

*Getting the Most Out of the Visual Studio .NET Environment*



*Mastering*

# Visual Studio .NET

O'REILLY®

*Jon Flanders, Ian Griffiths & Chris Sells*

---

# Mastering Visual Studio .NET

---

# Mastering Visual Studio .NET

*Ian Griffiths, Jon Flanders, and Chris Sells*

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

---

# Integrating Components with Visual Studio .NET

Visual Studio .NET presents a great deal of information about controls and other components you use in the development environment. When you drag a component from the toolbox into your project, VS.NET appears to know everything about it—the events and properties it supports are displayed in the property panel, neatly categorized, with a short description available for each member. Some controls have their own unique interactive editing features. Many add extra items to Visual Studio .NET’s menus.

You might suspect that this level of extensive and often highly specialized support is something that is available only for the built-in controls, but that is not the case. Visual Studio .NET has a very open architecture for allowing components to customize the way in which they integrate with the environment.

## Basic Integration

To build components that exploit Visual Studio .NET’s integration facilities, you must understand the basic mechanisms involved. Component integration relies heavily on the .NET runtime’s reflection mechanism—the facility that allows type information to be examined at runtime. VS.NET uses reflection to discover what properties and events your component provides.



Strictly speaking, Visual Studio .NET doesn’t use reflection directly. It uses the `TypeDescriptor` class and its friends in the `System.ComponentModel` namespace. These provide a virtualized view of type information, which allows a component’s properties to be extended dynamically. The `TypeDescriptor` API is implemented using the reflection API however.

One of the advantages of this reflection-based approach is that components get a great deal for free. All of their public properties and events will be detected automatically. So a component as simple as that shown in Example 7-1 can participate fully in visual editing in a Visual Studio .NET project.

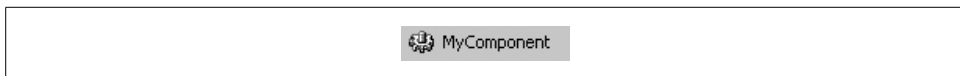
*Example 7-1. A very simple component*

```
using System.ComponentModel;

public class MyComponent : Component
{
    public string Title
    {
        get { return myTitle; }
        set { myTitle = value; }
    }
    private string myTitle;
}
```

If you compile the code in Example 7-1 into a class library, you can drag the compiled DLL from Windows Explorer onto a toolbox. (Alternatively, you can add it by right-clicking on the toolbox, selecting *Customize Toolbox...*, choosing the *.NET Framework Components* tab, clicking the *Browse...* button, and locating your component. This will have the same effect but is considerably more long-winded than the drag-and-drop approach.)

When you add a DLL to a toolbox, Visual Studio .NET searches it for classes that implement *System.ComponentModel.IComponent* and will add an entry to the toolbox for each such class that it finds. This includes all classes that derive from *ComponentModel*—it implements *IComponent*. If the DLL just contains the class shown in Example 7-1, the toolbox will grow one extra entry, as shown in Figure 7-1. (The cog icon is the default graphic used when a component does not provide its own bitmap. We will see how to supply a custom bitmap later.)



*Figure 7-1. A newly added toolbox item*

You will now be able to drag this component onto Visual Studio .NET design windows just like any other component. Because it derives directly from *Component* (and not the *Windows Forms* or *Web Forms Control* classes), it will appear in the component tray of any form you add it to, rather than on the form itself. This makes perfect sense—we didn't write a visual component, so it would have no business appearing on the form itself.

Once you have added an instance of your component to the component tray, you can select it and edit its properties by displaying the *Properties* window, just as you can for any built-in component. Figure 7-2 shows the *Properties* window for this component.

