Chapter 8

Application Development

In this chapter I want to discuss the issues that will arise when you come to develop applications. A .NET application that lacks a user interface typically is made up of components, and UI applications consist of forms that have components and controls. Visual Studio.NET has been designed to make developing applications simple and to aid rapid application development (RAD) by allowing you to generate code simply by dragging and dropping components.

I will start by explaining what constitutes a component and what extra facility you get with controls. I will then show you how to develop a control that will integrate with the IDE's **Toolbox** and **Properties** windows, and how you can make it easier for the users of your controls to change their properties.

The world we live in moves day by day to a single global market, so it is extremely important that your application, components, and controls not be tied to a single locale. Internationalization in .NET is carried out with locale-specific resources. The VS.NET IDE has been designed to make the internationalizing process as simple as possible, and I'll show you how this works and how the localized resources are deployed. .NET resources are handled differently from Win32 resources, so I will show you how to create resources and how to include Win32 resources in your assemblies when .NET resources are deficient.

8.1 Developing Components

I mentioned in Chapter 6 that components are items that implement `IComponent`. This interface derives from `IDisposable`, which means that components allow explicit management of their resources, and will inform interested

classes when they are disposed of. In addition, the `IComponent` interface has a `Site` property that is initialized to the site of a container:

```
// C#
public interface IComponent : IDisposable
{
   ISite Site { get; set; }
   event EventHandler Disposed;
   // IDisposable members
   void Dispose();
}
```

A container implements the `IContainer` interface:

```
// C#
public interface IContainer : IDisposable
{
   ComponentCollection Components { get; }
   void Add(IComponent component);
   void Add(IComponent component, String string);
   void Remove(IComponent component);
   // IDisposable members
   void Dispose();
}
```

As the name suggests, the `ComponentCollection` class is an enumerable collection. The `Add()` and `Remove()` methods allow you to add components to this collection and remove them. The fact that `IContainer` derives from `IDisposable` is important because containers are used to hold resources (the components), so when an object that uses components is disposed of, the components should be disposed of, too, through the implementation of `IContainer.Dispose()`.

When you create a Windows Forms application you'll find that a container is created for you, as in this example:

```
// C#
public class MyForm : System.Windows.Forms.Form
{
   private System.ComponentModel.Container
      components = null;
   public MyForm() { InitializeComponent(); }
```

```
    protected override void Dispose(bool disposing)
    {
       if (disposing)
       {
          if (components != null) components.Dispose();
       }
       base.Dispose( disposing );
    }
    private void InitializeComponent()
    {
       this.components =
          new System.ComponentModel.Container();
       this.Size = new System.Drawing.Size(300,300);
       this.Text = "My Form";
    }
}
```

The `container` member is not required. Indeed, if you write forms code by hand (e.g., in C++), you can dispense with the `container` member as long as you call `Dispose()` on each component.

The VS.NET **Toolbox** window contains a tab with standard components: classes for the event log, MSMQ queues, performance counters, and of course the timer component. These components lack a user interface, but you can still drag and drop them onto a form in the Windows Forms designer, which will generate code to add the component to the form class and will display that you are using the component by showing an icon at the bottom of the **Designer** window. You can access a component's properties by selecting the component in the **Designer** window and then switching to the **Properties** window.

Creating components is relatively straightforward. Your component, of course, must derive from `IComponent`, and the best way to do this is to derive from `Component`. The component will most likely be used as an item in the **Toolbox** window, so it will need a **Toolbox** image (which I'll explain later).

When you drag a component from the **Toolbox** window and drop it on a form in the **Designer** window, you are actually creating an instance of the component. It is important therefore that the component does not rely on constructor parameters; instead, your component should be initialized through properties. For example, look at how the `System.Timers.Timer` component is used.

When you drag and drop a timer onto a form, the generated code will create the component using its default constructor, and the interval of the timer is initialized through the `Interval` property, even though the `Timer` class has a constructor that takes an interval parameter.

Some components—for example, `EventLog` and `MessageQueue`–allow you to access resources. Such components are great examples of components because they consist of code that GUI applications will use, but they themselves do not have a UI. Your components are likely to give access to similar resources.

8.2 Developing Controls

I find the process of developing a control rather odd. The whole point about a control is that it has a visual element, but the Windows Forms designer does not show the visual representation of a class derived from `Control`. Instead it gives a schematic showing the components that the control uses that you've dragged from the **Server Explorer** or **Toolbox** window. This makes developing the visual aspect of a control a bit of a nuisance: In effect you have to add a forms project to your control solution with a form that contains the control so that as you develop the control, you can see the effects as you make them. You do not have this problem when developing a `UserControl` class because the designer shows the control as a captionless, borderless form onto which you can drag and drop controls from the **Toolbox** window.

In this section I will describe the process of developing a simple control and point out some of the issues you will face. `UserControl` is composed of other controls, so developing a `UserControl` object is similar to developing a `Control` object, except that you have the additional steps of adding controls from the **Toolbox** window and adding event handlers generated by these constituent controls.

8.2.1 Developing a Sample Control

The control that I'll develop, called `DiskSpace`, is shown in Figure 8.1. This control has a property called `Disk` that holds the logical name of a disk on your machine. The control will display in its area the name of the disk and either the size

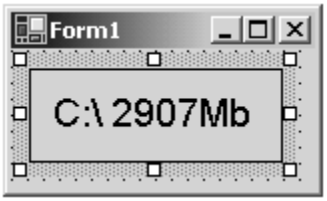

Figure 8.1 The `DiskSpace` control

of the disk or the amount of free space available to the current user on the disk. Which of these sizes will be displayed is determined by another property, called `Display`. These properties can be changed at design time in the **Properties** window.

The first step is to create a control library. I will use C# as the development language so that I can use the designer tool for some of the work. The project type to use is the Windows Control Library project, and I will call my project `DiskData`. Once you have created the project, the first thing you'll notice is that the **Designer** window will start up showing a gray, borderless box. Don't be fooled; you get this because the project wizard has generated control code derived from `UserControl`. In this example the control should derive from `Control`, and you can make this change in a moment.

While the control is visible in the **Designer** window, you should select its properties (through the context menu) and change its name (in the **Name** field) to `DiskSpace`. Now switch to code view by selecting **View Code** from the control's context menu, and edit the code so that the `DiskSpace` class derives from `Control`. Because the designer is not much use for this code, and because the control will not use other components, it is safe to remove the code that the wizard added for the designer. Finally, by selecting **Rename** in the context menu of the **Solution Explorer** window, change the name of the code file from `UserControl1.cs` to `DiskSpace.cs`.

After all these changes, the class should look like this:

```
// C#
namespace DiskData
{
   public class DiskSpace : Control
```

```
   {
      public DiskSpace()
      {
      }
   }
}
```

If you now switch back to the designer, you'll see that the gray surface of the control has been removed, and there will be a message telling you to add components from the **Server Explorer** or the **Toolbox** window. The control has a UI and hence needs to draw itself. To do this it has to implement the `OnPaint()` method:

```
// C#
protected override void OnPaint(PaintEventArgs pe)
{
   Graphics g = pe.Graphics;
   g.FillRectangle(new SolidBrush(BackColor),
                   0, 0, Size.Width, Size.Height);
   g.DrawRectangle(new Pen(ForeColor),
                   0, 0,
                   Size.Width - 1, Size.Height - 1);
}
```

This code accesses the inherited properties `BackColor` and `ForeColor`; it fills the area of the control with the color specified by `BackColor` and draws a rectangle within the inside edge with the color specified by `ForeColor`. The area of the control is accessed through the `Size` property. Even though this control clearly has a user interface, the **Designer** window *still* does not give a visual representation, so the only way you can see what you are creating is by adding the control to a form.

Before doing this, you should close the **Designer** window, compile the project and then use **Solution Explorer** to add a new **Windows Application** project to the solution (I call my project `CtrlTest`). When this project is created, you'll see an empty form, and if you open the **Toolbox** window, you'll see **Components** and **Windows Forms** tabs. Select one of these tabs (or even create your own tab), and while the **Toolbox** window is open, switch to Windows Explorer, drag the `DiskData.dll` file from the `bin\Debug` folder of the project

folder, and drop it on the **Toolbox** window. You'll see that the control will be added to the **Toolbox** window as shown in Figure 8.2. Notice that the image next to the control name is a cog. This is a standard image used when components don't specify a particular image; I'll explain how to change this later.

Now you can drag the `DiskSpace` control from the **Toolbox** window and drop it on the form, and you should see a gray square bordered by a black edge. As a quick test, grab one edge of the control and reduce the width or height; you'll see that the control does not repaint the edge that you have moved. Furthermore, if you increase the control's size, you'll see that the edge is moved but the interior is not repainted. The result is that lines appear on the control surface as the control is resized, as Figure 8.3 shows.

