the Select command. This approach would be far more efficient than reading all the data and discarding those pieces you aren't interested in is.

The GenericFilter Companion Type

The CustomerFilter type always filters in only customers that reside in Germany, which is a rather unrealistic assumption. A more useful filter class should let client applications decide which filter to apply. In general, a companion object might require that the client application provide one or more values to its constructor. In this case, the client application should instantiate the companion object directly, initialize its properties, and then add it to the data object's Companions collection:

```
IDataObject<NWINDDataSet> doCustomers = (IDataObject<NWINDDataSet>)
   factory.Create("Customers");
// Add a companion filter that filters in only customers whose City is Berlin.
doCustomers.Companions.Add(new GenericFilter("Customers", "City", "Berlin"));
```

Obviously, the previous code works only if the main client has a reference to the assembly where the GenericFilter type is defined.

A companion object meant to be instantiated directly by the client application shouldn't be marked by a DataObjectCompanion attribute because this attribute would force the CAP framework to instantiate the attribute directly, and such instantiation would fail if the

constructor takes one or more arguments. The following code shows how such a companion object can be implemented:

```
public class GenericFilter : IDataObjectCompanion
{
   public readonly string TableName;
   public readonly string FieldName;
   public readonly object FieldValue;

   public GenericFilter(string tableName, string fieldName, object fieldValue)
   {
      this.TableName = tableName;
      this.FieldName = fieldName;
      this.FieldValue = fieldValue;
   }

   public void AfterFill(IDataObject obj, DataSet ds, DataObjectCommand command)
   {
      DataTable dt = ds.Tables[TableName];
      // Remove all rows that don't match the condition.
      for ( int i = dt.Rows.Count - 1; i >= 0; i-- )
      {
         if ( !dt.Rows[i][FieldName].Equals(FieldValue) )
         {
            // Remove this row from the result.
            dt.Rows.RemoveAt(i);
         }
      }
   }

   public void BeforeUpdate(IDataObject obj, DataSet ds, DataObjectCommand command)
   {
      DataTable dt = ds.Tables[TableName];
      // Check that all rows match the filter condition.
      foreach (DataRow row in dt.GetChanges().Rows )
      {
         if ( ! row[FieldName].Equals(FieldValue) )
         {
            throw new Exception("Record doesn't match the filter criteria");
         }
      }
   }

   // AfterFill and AfterUpdate methods contain no statements.
   …
}
```

By now you should be convinced that companion objects can add a nearly unlimited degree of flexibility to your data-centric applications. Here are a few suggestions:

- Use the BeforeUpdate method to fix values that are out of the valid range.
- Implement sophisticated security and audit rules, for example, to prevent certain users from accessing data outside the normal working hours, and log all such attempts.

- Keep a log of queries that take longer so that you can later fine-tune the application or add new indexes to the database structure.
- Send an e-mail to a supervisor when data with exceptionally high or low values is entered, for example, when an employee enters an order with a value higher than a given threshold.
- Build more sophisticated caching policies than those adopted by the sample CustomerCache type.

You might also consider the opportunity to extend the CAP framework with more features. For example, the DataObjectCommand type might support additional properties to specify whether a transaction is requested or already exists, whether data pagination is requested, and so forth.