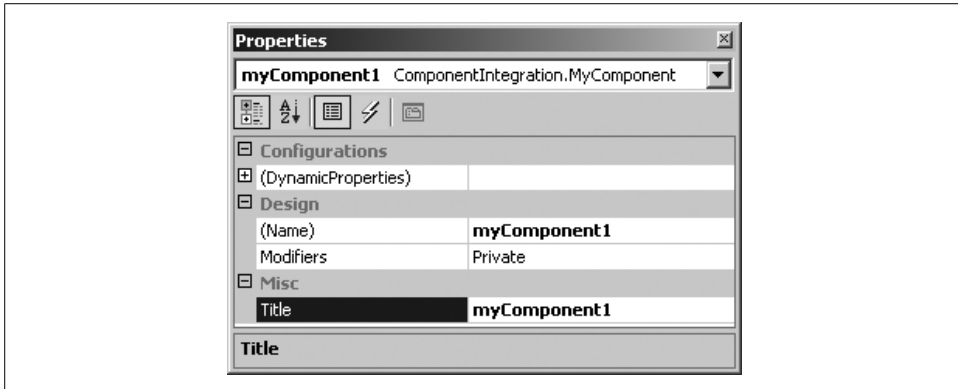
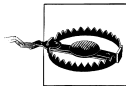


Figure 7-2. A custom component in the Properties window

Notice that the one public property defined by our class—`Title`—has appeared in the Properties window, along with the standard pseudo properties that the designer always adds. (The `(Name)` entry simply determines the name of the designer-generated field that will hold a reference to the component. Note that if you add a property called `Name`, this can lead to confusion, as VS.NET tends to want the `Name` property to be the same as the name of the field that holds the object. The `Modifiers` property determines that field’s protection level.) If the component has any public events, and you are using it in a C# project, the lightning bolt button will appear at the top of the window, allowing us to browse the events instead of the properties. (For a Visual Basic .NET project, the events will instead be listed at the top of the source editor window.) Our property has appeared in the default `Misc` category, but we will see how to change that shortly.



If you are trying things out as you read this, be aware that Visual Studio .NET appears to cache information about components. This is reasonable, since most of the time components do not change while you are using them, but it is slightly inconvenient if you are developing a component’s design-time features. If you modify your component (e.g., you add an event), you may need to do the following steps to make changes visible in the client project::

- Delete all instances of the component in the client project.
- Remove the reference to the component from the client project (in the project’s `References` section in the Solution Explorer).
- Add the component back into the client project.

These first three steps are usually sufficient, but if they do not work, try performing these extra steps before adding the component back to the project:

- Delete the component from the toolbox.
- Recompile the component.
- Add the component back to the toolbox.

This reflection-based designer integration is great because it requires no effort to get a component working in Visual Studio .NET. However, this is all pretty bare-bones stuff. Even for an extremely simple component such as the one in Example 7-1, we are missing basic features, such as property categorization and description. Fortunately, .NET's reflection mechanism is extensible—types and their members can be annotated with custom attributes. Visual Studio .NET exploits this extensibility, and many of its integration features can be harnessed simply by adding the right attributes to your components.

## Simple Integration Attributes

There are certain custom attributes that Visual Studio .NET will look for when you use a component. These allow you to improve your component's integration with the development environment, often without having to write any additional code at all. (Some of the more advanced integration features require both attributes and code, but we shall look at those later.)

### Toolbox Bitmap

One of the first things you will want to do to your component is to change its toolbox bitmap. By default, all components get the cog icon shown in Figure 7-1, and there is no way for the end user to change it. It would be hard to locate specific items in a full toolbox if they all had the same bitmap, so it is best to supply a graphic for your component. You can specify a custom toolbox bitmap by applying the `ToolboxBitmap` attribute (defined, inexplicably, in the `System.Drawing` namespace) to your class.

The `ToolboxBitmap` attribute has three constructors. The first takes a string specifying the name of the file containing the bitmap. You will not normally use this, since it requires the bitmap to be stored in a separate file. It is much more convenient for a component to be a single self-contained file, so you will want to use one of the other constructors—both of these expect the bitmap to be an embedded resource in the component (see the “Embedded Resources” sidebar). The most commonly used constructor is the one that takes just a type object; its use is illustrated in Example 7-2.