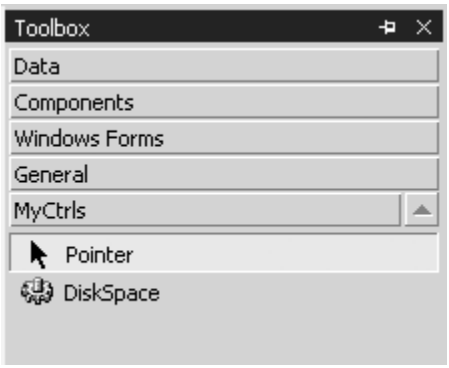

Figure 8.2 A control added to the Toolbox window

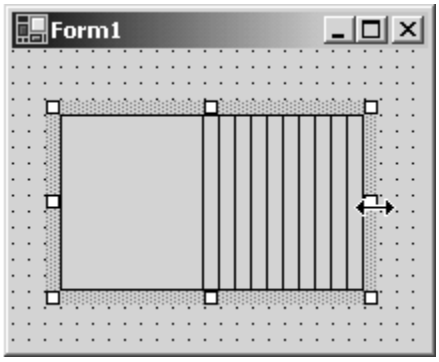

Figure 8.3 Resizing the control

The solution to this problem is to ensure that when the control is resized, it is redrawn. To do this you need to add the following method:

```
// C#
protected override void OnSizeChanged(EventArgs e)
{
   Invalidate();
   base.OnSizeChanged(e);
}
```

When the control is resized, this method will be called. I handle the resize event by indicating that the entire control should be redrawn. I could be more sophisticated and track the size of the control and then invalidate only the area that has changed, but for this example my code is sufficient.

The control has two properties, so the following code needs to be added to the class:

```
// C#
public enum DisplayOptions {TotalSize, FreeSpace};
private string disk = "C:\\";
private DisplayOptions
   display = DisplayOptions.TotalSize;
public string Disk
{
   get { return disk; }
   set
   {
      disk = value;
      Invalidate();
   }
}
public DisplayOptions Display
{
   get { return display; }
   set
   {
      display = value;
      Invalidate();
   }
}
```

The `Display` property indicates whether the size of the disk or its free space is shown, as identified by the enum. The `Disk` property indicates the

drive to be displayed. If you compile the project at this point and then select the control on the test form, you'll see that the **Properties** window has been updated with the new properties (Figure 8.4). Furthermore, the **Properties** window reads the metadata of the control to see that the `Display` property can have one of two named values, and it will insert these values in a drop-down list box.

By default the **Properties** window allows you to either type in a value (as in the case with `Disk`) or select a value from a list, and the **Properties** window will do the appropriate *coercion* from the value you input to the type needed by the property. You can change this behavior by applying the `[TypeConverter]` attribute to the property and pass the type (or name) of a class derived from `TypeConverter` as the attribute parameter. In addition, you can provide values for a drop-down list box, or even provide a dialog to edit the property, as I'll show later.

Figure 8.4 shows the properties in alphabetical order. The properties can also be listed by category; to do this you need to use an attribute on the property—for example:

```
// C#
[Category("DiskSpace")]
public string Disk{/* code */}
[Category("DiskSpace")]
public DisplayOptions Display{/* code */}
```

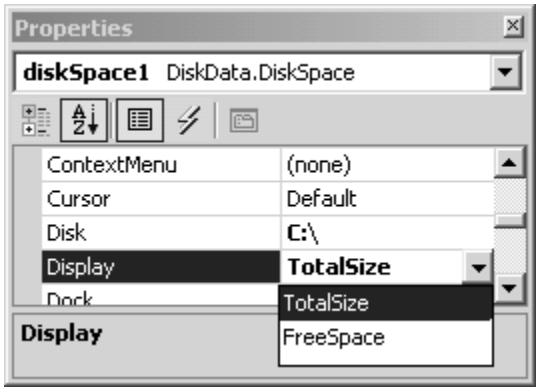

Figure 8.4 Properties window showing the new properties

The category can be one of the predefined categories documented in the MSDN entry for `System.ComponentModel.CategoryAttribute`, or you can create your own category, as I have done here. When you compile the assembly and look at the control's categorized properties, you'll see a new category called **DiskSpace**. Under this category are the two properties (see Figure 8.5).

The properties are shown in the **Properties** window because by default, all properties are browsable. If you want to indicate that the property should not be shown in the **Properties** window, you can use the `[Browsable(false)]` attribute. In a similar way, if you write code that uses an instance of the `DiskSpace` control, IntelliSense will show the property names in a list box when you type a period after the name of a variable of `DiskSpace` (for C#). You can use the `[EditorBrowsable]` attribute to alter this behavior: The parameter is `EditorBrowsableState`, and if you use the value `Never`, the property will not be shown in the IntelliSense list box; the default is `Always`. At the bottom of the **Properties** window is a space for a description of the property, and because these properties do not have descriptions, just the property name is given. To add a description to a property, you should use the `[Description]` attribute. Here are the changes:

```
// C#
// initialized to default value
private string disk = "C:\\";
[  Category("DiskSpace"),
```

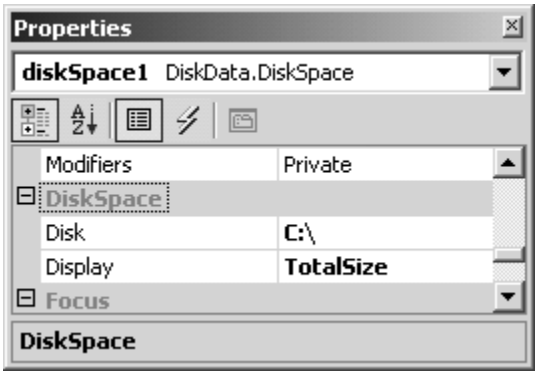

Figure 8.5 Categorized properties

```
   Browsable(true), EditorBrowsable,
   Description("The name of the disk") ]
public string Disk  { /* code */ }
[ Category("DiskSpace"),
  Browsable(true), EditorBrowsable,
  Description("Whether the total size or free "
              + "space on the disk is shown") ]
public DisplayOptions Display { /* code */ }
```

The name of the property given in the **Properties** window will be the name of the property in the class. You can use the `[ParenthesizePropertyName]` attribute to indicate that the name should be shown in parentheses, which means that the property will appear near the top of the **Properties** window when properties are shown in alphabetical view, or near the top of the category when they are shown in categorized view. You will notice that all of the screen shots of the **Properties** window that you have seen here show the values of the `Disk` and `Display` properties in bold. The **Properties** window uses the convention of showing in bold any properties that have been changed from their default values. This poses the question, How do you specify a default value?

There are two ways to do this. The first is to use the `[DefaultValue]` attribute on the property, passing the value as the constructor parameter. This option is fine for primitive types (the attribute constructor is overloaded for all of the base types). If the type is more complex, you can provide a string version of the default value, as well as the type to which the value should be converted, and the system will attempt to find a `TypeConverter` class to do the conversion. If there is no type converter, you can use the second way to specify a default value: adding two methods to the class with the names `Reset<property>()` and `ShouldSerialize<property>()`, where `<property>` is the property name. `Reset<property>()` should change the property to its default value, and `ShouldSerialize<property>()` should return a `bool` value indicating whether the property has a value other than its default. This last method gets its name from the fact that if the property does not have its default value, the value should be stored so that it can be used at runtime (for a form generated by the C# or VB.NET designer, this means initializing the control's property with the value).

If the property has a default value, the value does not need to be serialized because when the control is created, the property will have the default value.

Your implementation of the property must be initialized to the default value. Examples of the `Reset<property>()` and `ShouldSerialize<property>()` methods are shown in the following code:

```
// C#
// initialized to default value
private string disk = "C:\\";
[  Category("DiskSpace"),
   Browsable(true), EditorBrowsable,
   Description("The name of the disk"),
   DefaultValue("C:\\") ]
public string Disk  { /* code */ }
// default value
private DisplayOptions
   display = DisplayOptions.TotalSize;
[ Category("DiskSpace"),
  Browsable(true), EditorBrowsable,
  Description("Whether the total size or free "
              + "space on the disk is shown") ]
public DisplayOptions Display { /* code */ }
public void ResetDisplay()
{ display = DisplayOptions.TotalSize; }
public bool ShouldSerializeDisplay()
{ return display != DisplayOptions.TotalSize; }
```

In both cases you'll find that the property value will be shown in normal text if it is the default value.

Properties can be changed at runtime, and the change in a property value can have effects on other code. A good example is the `Size` property of a control: If the size changes, in most cases the control will need to be redrawn; thus you need to catch the event of the property changing. This is what I showed earlier with the code that overrides the `OnSizeChanged()` method. You should also add events that are generated when your properties change, by adding an event and an event generation method, as illustrated here:

```
// C#
public event EventHandler DiskChanged;
public event EventHandler DisplayChanged;
protected virtual void OnDiskChanged(EventArgs e)
{ if (DiskChanged != null) DiskChanged(this, e); }
protected virtual void OnDisplayChanged(EventArgs e)
```

```
{
   if (DisplayChanged != null)
      DisplayChanged(this, e);
}
```

The event generation method should be named `On<property>Changed()` and should generate the event. The `set` methods for the properties should call this method:

```
// C#
public string Disk
{  get { return disk; }
   set { disk = value;
         OnDiskChanged(null);
         Invalidate(); }
}
public DisplayOptions Display
{  get { return display; }
   set { display = value;
         OnDisplayChanged(null);
         Invalidate(); }
}
```

Sometimes several properties may depend on one property. If that is the case, when that property changes the dependent properties will change too. In this case the **Properties** window should refresh all the values. To indicate this requirement, such a property should be marked with the `[RefreshProperties]` attribute.