## Embedded Resources

Many applications and components require resources, such as bitmaps, icons, and mouse cursors. Although such items can be stored in separate files, this is inconvenient as it means that the component will no longer be self-contained: all of the associated resource files would need to be distributed along with the component in order to make it work.

Windows has long had a solution to this problem. The PE file format (the format used by all DLL and EXE files) allows arbitrary byte streams, as well as the usual executable code and data, to be embedded in the file. Most applications embed icons, bitmaps, and the like using this technique.

.NET provides its own solution, which is similar but different. *Assembly manifest resources* are conceptually similar to PE file resources—they are just byte streams stored within the file. The main difference is that, whereas PE file resources are identified by numbers, assembly manifest resources have names (e.g., *MyApplication.MyPicture.bmp*).

Because .NET components are compiled into DLL or EXE files, it is technically possible for a single file to contain both types of resources. However, because most .NET applications don't need to use PE file resources, Visual Studio .NET 2003 allows only a single PE file resource to be added to any given component—the one that will be used for the file icon. (This is the *App.ico* item that most .NET projects have. You can change which file will be used as the icon in the Project Property Pages—this property is in the Common Properties → General page, under the Application category.)

PE file resources, which are supported for only non-.NET projects, can be added using Add Resource... in the project context menu in the Solution Explorer.

Assembly manifest resources can be added to a component in a Visual Studio .NET project by selecting the item in the Solution Explorer and using the Properties window to set its Build Action to Embedded Resource.

In a Windows Forms application, each form will have its own *.resx* file containing resources associated with the form. Each *.resx* file can contain many resources, but it will be compiled into the component as a single assembly manifest resource. (The .NET ResourceManager class knows how to extract the individual resources from such a container.)

### Example 7-2. The *ToolboxBitmap* attribute

```
using System.Drawing;
using System.ComponentModel;

[ToolboxBitmap(typeof(MyComponent))]
public class MyComponent : Component
{
    . . . as before
}
```

By convention, the type object passed to the `ToolboxBitmap` attribute is the component's type, but this is not a requirement. The type object serves two roles here. First, Visual Studio .NET will look for the bitmap resource in the assembly in which the type is defined. (Although in principle you could use a bitmap defined in an external assembly by referring to a type defined in that assembly, in practice you will always want this to be the same assembly as the component itself.) Second, the resource name will be based on the fully qualified name of the type. So if `MyComponent` in Example 7-2 is defined in the `MyNamespace` namespace, Visual Studio .NET will look for an embedded resource called `MyNamespace.MyComponent.bmp`. (In other words, it takes the fully qualified name of the type and appends `.bmp`.)

For this to be of use, you must make sure that your component contains an embedded bitmap resource with the right name. To do this, first add a bitmap to the project. (Select `Add New Item...` from the project's context menu in the Solution Explorer and choose `Bitmap File` from the `Resources` category.) Your bitmap should have the same name as the component class. Visual Studio .NET automatically prepends the project default namespace to the bitmap when embedding it as a resource, so you do not need to supply the fully qualified name. In our example, the bitmap filename would be `MyComponent.bmp`. Your bitmap should be 16x16 pixels. Visual Studio .NET will look at the color of the bottom-left pixel and will draw all pixels with that color as transparent.

By default, bitmaps do not get compiled into projects, so simply adding a bitmap to the project is not enough. You must change the bitmap's `Build Action` to be `Embedded Resource`. You can do this from the bitmap project item's `Properties` page, shown in Figure 7-3. Note that bitmaps have two different property pages: the one that appears when you are editing the bitmap and the one that appears when you select the bitmap item in the Solution Explorer. The `Build Action` is located in the latter.

Note that the filenames are, somewhat unusually, case sensitive. This is because the mechanism by which the bitmap resource is retrieved from the component is case sensitive. (`Assembly.GetManifestResourceStream` is used.) So you must make sure that the bitmap name's case matches that of your class exactly and that the `.bmp` extension is all lowercase.