The next task that needs to be carried out for this control is to make it actually do something! The first thing is to implement the `Disk` property so that it checks that the value passed to the property is valid:

```
// C#
public string Disk
{
   get { return disk; }
   set { string str;
         str = Char.ToUpper(value[0]) + ":\\";
         string[] disks
            = Environment.GetLogicalDrives();
         if (Array.BinarySearch(disks, str) < 0)
            throw new IOException(value
                        + " is not a valid drive");
```

```
            disk = str;
            OnDiskChanged(null);
            Invalidate(); }
}
```

For this code to compile, you will need to add a `using` statement for the `System.IO` namespace at the top of the file. First I construct the disk name; then I obtain the list of logical drives on the current machine and perform a binary search to see if the requested disk is within the array of logical drive names. Now that I have a valid drive name, I need to obtain the size of the disk. I do this through interop to call the Win32 `GetDiskFreeSpace()` method:

```
// C#
[ DllImport("kernel32", CharSet=CharSet.Auto,
  SetLastError = true) ]
static extern bool GetDiskFreeSpace(
   string strRoot, out uint sectersPerCluster,
   out uint bytesPerSector, out uint numFreeClusters,
   out uint totalClusters);
protected override void OnPaint(PaintEventArgs pe)
{
   Graphics g = pe.Graphics;
   g.FillRectangle(new SolidBrush(BackColor),
                   0, 0,
                   Size.Width, Size.Height);
   g.DrawRectangle(new Pen(ForeColor),
                   0, 0,
                   Size.Width-1, Size.Height-1);
   uint spc, bps, fc, tc;
   GetDiskFreeSpace(disk, out spc, out bps,
                    out fc, out tc);
   long free, total;
   long bPerCluster = (spc*bps);
   free = bPerCluster*fc/(1024*1024);
   total = bPerCluster*tc/(1024*1024);
   StringFormat sf = new StringFormat();
   sf.Alignment = StringAlignment.Center;
   sf.LineAlignment = StringAlignment.Center;
   string str;
   if (display == DisplayOptions.FreeSpace)
      str = disk + " " + free + "Mb";
   else
```

```
        str = disk + " " + total + "Mb";
    g.DrawString(str, this.Font,
                 new SolidBrush(ForeColor),
                 new RectangleF(0, 0,
                       Size.Width, Size.Height),
                 sf);
}
```

For this code to compile, you should add a `using` statement for the `System.Runtime.InteropServices` namespace to the top of the file. The `OnPaint()` method calls the imported `GetDiskFreeSpace()` function and passes the `Disk` property. Depending on the value of `Display`, the string printed on the control is formatted as showing the total space on the disk or just the free space. Notice again how the control's properties are used. In the `DrawString()` method at the end of `OnPaint()`, I draw the string in the color specified by `ForeColor`, using the default font for the control.

Once you have rebuilt the control, you should be able to view it on the test form, and you should be able to change the `Disk` and `Display` properties and see the control on the form change its view at design time. Before I leave this section, I ought to explain one property that you'll see in the **Properties** window: the parenthesized `DynamicProperties` complex property, which will have a subproperty with the parenthesized name `Advanced`. If you select this property, you get a list of most of the properties that the control supports and a check box next to each. If you check a property in this list, the designer will add a section for the property in the application's `.config` file (an XML file that is installed in the same folder as the application), and at runtime when the control is loaded, its values will be set according to the values in this `.config` file. For example, if I use `DynamicProperties` to select the `Disk` property, the `.config` file will look like this:[1]

```
<configuration>
  <appSettings>
    <add key="diskSpace1.Disk" value="D:\" />
  </appSettings>
</configuration>
```

1. You will need to build the project to get the values written to the `.config` file.

Here I have specified that the `Disk` property of the control `diskSpace1` should have the value `D:\` when the control is loaded. The code on the form can still change this property; however, this is a useful facility because it allows you to give your users some control over how the controls on your forms are initialized.

8.2.2 Property Editor

When you type a value into the **Properties** window, what you are actually typing is a text value. Some types—for example, `Point`–are complex and are made up of subtypes. The **Properties** window reads the type of the property, recognizes that the property has subtypes, and displays these subtypes in the grid as nodes in a tree view. The grid allows you to edit each subobject individually or, through an editor class, the entire property as one.

When the values of the property have been edited, the values are converted to the appropriate types through a converter class. The framework type converter classes are shown in Table 8.1. If your type is not covered by one of these converters, you can create your own converter by deriving from `TypeConverter` and then pass the type of this class to the constructor of the `[TypeConverter]` attribute, which you should apply to the definition of the type that is converted.

Table 8.1 Type converter classes

`ArrayConverter`	`DecimalConverter`	`SByteConverter`
`BaseNumberConverter`	`DoubleConverter`	`SingleConverter`
`BooleanConverter`	`EnumConverter`	`StringConverter`
`ByteConverter`	`ExpandableObjectConverter`	`TimeSpanConverter`
`CharConverter`	`GuidConverter`	`TypeConverter`
`CollectionConverter`	`Int16Converter`	`TypeListConverter`
`ComponentConverter`	`Int32Converter`	`UInt16Converter`
`CultureInfoConverter`	`Int64Converter`	`UInt32Converter`
`DateTimeConverter`	`ReferenceConverter`	`UInt64Converter`

Imagine that you have developed a control class that has an array property:

```
// C#
int[] b = new int[4];
public byte[] Data
{
   get { return b; }
   set { b = value; }
}
```

When you view the property in the **Properties** window, you'll see it shown as its constituent parts, and you can edit each item. If you select the property itself, an ellipsis button will appear (see Figure 8.6); and when you click this button, an appropriate UI editor will be shown. In the case of an array of `Int32` members, the **Int32 Collection Editor** will be shown (Figure 8.7). This editor allows you to edit the values in the array, and to add and remove items in the array.

You can also write your own editor. For example, imagine that you want to create an editor for the `Disk` property so that it gives you only the option of the disks that are available on the current machine. The first action is to design an appropriate editor dialog, by adding a form to the project called `DiskEditor.cs` through the **Solution Explorer** window. Next you edit the class to look like this:

```
// C#
public class DiskEditor : Form
{
   private ComboBox cbDisks;
   private Button btnOK;
   private Container components = null;
   private string str;
   public string Value { get { return str; } }
```