Visual Studio .NET always prepends the project's default namespace to an embedded resource. (The project's default namespace can be set in the project property pages. Right-click on the project in the Solution Explorer, and in the Properties window that appears, select the Common Properties → General tab. The Default Namespace property is in the Application category.) If the resource file is in a subdirectory within the project, VS.NET will also add the folder name between the namespace and the filename. So if a bitmap called *Picture.bmp* is in a folder called *SomeFolder*, the embedded resource will be named *MyNamespace.SomeFolder.Picture.bmp*.

There is no way to disable this behavior. This means that if you want to embed a resource whose name does not start with the default namespace, your only option is to give the project a blank default namespace.

There is a third constructor for the `ToolboxBitmap` attribute, which takes both a type reference and a string. The documentation is a little misleading here, as it suggests that the type reference is used only to determine which assembly the name is in and that you can specify the name of the resource with the string. This is not quite true—the type object is used in two ways. The embedded resource name is formed by taking the namespace of the type and appending the string supplied. So if you were to supply the following reasonable-looking parameters to the custom attribute:

```
[ToolboxBitmap(typeof(MyComponent), "MyNamespace.MyComponent.bmp")]
```

Visual Studio .NET would look for an embedded resource called *MyNamespace.MyComponent.bmp*!

Sadly, just as there is no way to force Visual Studio .NET to compile in an embedded bitmap with the exact name that you require, there is also no way to specify the precise name of the embedded resource to use with the `ToolboxBitmap` attribute. In both cases, the namespace will be added, and there is nothing you can do about it. So you should stick to the following rules:

- Name the bitmap resource after the unqualified class name
- Define your component in the project default namespace

Since the Visual Studio .NET wizards will always add new components into the project default namespace, it is easy to stick to these rules in practice.

## Categories and Descriptions

When Visual Studio .NET displays properties and events in the Properties window, it usually provides two hints as to their use: it groups members by category, and it provides a short textual description when an item is selected. By default, your components' members will appear in the Misc category and will have an empty description, but it is easy to fix this.

You can use the `Category` attribute (defined in the `System.ComponentModel` namespace) to determine the category in which members appear in the Properties window. The attribute just takes a string, which is the name of the category. You can use whatever you like as a string, but it is recommended that you use one of the standard categories if possible. (These are `Action`, `Appearance`, `Behavior`, `Data`, `Design`, `DragDrop`, `Focus`, `Format`, `Key`, `Layout`, `Mouse`, and `WindowStyle`.)

The `Description` attribute is also very simple to use. As with the `Category` attribute, it just takes a string. This string will be displayed in the Description pane of the Properties window when the property is selected.

Example 7-3 shows both the `Category` and the `Description` attributes in use on the `Title` property of our component. With these in place, the Properties window will look like Figure 7-3.

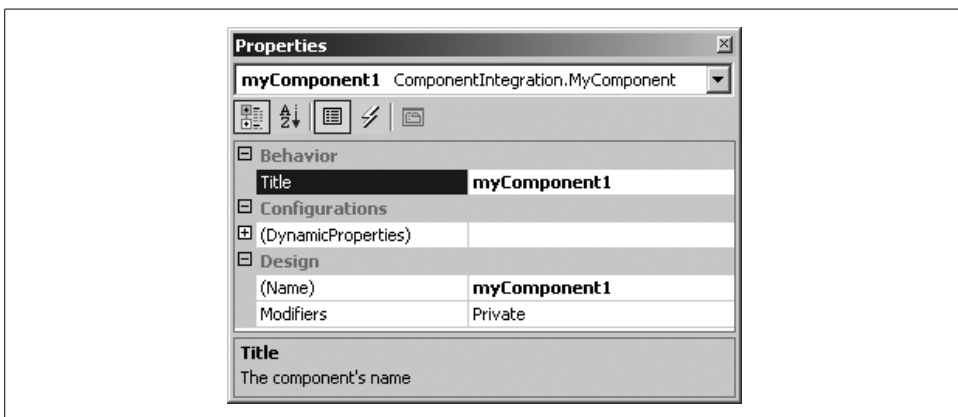


Figure 7-3. Category and description in the Properties window

Example 7-3. A property with a category and a description

```
using System.ComponentModel;
```

```
...
```

```
[Category("Behavior")]  
[Description("The component's name")]  
public string Title  
{  
    ... as before  
}
```

## Localization

Both the `Category` and the `Description` attributes cause text to be displayed in Visual Studio .NET's user interface. If your component might be used at design time in different countries, this presents a localization issue—how do you ensure that the category and description strings are appropriate to the locale?