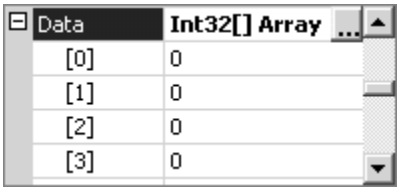

Figure 8.6 Array property in the Properties window

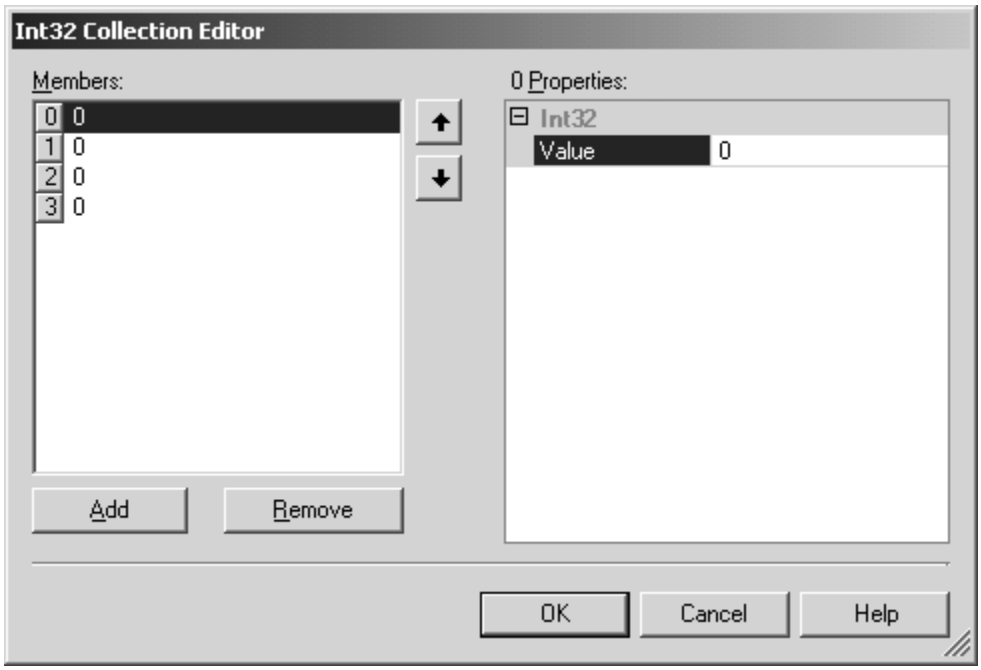

Figure 8.7 The collection editor for an array of `Int32` members

```
public DiskEditor(string currentVal)
{
   str = currentVal;
   ClientSize = new Size(120, 70);
   components = new Container();
   cbDisks = new ComboBox();
   components.Add(cbDisks);
   cbDisks.DropDownStyle
      = ComboBoxStyle.DropDownList;
   cbDisks.Location = new Point(10, 10);
   cbDisks.Size = new Size(100, 20);
   string[] disks
      = Environment.GetLogicalDrives();
   cbDisks.Items.AddRange(disks);
   cbDisks.Text = (string)cbDisks
                  .Items[cbDisks.FindString(str)];
   btnOK = new Button();
   components.Add(btnOK);
   btnOK.Location = new Point(30, 40);
   btnOK.Size = new Size(60, 20);
```

```
      btnOK.Text = "OK";
      btnOK.Click += new EventHandler(OnOK);
      Controls.AddRange(
         new Control[] {btnOK, cbDisks});
      FormBorderStyle = FormBorderStyle.FixedDialog;
      Text = "Disks";
   }
   protected override void Dispose( bool disposing )
   {
      if (disposing)
         if (components != null)
            components.Dispose();
      base.Dispose( disposing );
   }
   private void OnOK(object sender, EventArgs e)
   { Close(); }
   protected override void OnClosing(CancelEventArgs e)
   { str = (string)cbDisks.SelectedItem ; }
}
```

The `DiskEditor` constructor takes the current value of the property. The dialog has two controls: a drop-down list box that is initialized to the logical disk drives on the machine, and an **OK** button that, when clicked, will close the dialog. The form has a property called `Value` that is initialized to the disk that you selected, and this property is updated when the dialog closes.

Next you need a class derived from `UITypeEditor` that will be called to determine how the type should be edited. The `GetEditStyle()` method is called by the **Properties** window to determine how the value should be edited. `UITypeEditorEditStyle` has three values: `None`, which means that no UI element will be used to edit the value; `DropDown`, which means that a drop-down list will be shown; and `Modal`, which means that a modal dialog will be shown. I will first show an example of using a modal dialog. In this case the type editor class in `DiskEditor.cs` should be edited to look like this:

```
// C#
public class DiskTypeEditor : UITypeEditor
{
   public override object EditValue(
      ITypeDescriptorContext context,
      IServiceProvider provider, object value)
```

```
    {
       IWindowsFormsEditorService edSvc;
       edSvc = (IWindowsFormsEditorService)
               provider.GetService(
                  typeof(IWindowsFormsEditorService));
       DiskEditor editorForm;
       editorForm = new DiskEditor((string)value);
       edSvc.ShowDialog(editorForm);
       return editorForm.Value;
    }
    public override
       UITypeEditorEditStyle GetEditStyle(
          ITypeDescriptorContext context)
    { return UITypeEditorEditStyle.Modal; }
}
```

For this code to compile you need to add a `using` statement for both the `System.Drawing.Design` and `System.Windows.Forms.Design` namespaces to the top of the file. After the `GetEditStyle()` method is called, the **Properties** window will show either an ellipsis button (for the modal dialog) or a down-arrow button (for a drop-down list). When this UI button is clicked, the `EditValue()` method will be called to create the dialog to fill the list. The code here shows how to create the form. The first parameter of `EditValue()` provides information about the container, the **Properties** window. The second parameter gives access to the services that the **Properties** window provides, and in this case I request `IWindowsFormsEditorService`, which I use to call `ShowDialog()` to show the modal form. The final parameter of the method is the actual property that is being edited, so this parameter is used to initialize the form. When the modal form is closed, `ShowDialog()` will return; I access the value that the user selected through the `DiskEditor.Value` property. The final step is to indicate that a property will be edited with this particular editor; for this purpose the `[Editor]` attribute is used as follows:

```
// C#
[ Editor(typeof(DiskTypeEditor),
  typeof(UITypeEditor)) ]
public string Disk { /* code */ }
```

You will need to add a `using` statement for the `System.Drawing.Design` namespace to the top of the `DiskSpace.cs` file. Now when the ellipsis box of

the `Disk` property is clicked, the dialog will be shown (Figure 8.8). When the dialog is dismissed, the selected value will be written to the property.

It may seem a little over the top to have a whole dialog to present this data; the alternative is to use a drop-down list box, which the following class does, and you should add to the `DiskEditor.cs` file:

```
// C#
public class DiskTypeEditor2 : UITypeEditor
{
   private IWindowsFormsEditorService edSvc;
   public override object EditValue(
      ITypeDescriptorContext context,
      IServiceProvider provider, object value)
   {
      edSvc = (IWindowsFormsEditorService)
              provider.GetService(
                 typeof(IWindowsFormsEditorService));
      ListBox cbDisks;
      cbDisks = new ListBox();
      string[] disks = Environment.GetLogicalDrives();
      cbDisks.Items.AddRange(disks);
      cbDisks.Text = (string)cbDisks
         .Items[cbDisks.FindString((string)value)];
      cbDisks.SelectedValueChanged
         += new EventHandler(TextChanged);
      edSvc.DropDownControl(cbDisks);
      return cbDisks.Text;
   }
   public override UITypeEditorEditStyle
      GetEditStyle(
          ITypeDescriptorContext context)
   { return UITypeEditorEditStyle.DropDown; }
```

Figure 8.8 The disk editor dialog

```
    private void TextChanged(
       object sender, EventArgs e)
    { if (edSvc != null) edSvc.CloseDropDown(); }
}
```

The `GetEditStyle()` method of the `DiskTypeEditor2` class returns `UITypeEditorEditStyle.DropDown` to indicate that the **Properties** window should show the down arrow button. The `EditValue()` method creates a list box and initializes it with the names of the logical disks. This list box is shown by a call to the blocking method `IWindowsFormsEditorService.DropDownControl()`, and it is removed by a call to `CloseDropDown()`. The user expects to have drop-down list box behavior; that is, when an item is selected, the drop-down box should be removed. To get this behavior I add a handler to the list box that calls `CloseDropDown()`, which makes the blocked `DropDownControl()` method return. At this point I can access from the list box control the item that was selected and return it from `EditValue()`.

8.2.3 Licensing

Controls can be licensed; therefore you can add code to check whether the control is being used in a context where it is permitted. The licensing model recognizes two contexts: *design time* and *runtime*. Design time is the time when the control is being used in a designer (such as the Windows Forms designer) and as part of other code, such as a form. A developer must have a design-time license to be able to integrate your control into his application. Once the application has been compiled, it will be distributed to users and run, creating a new situation: The licensed control will perform a check for a runtime license when the application is run; if the runtime license is valid, the control can be created.

Having two licenses like this means that you can have a licensing scheme that is more secure for the design time than for the runtime. The licensing is based on a class called a *license provider* that is called to generate a license when an attempt is made to instantiate the object. Here is a license provider class:

```
// C#
public class LicProvider : LicenseProvider
```

```
{
   public override License GetLicense(
      LicenseContext context,
      Type type, object instance,
      bool allowExceptions)
   {
      if (context.UsageMode
            == LicenseUsageMode.Designtime)
      {
         if (!CheckForLicense())
         {
            if (!allowExceptions)
               return null;
            throw new LicenseException(GetType());
         }
         return new MyLic(type.Name
                          + " design time");
      }
      else
         return new MyLic(type.Name
                          + " runtime time");
   }
}
```

This provider is passed a `LicenseContext` object that indicates the context in which the license is being requested—either `LicenseUsageMode.Designtime` or `Runtime`. Your license provider can then check whether the license is available (as my method `CheckForLicense()` does)—for example, by looking for the location of a valid license file or a registry value. If the check succeeds, a new license can be created. If the license check fails, the license provider should throw a `LicenseException` exception if `allowExceptions` is `true` or just return `null` if it is `false`. In this example I have decided that the control should be freely available at runtime, so I don't perform any runtime checks; I merely return the license.

The license object should derive from `License` and provide implementations of the `LicenseKey` property and the `Dispose()` method. In my implementation I simply store a string:

```
// C#
public class MyLic : License
```

```
{
   string str;
   public MyLic(string t){ str = t; }
   public override string LicenseKey
   { get { return str; } }
   public override void Dispose(){}
}
```

The `LicenseKey` property is not intended to be a secure key. Instead it should be treated as an opaque cookie—an encoded string perhaps—that gives access to other data. This string could be stored as a resource in an assembly.