With the `Category` attribute, life is very easy if you stick to the built-in category names (listed earlier). Visual Studio .NET recognizes these names and will translate them for you automatically. The `Description` attribute presents more of a challenge. (As does using nonstandard category names.)

If you want your description strings to be localized, you must create your own attribute class that derives from the `Description` attribute, overriding its `Description` property to perform the appropriate lookup. (You would normally use the `ResourceManager` class in the `System.Resources` namespace to look the name up in the appropriate satellite resource assembly.)

To make custom category names localizable, you use a similar technique—you create your own class that derives from the `Category` attribute. For some reason, instead of overriding the `Description` property, you are required to override a protected method called `GetLocalizedString` and look up the translated resource there; the `Category` attribute class will call this when translation is required.

## Default Events and Properties

With most of the .NET Framework's built-in components, double-clicking on them will cause an event handler to be added to your code. (This is true of all components that raise events, not just controls.) With components that raise multiple events, Visual Studio .NET always seems to know which event handler to add—for a button it will handle the `Click` event, for a text box it will handle `TextChanged`, and so on. And likewise, if you drag a new component onto a form and just type in some data without first selecting a property in the grid, it will usually pick a sensible property to modify (e.g., `Text` for most controls).

For our own components, we can determine which event and property Visual Studio .NET will choose under these circumstances. There are attributes for choosing a default property and event. These are applied to the class and simply take the name of the relevant member as a construction parameter, as shown in Example 7-4.

*Example 7-4. A component with a default event and property*

```
[DefaultEvent("OnTitleChanged")]
[DefaultProperty("Title")]
public class MyComponent : Component
{
```

## Property Visibility

Sometimes your components may have properties that you will not wish to be displayed in the Properties window. This is particularly common with controls—the base control classes in Windows Forms and Web Forms provide many standard properties, not all of which make sense in derived controls. (For example, the `Panel` control does not use the `Text` property.) Also, some properties are designed to be

used only from code, such as the Windows Forms Control class's Created property, and it would be confusing and unhelpful for them to appear in the property grid.

To prevent such properties from appearing, you can mark them with the `Browsable` attribute. This takes a Boolean; specifying `false` prevents the property from appearing in the Properties window. (If you are using this to hide an unused property inherited from the base class, you will need to override that property in order to use this attribute. If the only reason you are overriding the property is to apply an attribute, you should just defer to the base class in the implementation, as Example 7-5 does.)

*Example 7-5. A nonbrowsable property*

```
[Browsable(false)]  
public override string Text  
{ get { return base.Text; } set { base.Text = value; } }
```

Although the property in Example 7-5 will not appear in the Properties window, it will still show up in IntelliSense in source editing windows. If you wish to prevent it from appearing even in source windows, you can apply the `EditorBrowsable` attribute. If you pass the `EditorBrowsableState.Never` enumeration value to the constructor, the member will not appear in IntelliSense lists. (Developers who know the property is there will still be able to use it however—the compiler itself ignores this attribute.) If you do not supply an `EditorBrowsable` attribute, the effective default is `EditorBrowsable.Always`. There is also an `EditorBrowsable.Advanced` setting, which is supposed to hide the property for all but advanced users. By default, this hides the property in Visual Basic .NET projects but does not hide it in C# projects. (See Appendix F for information on how to change this and other text editor settings.)