The license provider is associated with the control through the `[LicenseProvider]` attribute, and the control should call the `LicenseManager` object to check that the license is valid:

```
// C#
[LicenseProvider(typeof(LicProvider))]
public class DiskSpace : Control
{
   public DiskSpace()
   {
      LicenseManager.Validate(
         typeof(DiskSpace), this);
   }
   // code
}
```

If the control is not licensed, the call will throw an exception. If this happens in the Windows Forms designer, you'll get a message like the one shown in Figure 8.9. If the call to `Validate()`fails at runtime (a runtime license was not available), a `LicenseException` exception will be thrown. In the code I show

An error occurred while loading the document. Fix the error, and then try loading the document again. The error message follows:

An exception occurred while trying to create an instance of DiskData.DiskSpace. The exception was "An instance of type 'DiskData.DiskSpace' was being created, and a valid license could not be granted for the type 'DiskData.DiskSpace'. Please contact the manufacturer of the component for more information.".

Figure 8.9 Error message received if a form opened in the Windows Forms designer has a control that is not licensed

here I do not catch this exception because I want to make sure that if `Validate()` fails, the control will not load.

The Framework Class Library comes with one implementation of `License` called `LicFileLicenseProvider`. `LicFileLicenseProvider` will check for the existence of a license file, in much the same way as many ActiveX controls are licensed today.

8.2.4 Toolbox Items

The **Toolbox** can take any item derived from `IComponent`. When you add a control to the **Toolbox** window, the control will be shown with the standard control bitmap image, a cog. To change this image you have to apply the `[ToolboxBitmap]` attribute to the control class. The image should be a 16×16 bitmap embedded as part of your assembly, and it should have the same name as your class; for example, if your class is called `DiskSpace`, the bitmap should be called `DiskSpace.bmp`. To add the bitmap you use the C# **Solution Explorer** window's **Add New Item** on the **Add** context menu. The bitmap should be an embedded resource (which I'll explain later), so through the bitmap's **Properties** window you should change its **Build Action** property to **Embedded Resource**. Finally, the constructor parameter of the `[ToolboxBitmap]` attribute should take the type of the class to which it is applied:

```
// C#
[ToolboxBitmap(typeof(DiskSpace))]
public class DiskSpace : Control
{
   // code
}
```

For this new bitmap to be shown in the **Toolbox** window, you will need to remove the old control (from the context menu, select **Delete**) and then add it again by dragging and dropping it from Windows Explorer to a tab in the **Toolbox** window.

8.3 Resources and Internationalization

.NET supports a different model of resources from that supported by Win32. In Win32, resources are held in a section that is part of the PE (portable executable) file format; the resources are embedded within this segment. .NET

resources are part of an assembly, but they can be embedded within the assembly or supplied as separate files. In this section I'll explain how resources are generated with Visual Studio.NET and how your code can access them.

8.3.1 Resources and .NET

.NET has been designed with internationalization in mind. Imagine that you download an application from a Web site that you trust and the Web site is in a locale different from yours. You would expect the application's developers to have created the application in their own locale. However, if the language is different from yours, you will hope that the application has been localized to your locale and that the Web site gives you the option of downloading different localized versions. Win32 applications typically used this scheme. It is possible in Win32 to create resource DLLs for locale-specific resources, but this means that the developer has to explicitly load the resource from the DLL.

.NET allows you to create locale-specific resources, but it is far more sophisticated than Win32 because the Framework Class Library provides a class (`ResourceManager`) that will automatically load the resources for the current locale. These resources can be part of the current assembly, or they can be part of a separate assembly called a *satellite assembly*.

8.3.2 Locales, Cultures, and Languages

.NET uses the naming convention defined in RFC 1766. Cultures are named with the following pattern: `xx-yy`, where the two letters `xx` represent a language (e.g., `en` for English, `de` for German, or `fr` for French), and `yy` represents an area where that language is used (e.g., `GB` for the United Kingdom, `AU` for Australia, and `US` for the United States). Together, a language and an area represent a particular *culture,* so `en-US` represents English spoken in the US and implies hamburgers, Coke, and baseball. Whereas `en-GB` is the Queen's English and implies roast beef, tea in china cups, and cricket. (Well, you get the idea.) Without the area (e.g., `en`), a resource is area neutral; without a language, a resource is both language and area neutral. Most cultures can be represented by this four-letter style, but if further delineation is required, you can add extra pairs of letters.

The Framework Class Library provides the `CultureInfo` class to represent a particular culture. You can initialize this class by passing to the constructor either the RFC 1766 string or a locale ID (LCID). As I mentioned in Chapter 2, a culture can be used to format items like dates:

```
// C#
CultureInfo ci = CultureInfo("en-GB");
Console.WriteLine(DateTime.Now.ToString(
   "F", ci.DateTimeFormat));
```

Here the date is printed at the command line in the UK format. Because different cultures that use the same language have different rules for formatting, the `CultureInfo` class must be initialized with enough information, and a language identifier is not enough. If you do not specifically use a culture in format code, the current culture will be used. This culture is a per-thread value and is a read/write property of the current thread:

```
// C#
CultureInfo ci = CultureInfo("en-GB");
System.Threading.Thread
   .CurrentThread.CurrentCulture = ci;
Console.WriteLine(DateTime.Now.ToString());
```

.NET resources are not as strict as formatting code, so the `ResourceManager` class (which is used to locate and load locale-specific resources) allows you to provide resources that are totally neutral, area neutral, or culture specific. Again, this information is set on a per-thread basis through the `Thread.CurrentUICulture` property.

8.3.3 Creating Resources

Assemblies contain either compiled resources or uncompiled resources, and these can be either embedded within the assembly or supplied as a separate file and a link provided within the manifest of the assembly. Resources in an assembly are named. For example, here is some IL:

```
// IL
.assembly App
{
```

```
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.mresource public MyRes.resources
{
}
.module App.exe
```

This code indicates that an assembly called `App.exe` has a resource called `MyRes.resources`, which can contain several items, but ILDASM does not decompile the resource format, so these resources are not shown in IL. If the resource is a compiled resource, it can be read with the classes in `System.Resources`, as I'll explain in a moment. Otherwise the resource should be read as a single item, through the assembly object.

A resource can be embedded in an assembly with the C# compiler through the `/res` switch:

```
csc /res:MyRes.resources app.cs
```

This command will compile a C# file called `app.cs` and embed an already compiled resource called `MyRes.resources` in the assembly. The resource in the assembly will also be called `MyRes.resources`. If this is not what you want, you can append the switch with the name of the resource (separated by a comma).

The C++ linker has an `/assemblyresource` switch that you can use to embed a resource in an assembly. The resource will have the name of the resource file that you embed, and unlike the C# compiler, you cannot rename it through the switch:

```
link /out:app.exe
   /assemblyresource:MyRes.resources app.obj
```

If the resource is not compiled, it can be read only by explicit access of the resource through the assembly manifest:

```
// C#
Assembly assem
   = Assembly.GetCallingAssembly();
Stream stm;
```

```
stm = assem.GetManifestResourceStream(
   "MyRes.resources");
```

This code will return a stream that has all of the resource. It does not matter whether this resource is compiled, uncompiled, linked, or embedded. The following code will print out this stream to the command line:

```
// C#
while(true)
{
   int i = stm.ReadByte();
   if (i == -1) break;
   if (i < 32 || i > 127)
      Console.Write(".");
   else
      Console.Write((char)i);
}
Console.WriteLine();
```

An assembly can have a link to a resource. You can create this link with the C# compiler using the `/linkres` switch:

```
csc /linkres:data.txt,MyRes.resources app.cs
```

This command will compile the file `app.cs`, add a link to the file `data.txt`, and call the resource `MyRes.resources`. Clearly, if the resource is linked, it must be available through the link at runtime. Here is the IL produced by the preceding code:

```
// IL
.file nometadata data.txt
    .hash = (1E 7B 82 95 E5 DA 4B 04
             7A 56 47 DE EE C2 E7 7E
             1D 19 26 90 )
.mresource public MyRes.resources
{
  .file data.txt at 0x00000000
}
```

The `resgen` tool is used to compile or decompile resources. When resources are being compiled, the input can be either a text file (with the extension `.txt`) or an XML file (with the extension `.resx`). The text file can be used only

for string resources; it is structured as a series of name/value pairs with the two separated by an equal sign. Here is an example:

```
; text resource file
ErrNoFile=File {0} cannot be found
MsgStarted=Application has started
```