## Designer Serialization

When users change your component's properties in the Properties window, Visual Studio .NET generates code that will set the property at runtime. (It does this in the autogenerated `InitializeComponent` method; it effectively serializes the properties as code.) Of course, you will want code to be generated only when the property has actually been changed. Visual Studio .NET relies on knowing what your property's default value is to work out whether the value has been changed. (It doesn't just remember what the value was before the user started making edits.)

You can tell Visual Studio .NET what a property's default value is by applying the `DefaultValue` attribute. This has a wide array of constructors—most of the intrinsic types get their own constructor, and there is also one that takes an object, allowing you to pass any value at all. When Visual Studio .NET generates the `InitializeComponent` method, it will compare your component's property's current value to the default value, and generate initialization code only if they differ.

Some properties' default values are determined at runtime. For example, a `Control` object's default `BackColor` property value is determined by its parent. Under these circumstances, a `DefaultValue` attribute cannot be used. Instead, the property should have an associated `ShouldSerialize` method. (For example, if the property in question is called `Title`, then there should be a `ShouldSerializeTitle` method.) This method should return a `Boolean`, indicating whether the property has been set, and the designer therefore needs to generate code to serialize this property. If you supply a `ShouldSerialize` method, you should also supply a corresponding `Reset` method (e.g., `ResetTitle`). Visual Studio .NET will use this when the user selects the `Reset` item from the property's context menu. This method should cause the property to return to its original state (i.e., the value should revert to the dynamically determined default, and the `ShouldSerialize` method should return `false`).

You can disable designer serialization entirely if necessary. (For example, it would not be worth serializing a property whose value is calculated from other properties at runtime.) The `DesignerSerializationVisibility` attribute allows you to control what code is generated at serialization time. If you construct the attribute with the `DesignerSerializationVisibility.Hidden` enumeration member, the property will never have any code generated for it in the `InitializeComponent` method. The default setting is the `Visible` enumeration member. There is also a `Content` member, which indicates that the designer should enumerate the property's contents, rather than trying to serialize the whole property in one step. You would normally do this only if the property's type does not serialize correctly, but each of its individual member properties can be serialized. (For example, if you have a custom type that has no corresponding `TypeConverter`, Visual Studio .NET will not know how to generate code to serialize properties of this type. But if this custom type's own properties are all of standard types, the `Content` setting would cause Visual Studio .NET to generate code to serialize each of these individually.)

## Data Binding

If you are writing a control, you may wish for certain properties to be presented under the (`DataBindings`) section of the Properties window—this is where Visual Studio .NET allows data bindings to be configured interactively. Programmatically, any control property can be bound to a data source, but only those explicitly marked as bindable will appear under (`DataBindings`).

By default, the `Control` class's `Tag` and `Text` properties are bindable, but you can add your own with the `Bindable` attribute. Simply mark any property that you want to appear in the data binding section with this attribute, passing in `true` as a constructor parameter.

# Custom Property Types

The Visual Studio .NET property grid (which, incidentally, is available for use in your own applications as the `System.Windows.Forms.PropertyGrid` control) is able to deal with a wide range of different property types. It can supply appropriately specialized user interfaces for the types commonly used for control properties, such as `Color` and `Size`. But what if your component has a property of some custom type?

Even with custom types, the property grid can display the value of your property. In the absence of other information, it will simply call the `ToString` method and display the results. However, the property will be grayed out, so users will not be able to edit it. Also, `ToString` may not produce the desired result—by default, this simply returns the name of the type.

You can enable editing of properties with custom types in two ways. Both involve writing special support classes—you cannot support custom types with attributes alone. You can enable full text-based editing by supplying a custom *type converter*. You can also provide a graphical user interface for editing the property by writing a custom *UI type editor*.

## Type Converters

A type converter is a class derived from `TypeConverter`, which is defined in the `System.ComponentModel` namespace. (Despite the similar name, this class is in no way connected to the `System.Convert` class.) Its job is to convert between types, usually between a custom type and a string. If a custom type has an associated type converter, Visual Studio .NET will use that to convert properties of that type to strings in the property grid. And if the user modifies the properties in the grid, the type converter will be used to convert the modified strings back to property values. The framework class libraries supply type converters for many widely used types, such as `Point` and `Rectangle`. You will usually need to supply converters only for your own custom types.