The code that uses the resources refers to the first string with the identifier `ErrNoFile`. If you want to use binary resources (e.g., images), you have to use XML resources. The XML file equivalent to the name/value pairs just shown is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- schema -->
<root>
  <data name="MsgStarted">
    <value>Application has started</value>
  </data>
  <data name="ErrNoFile">
    <value>File {0} cannot be found</value>
  </data>
  <resheader name="ResMimeType">
    <value>text/microsoft-resx</value>
  </resheader>
  <resheader name="Version">
    <value>1.0.0.0</value>
  </resheader>
  <resheader name="Reader">
    <value>
      System.Resources.ResXResourceReader
    </value>
  </resheader>
  <resheader name="Writer">
    <value>
      System.Resources.ResXResourceWriter
    </value>
  </resheader>
</root>
```

The `<resheader>` nodes give information about the format of the resources and the names of the classes used to read and write the resources. All of these `<resheader>` nodes except the `Version` node are required. For space reasons, I have not shown the schema, but in any case it is not needed. Although it is possible to write `.resx` files by hand, it is much easier to use the

VS.NET IDE, especially when you consider binary resources. Binary resources still have to have `<value>` nodes in the XML file, and to do this they must be converted to a readable format by something like base64 encoding. It is much easier to allow the IDE to do this for you, as I'll show in the next section.

`resgen` can also be used to decompile resources. If the input file has the extension `.resources`, `resgen` knows that it has to decompile resources. It determines the format that you require by the extension of the output file you specify. The general process of compiling resources is shown in Figure 8.10.

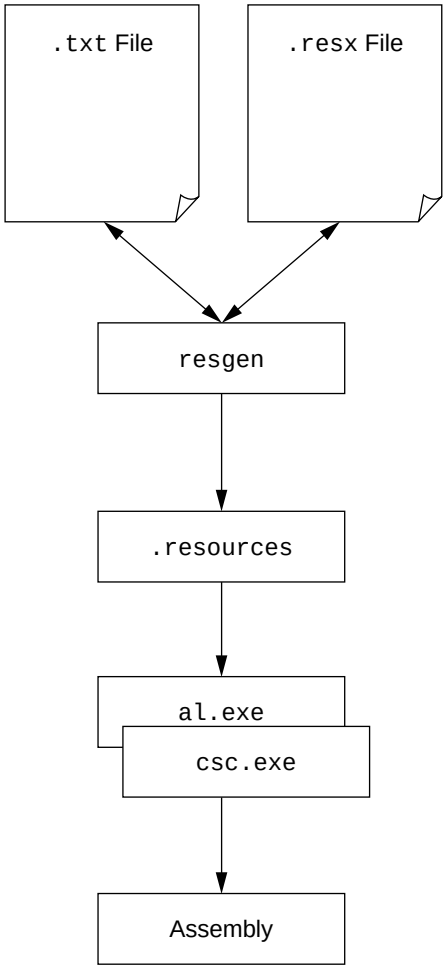

Figure 8.10 Resource compilation process

8.3.4 Managed C++ and Resources

Managed C++ projects allow you to add resources through the **Solution Explorer** or **Class View** window, but these will be Win32 resources. If you want to add your own .NET resources, you need to edit the project settings. Here are the steps: First you need to add an XML file to your project. To do this you should use the **Add New Item** dialog of **Solution Explorer**, and ensure that the extension of the file is `.resx` (the `resgen` utility insists that XML resource files have this extension). If you forget to give the file this extension, you will have to remove the file from the project, rename it using Windows Explorer, and add the renamed file to the project with **Add Existing Item** from the C++ **Solution Explorer** context menu. The reason is that the C++ **Solution Explorer** (unlike the C# **Solution Explorer**) does not allow you to rename a file that has been added to a project.

Once you have added the `.resx` file to the project, you should add the bare minimum of resource file contents: the `<root>` node and the three `<resheader>` nodes I mentioned earlier: `ResMimeType`, `Reader`, and `Writer`. After that it makes sense to add at least one `<data>` node (essentially as a template), and then you can edit the resource file using the XML designer.

The next task is to add the `.resx` file to the build. To do this you should select properties of this file from the **Solution Explorer** context menu by selecting **General Configuration Properties** and making sure that the **Tool** property option selected is **Custom Build Tool**. You can then set the tools command line through the **Custom Build Step** option (Table 8.2).

Choosing **Custom Build Step** will allow you to build the resource; however, you also need to embed the resource in the assembly, and to do this you need to edit

Table 8.2 Custom Build Step properties for an `.resx` file

Property	Value
Command Line	`resgen $(InputFileName) $(OutDir)\$(InputName).resources`
Description	`Building .NET resources`
Outputs	`$(OutDir)\$(InputName).resources`

the linker options. You select the properties of the project through the **Solution Explorer** window, and then in the **Property Pages** dialog you select the **Linker** node and then the **Input** node. Within the grid you'll see a property called **Embed Managed Resource File**; you change the value of this property as follows:

```
$(OutDir)\$(InputName).resources
```

This parameter assumes that the name of the `.resx` file that was compiled had the same name as the project. Once you have made these changes, you should be able to add string resources to the project through the `.resx` file.

Image files are not so easy; the problem is that you have to encode image files into a format that can be put in an XML file. A utility called `resxgen` will allow you to do this; it is supplied as an example in the .NET Framework `Samples` folder. However, the problem with this tool is that it will generate an entire `.resx` file from a single binary file. You cannot use it to add a binary resource to an existing `.resx` file.

8.3.5 C# and Resources

In this section I will give just a basic overview of using resources in C# projects; the sections that follow will go into more detail. To add a resource to a C# project you use the **Add Class** dialog of **Solution Explorer**. The **Resources** category shows that you can add bitmaps, icons, cursors, and string resource files. The resource files that it mentions here are `.resx` files that you'll typically use to add strings to the assembly, similar to adding a string table in a Win32 resource file. `.resx` files are XML files and are used as an input to the resource compiler, `resgen`, which I'll cover later. These resource files can also contain binary data like icons, but the data is stored in the `.resx` file as base64 encoded.

When you add one of the image files to the project, you can use the item's properties to see how the resource will be added to the assembly. **Build Action** gives the options of **None**, **Compile**, **Content**, and **Embedded Resource**. **Content** does not add the resource to the assembly, but it does indicate that the file should be deployed with the output of the project; **Compile** requires that you specify the compile tool through the **Custom Tool** property, and **Embedded Resource** will add the resource to the assembly without compiling.

For example, if you add an icon to your project and change its **Build Action** value to **Embedded Resource**, you will get the following IL when you build the assembly:

```
// IL
.mresource public myAssem.myIcon.ico
{
}
```

Here the icon file is called `myIcon.ico`, and the assembly is called `myAssem`. You can read this resource using `Assembly.GetManifestResourceStream()` and pass the stream as a construction parameter to the `Icon` class. For example, the following code loads an embedded resource as an icon for the `NotifyIcon` class that is used to create a tray icon:

```
// C#
// NotifyIcon trayIcon is a private class member
// this code is in constructor and components
// is the Container created in InitializeComponents
trayIcon = new NotifyIcon(components);
Assembly assem = Assembly.GetExecutingAssembly();
// assembly is called Tray; icon file is called MyIcon.ico
trayIcon.Icon = new Icon(
   assem.GetManifestResourceStream(
      "Tray.MyIcon.ico"));
```

Other resources, such as bitmaps and cursors, can also be loaded in this way. If you add a resource to a project as **Content**, it will be distributed with the output of the project as a separate file. Note that this is *not* the same as being part of a multifile assembly, as I mentioned in Chapter 1. When you add a link to an external resource file (through the `/linkres` switch to `csc`), the compiler will add a hash of the resource file to metadata in the assembly's manifest. To get the names of all such resources, you can use `Assembly.GetManifestResourceNames()`, and the names returned can be passed to the constructor of `Icon`, `Cursor`, or `Bitmap` to load the resource. When you specify that the **Build Action** value of a resource is **Content**, there will be no information about this in the assembly's manifest, so your code needs to know the name of the file.

As you'll see in a moment, the icon for a form is shown as a property for that form when viewed in the **Designer** window. However, the **Cursor** property shows only standard cursors in the **Properties** window. If you want to use a custom cursor, you can simply add a cursor as an embedded resource and use code similar to that shown here to load the cursor and make it the cursor for the form.

When you add an icon to a project, the wizard will show a 32×32 icon with 16 colors. This size is fine for the large-icon view in Windows Explorer, but it is too large for the form's icon. Icon files can contain images of different sizes and color depth, and it turns out that the icon created by the wizard also has a 16×16 icon image with 16 colors. To switch between the two sizes, you should select **Current Icon Image Types** on the **Image** menu (or use the `Image.MoreIcons` command, which will list all the icons). If you want to add another icon type to the icon, there is a **New Icon Type** menu icon (for the command `Image.NewImageType`).

Form icons are a different situation. When you add a form to a C# project, the IDE will create a `.resx` file with the same name as the form specifically for the resources that the form will use. One of these resources, of course, is the form's icon. Normally you will not see this `.resx` file in the **Solution Explorer** window because it will be a hidden file. To view this file you need to click on the **Show All Files** button, and you need to close the form in the Windows Forms designer.

To add an icon to a form, first you have to add an icon to the project as I have shown here, but leave the icon's **Build Action** value as **Content**. Next you should select the form's properties in the Windows Forms designer and click on the form's **Icon** property. This will bring up a dialog that will allow you to browse for the icon you just created. When you have done this, the IDE will insert the icon as a node in the `.resx` file. In a similar way, if you add a background image (the `BackgroundImage` object) to the form, the image file will be added to the `.resx` file. These are just special cases: They are resources required by the form, so they have to be stored along with the form.

8.3.6 Forms and Localization

Every form has a property called `Localizable`. This is not a property inherited from the `Form` base class; it is a pseudoproperty created by the **Properties**

window for `Form` objects and `UserControl` objects (but *not* `Control` objects). When you change this property from the default of `False` to `True`, the **Properties** window will copy all the form's properties to the form's `.resx` file. The `.resx` file with the form's name will have the default values for the form.

When you change the `Language` property (another pseudoproperty), the IDE will create a `.resx` file for the selected language, named according to the language (so if the form is called `myForm`, the UK English resource file will be called `myForm.en-GB.resx`). This resource file will contain the difference between the default resource and the localized resource. So if you have set the `Icon` property in the default resources, this value will be used by all cultures *unless* you explicitly change it for a specific culture. Thus, localizing your forms is as simple as generating the default resource for the form, then specifying the values for only those properties that you want to localize by changing the `Language` property in the **Properties** window, and finally changing the property to its localized value in the **Properties** window.

The effect of `Localizable` is recursive, so if you have controls on a form, you can change the properties of those controls for a specific culture, and those properties will be written to the appropriate `.resx` file. You are most likely to use this option if the form has a menu. You add a menu to a form by adding a `MainMenu` control, and the Windows Forms designer allows you to add submenus, handles, and embellishments like check marks and radio buttons. When you develop an application, you should start by building up the menu using the default `Language`. And once you have created the menu layout and the handlers, you can localize the menus by changing the form's `Language` property to a language other than the default and then changing the menu items' text values. The values that you change will be written to the resource file for the culture.

Of course, the Windows Forms designer generates code. When the form is not localized, the designer will add code to assign the property value to the property in the `InitializeComponent()` method. When you localize the form, the designer changes the code to use a `ResourceManager` object. I will go into more detail about this class in the next section, but as I have already mentioned, this class will locate the appropriate resources section in the assembly (or in satellite assemblies) and give access to the values. For example, here is some code that is generated for you:

```
// C#
private void InitializeComponent()
{
   System.Resources.ResourceManager resources =
      new System.Resources.ResourceManager(
         typeof(myForm));
   // some properties omitted
   this.Icon = ((System.Drawing.Icon)
      (resources.GetObject("$this.Icon")));
   this.Text = resources.GetString("$this.Text");
   this.Visible = ((bool)
      (resources.GetObject("$this.Visible")));
}
```

As you can see, `ResourceManager` is initialized with the type of the form, which gives the class one part of the information it needs to locate the resource in the assembly. This class reads the UI culture of the current thread, and using this and the name of the form, it can determine the name of the form's resources (i.e., it will search `myForm.resources` for the default resources, and `myForm.en-GB.resources` for UK English resources). It then accesses the string resources using `GetString()` (as shown here, with the `Text` property), and all other resources are accessed through `GetObject()`. The designer uses the convention of naming each resource `$this.<propertyname>`. Because `ResourceManager` determines the appropriate resources for the current locale, you do not need to write this locale-specific code.

When you compile the form, the project will add the default resources to the assembly that contains the form and will generate a satellite assembly for each of the other resource files. These satellites will be named according to the satellite convention: `<formAssem>.resources.dll`, where `<formAssem>` is the name of the form's assembly. Each satellite will be located in a folder named according to the locale of the satellite, as I'll describe later.

8.3.7 Resource Classes

The `System.Resources` namespace has the classes that are needed to read and write compiled resources. `ResourceReader` enumerates resources and gives access to them through an `IDictionaryEnumerator` interface. The constructor parameter takes either the name of a file or an already opened

stream, which, conveniently, is what `Assembly.GetManifestResourceStream()` will return:

```
// C#
System.Reflection.Assembly assem;
assem = Assembly.GetExecutingAssembly();
Stream stm;
stm = assem.GetManifestResourceStream(
   "myAssem.myResources.resources");
ResourceReader reader = new ResourceReader(stm);
foreach (DictionaryEntry de in reader)
   Console.WriteLine(de.Key +" = "+de.Value);
```

This code will look for a resource called `myAssem.myResources.resources` and will print the name/value pairs contained in it.

The `ResourceWriter` class is used to write compiled `.resource` files. It takes as a construction parameter either the name of the file or, if you have an already open file, a writable stream. You can then use one of the overloaded `AddResource()` methods to add a string or a binary value to the resource. If you choose to write a binary value, you can pass either a `byte[]` array with the object already serialized or a reference to the object. If you pass an object reference, it must be an instance of a serializable class. The actual resource is not created until you call the `Generate()` method, which is also called by the `Close()` method that closes the output stream. Thus you can write code like this:

```
// C#
ResourceWriter rw
   = new ResourceWriter("myResources.resources");
rw.AddResource("TestString", "Some string data");
byte[] b = new byte[]{0, 1, 2, 3, 4};
rw.AddResource("TestData", b);
rw.Generate();
rw.Close();
```

This code will add the resources to a file called `myResources.resources`. The ability to write resource files is useful when you consider the `ResourceManager` class. This class is used to provide convenient access to localized resources bound to an assembly or to satellite assemblies, or located

in separate files. As I have already mentioned, this class will read the UI culture of the current thread, and using this and a base name for the resource, it will locate the resource in the local file or in a satellite assembly. Typically you will add a resource for a form, so the base name of the resource will be derived from the form's type. This is why the type of the form was used as the constructor parameter in the code I showed earlier.

If you add a separate resource file to your project, you need to provide to the `ResourceManager` constructor the name of the resource that will be derived from the resource file's name. For example, if you add a resource file called `myRes.resources` to the project, the resource will be named `myRes.resources`, and the base name will be `myRes`. Thus the following code will initialize a `ResourceManager` object to load these resources:

```
// C#
ResourceManager rm
   = new ResourceManager("myRes", assem);
```

The `assem` reference is the `Assembly` object that contains the resource (or an assembly that has satellites that contain localized resources). You can get a reference through the type of an existing object (e.g., in a form you can call `this.GetType().Assembly`) or through the static members of `Assembly`: `GetAssembly()` to get the assembly for a particular type, `GetCallingAssembly()` for the assembly that loaded the current assembly, or `GetExecutingAssembly()` to get the current assembly.

If you have created resource files using a `ResourceWriter` object, you can load these resources using the static `CreateFileBasedResourceManager()` method of the `ResourceManager` class. In the previous example, then, you can load the resources in `myResources.resources` with the following code:

```
// C#
ResourceManager rm.
ResourceManager.CreateFileBasedResourceManager(
   "myResources", ".", null);
Console.WriteLine(rm.GetString("TestString"));
```

The first parameter is the name of the resource; the second parameter is the folder where the resources are located. You can use localized files, but note

that the location of these files differs from how `ResourceManager` locates satellite files. If you localize the resources in `myResources` for, say, French spoken in France, the resource file will be called `myResources.fr-FR.resources`. Yet you load this resource using the same code I showed earlier (assuming that the UI culture of the thread is `fr-FR`). Because the culture is part of a resource file's name, you do not need to place the file in a separate localized folder, as you do with satellites.

The final parameter passed to `CreateFileBasedResourceManager()` is the type of the resource set (identified in `ResourceSet`) that will be used. In this case I have used `null`, which indicates that `System.Resources.ResourceSet` should be used. `ResourceManager` uses the resource set to read the resources from the resource file (the type of the resource set that this class uses is accessed through its `ResourceSetType` property). A resource set has a resource reader to do the actual reading; this reader is accessed through the resource set's `Reader` field. You create your own resource set class so that you can use resources that are held in a format other then the compiled format produced by `resgen`.

Resource sets contain only the resources for a specific culture. You can create a resource set through its constructor (by passing a resource stream or the name of a resource file), or you can obtain it through `ResourceManager` by calling `GetResourceSet()` and pass a `CultureInfo` object. Because resource sets are specific to a culture, there is no "fallback" to a neutral culture if the specified culture does not exist. When you create a resource set, it will load all the resources and cache them in a hash table.

In addition to classes for accessing `resgen`-compiled resources, the `System.Resources` namespace has classes for reading and writing `.resx` XML files: `ResXResourceReader` and `ResXResourceWriter`, respectively. It also has an implementation of `ResourceSet` called `ResXResourceSet`.

8.3.8 Satellite Assemblies

As the name suggests, a satellite assembly is separate from the assembly that will use its resources. Do not confuse satellite assemblies with modules. Modules are constituent parts of an assembly and hence are subject to the version-

ing of the assembly to which they belong. A satellite assembly is an assembly in its own right, but unlike normal assemblies, it does not have code and hence does not have an entry point. To create a satellite assembly, you use the assembly builder tool, `al.exe`. For example, imagine that you have resources localized to German in a resource file called `App.de.resources`. This file is embedded into a satellite assembly as a result of the following command line:

```
al /t:lib /embed:App.de.resources
   /culture:de /out:App.resources.dll
```

This command creates a library assembly called `App.resources.dll` localized to German. If you choose, you can create an empty code file with the `[AssemblyVersion]` attribute to give the satellite assembly a version:

```
// C#, file: ver.cs
[ assembly:System.Reflection
     .AssemblyVersion("1.0.0.1") ]
```

The assembly is now compiled with the following:

```
csc /t:module ver.cs
al /t:lib /embed:App.de.resources
   /c:de /out:App.resources.dll ver.netmodule
```

The assembly is still resource-only because the module that is linked in has only metadata. You could do the same thing with the `[AssemblyCompany]` and `[AssemblyDescription]` attributes to add information about the company that created the assembly and a description. The problem with this approach is that there are now two files to deploy: `App.resources.dll` and `ver.netmodule`. To get around this problem, the assembly builder tool allows you to pass some of this information through command-line switches, which are listed in Table 8.3.

Using the `/version` switch, you can tell the assembly builder to specify the version of the assembly. In the absence of other version switches (`/fileversion`, `/productversion`), the version you specify will be used to provide a Win32 `FILEVERSION` resource in the library and will be the basis of the `PRODUCTVERSION` and `FILEVERSION` fields.

Table 8.3 Assembly builder switches used to change the assembly's metadata

Switch	Attribute Equivalent	Description
`/company`	`[AssemblyCompany]`	The company that created the assembly
`/configuration`	`[AssemblyConfiguration]`	Typically Retail or Debug
`/copyright`	`[AssemblyCopyright]`	Your copyright notice
`/culture`	`[AssemblyCulture]`	The culture of the assembly
`/delaysign`	`[AssemblyDelaySign]`	Specification of whether the assembly can be signed later by `sn.exe`
`/description`	`[AssemblyDescription]`	A description of the assembly
`/fileversion`	`[AssemblyFileVersion]`	The Win32 version of the library
`/keyfile`	`[AssemblyKeyFile]`	The name of the file with the key
`/keyname`	`[AssemblyKeyName]`	The name of the key in a cryptographic container
`/product`	`[AssemblyProduct]`	The product's name
`/productversion`	`[AssemblyInformationalVersion]`	The version of the product
`/title`	`[AssemblyTitle]`	The friendly name of the assembly
`/trademark`	`[AssemblyTrademark]`	Your trademark
`/version`	`[AssemblyVersion]`	The assembly version

Now imagine that you have resources for the same application localized to French in `App.fr.resources`. This data is embedded into a satellite assembly by the following command line:

```
al /t:lib /embed:App.fr.resources
   /culture:fr /out:App.resources.dll
```

This command also creates a library assembly called `App.resources.dll`. The code in `ResourceManager` does not use the name of the assembly to determine its culture; instead it uses the `[AssemblyCulture]` attribute (the `.locale` metadata) to locate the correct satellite assembly. Because the satellites have the same name, they should be installed in subfolders of the folder containing the assembly that uses the satellite. These folders should have the name of the locale of the satellite; for example, the German resources should be in a folder called `de`, and the French resources should be in a folder called `fr`.

If your satellite files are to be shared by several applications, you should install the satellites in the GAC (global assembly cache). If you do this, the satellites should have a strong name. Remember that the full name of an assembly includes its culture, version, and public key, so there are no problems with installing several satellite files in the GAC because although the short name of the assembly will be the same, the full names will differ by the culture element.

When a `ResourceManager` object is created and tries to locate localized resources, the runtime first looks in the GAC for the satellite assembly with the correct culture and checks whether it has the resource. If this check fails, the current folder is checked for the culture-specific assembly in a named folder. If this search fails, the runtime starts the search again, but this time for an assembly that has the appropriate "fallback" culture—first in the GAC, and then in the current directory. Each culture will have a fallback culture that will be searched in this way, until finally the runtime will attempt to locate the resource in the default resources for the assembly, which will be in the main assembly. If this search fails, the resource cannot be found and an exception will be thrown.

Because satellite assemblies can have a different version from the version of the main assembly, satellite versions can get out of sync with the main assembly. To get around this problem, the main assembly can specify a *base version* of the satellite assemblies that it uses; it does this with an assembly-level attribute:

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

Unlike versions applied through `[AssemblyVersion]`, the string version must have all four parts. The satellite assembly can be versioned independently from the main version, and the changes can be reflected in the application's configuration file or, if it is installed in the GAC, through a publisher policy file.

The name that I have used for the satellite assembly is `<assem>.resources.dll`. This is a standard naming convention and is vital to how the `ResourceManager` class works. The `<assem>` part is the name of the assembly that will use these resources. This is the only mechanism that exists to tie a satellite to the assembly with which it is used.[2]

If you use satellite assemblies, I urge you to make sure that you also provide locale-neutral resources. A locale-neutral resource is named without a locale and is bound to the assembly that uses the resource. In the preceding example, the main assembly will be compiled with a command line that looks like this:

```
csc /res:App.resources /out:App.exe app.cs
```

The assembly is called `App.exe`. When it is run, `ResourceManager` will check for an appropriate resource for the current culture within the satellite assemblies, and if that resource is not present, it will load the locale-neutral resource from the main assembly. Locating satellite assemblies is not part of Fusion's work, so if `ResourceManager` cannot find a satellite assembly, there will be no binding-error message in the Fusion log (viewable with `FusLogVW.exe`), and if your main assembly has a locale-neutral resource, you'll have no indication that there has been a problem. (If you do not have locale-neutral resources, `ResourceManager` will throw a `MissingManifestResourceException` exception.)

8.3.9 The Event Log, Again

So I am back to the event log again. As I mentioned earlier in this book, .NET has a poor implementation of the classes to write messages to the event log. The principal reason I say this is that these classes put the onus on the user of the

2. If you are careful to name the satellite assemblies and folders correctly, then loading locale-specific resources is straightforward. However, because the location of the correct assembly is so dependent on these names, I regard it as quite a fragile mechanism.

`EventLog` class to localize the messages that are written to the event log rather than taking the correct approach, which is to put the onus on the reader of the event log. If the writer is responsible for localizing messages, the messages can be read in only one locale, which is fine if your distributed application runs in only one locale. In these days of globalization, however, your application could conceivably have components running in different locales, and if a message is localized when it is generated, it ties that message to that locale. Unfortunately, there is little one can do with the current framework classes, and one can only hope that this horrible throwback to the broken event log classes that were present in VB6 will be fixed in a later version of the Framework Class Library.

To localize event log messages, your code merely creates localized format strings in a resource file and then at runtime uses a resource manager to load the appropriate string:

```
// C#
ResourceManager rm
   = new ResourceManager(typeof(myForm));
string errMsg = String.Format(
   rm.GetString("errNoFile"), strFileName);
EventLog el = new EventLog("Application");
el.WriteEntry(errMsg);
```

8.3.10 Win32 Resources

Assemblies are PE files, so they can have Win32 resources. For a C++ developer this is not a problem because managed C++ projects use the standard linker, which will link a compiled Win32 (`.res`) file into a PE file. Indeed, as I mentioned earlier, when you add a resource to a managed C++ project, that resource will be a Win32 resource and not a .NET resource. C# developers can also include Win32 resources using either the C# compiler (`csc.exe`) or the assembly builder tool (`al.exe`). Both of these tools have a `/win32res` and a `/win32icon` switch, the first of which allows you to add an already compiled resource to an assembly. It is probably best to do this within a C++ project, where you can edit an `.rc` file and use the **Resource View** window to add and edit the resources. The unmanaged resource compiler, `rc.exe`, is used to compile a `.rc` file into a `.res` file.

One type of resource that can be described in an `.rc` file is the icon. The first icon in a file's resources will be used by Windows Explorer when displaying the file, and if the file is an executable, this icon will be the application icon.

The `/win32icon` switch is the only way that you can set the icon for an assembly. This switch takes the name of the icon (`.ico`) file; you do not compile it. It is prudent to add both a 32×32 bit image and a 16×16 bit image so that you can determine the icon that will be shown in Windows Explorer, no matter what view the user chooses.

.NET code does not understand Win32 resources, so if you need to read Win32 resources, you have to resort to interop through Platform Invoke. The code is straightforward, and there is even a sample in the Framework SDK samples (called `TlbGen`) that shows how to add a Win32 resource to an assembly programmatically, using the Win32 resource APIs.

8.4 Summary

Visual Studio.NET allows you to create applications, as well as controls and components that can be used as part of applications. Components are objects that can have a site and can be disposed of. The **Toolbox** window contains components. Controls are components that are derived from the `Control` class and have a user interface. Some controls are composites of other controls; these are derived from `UserControl`. The VS.NET IDE allows you to develop `UserControl` objects and `Form` objects with the designer, which lets you construct a user interface by dragging and dropping components and controls from the **Toolbox** window and then using the **Properties** window to provide property values and events. The designer also allows you to develop components and controls, but it does not allow you to develop the user interface of a `Control` class.

Applications and `UserControl` classes can be localized so that they can be used in different locales. Localization involves creating resources for each locale, and these resources can be stored in satellite assemblies. The IDE makes localization straightforward through the **Properties** window; you simply have to set the `Language` property, and then all the properties that you set will be stored in an `.resx` file for the locale. The IDE will build the satellite assemblies for each locale's `.resx` file in your project.