We tell Visual Studio .NET that a custom type converter is available with the `TypeConverter` attribute. This attribute can be applied either to the custom type itself or to the property itself, shown respectively in Example 7-6 and Example 7-7. A converter specified for a property will take precedence over one specified for the type. So, although `ThreeDPoint` is associated with the `MyThreeDPointConverter` type converter in Example 7-6, the property in Example 7-7 has elected to use the `ExpandableObjectConverter` type converter instead.

*Example 7-6. Associating a type converter with a type*

```
[TypeConverter(typeof(MyThreeDPointConverter))]  
public class ThreeDPoint  
{
```

*Example 7-6. Associating a type converter with a type (continued)*

```
public ThreeDPoint(int x, int y, int z)
{
    this.x = x; this.y = y; this.z = z;
}

public ThreeDPoint()
{
}

public int X { get { return x; } set { x = value; } }
public int Y { get { return y; } set { y = value; } }
public int Z { get { return z; } set { z = value; } }
private int x, y, z;
}
```

*Example 7-7. Associating a type converter with a property*

```
public class HasPoint
{
    [TypeConverter(typeof(ExpandableObjectConverter))]
    public ThreeDPoint P1 { get { return p1; } set { p1 = value; } }
    private ThreeDPoint p1;
}
```

The type converter itself is just a class derived from `TypeConverter`. We typically overload four methods. Visual Studio .NET uses two of these to discover which conversions we support. It will call `CanConvertTo` to discover if we can convert to a particular type and `CanConvertFrom` to see if we can transform a particular type into the custom type. These are called when a property is displayed in the property grid. In both cases, VS.NET asks about support for conversion to and from strings.

The other two methods we overload are `ConvertTo` and `ConvertFrom`. These are called when Visual Studio .NET needs to perform a conversion. `ConvertTo` will be called (with a target type of string) when the property grid is being displayed. `ConvertFrom` is called (with a source type of string) when the user edits a property in the grid.

Example 7-8 shows a sample type converter for the three-dimensional point class, shown in Example 7-6. Its `CanConvertTo` and `CanConvertFrom` methods support conversions to and from strings.

*Example 7-8. A type converter*

```
using System;
using System.ComponentModel;

public class MyThreeDPointConverter : TypeConverter
{
    public override bool CanConvertTo(ITypeDescriptorContext context,
        Type destinationType)
    {
        if (destinationType == typeof(string)) return true;
    }
}
```

Example 7-8. A type converter (continued)

```
        return base.CanConvertTo(context, destinationType);
    }

    public override bool CanConvertFrom(ITypeDescriptorContext context,
        Type sourceType)
    {
        if (sourceType == typeof(string)) return true;
        return base.CanConvertFrom(context, sourceType);
    }

    public override object ConvertTo(ITypeDescriptorContext context,
        System.Globalization.CultureInfo culture, object value,
        Type destinationType)
    {
        if (destinationType == typeof(string))
        {
            {
                ThreeDPoint point = (ThreeDPoint) value;
                return string.Format("{0},{1},{2}", point.X, point.Y, point.Z);
            }
        }
        return base.ConvertTo(context, culture, value, destinationType);
    }

    public override object ConvertFrom(ITypeDescriptorContext context,
        System.Globalization.CultureInfo culture, object value)
    {
        if (value.GetType() == typeof(string))
        {
            {
                string src = (string) value;
                string[] points = src.Split(',');
                if (points.Length != 3)
                    throw new ArgumentException("String must be formatted as 'x,y,z'",
                        "value");

                return new ThreeDPoint(int.Parse(points[0]),
                    int.Parse(points[1]), int.Parse(points[2]));
            }
        }
        return base.ConvertFrom(context, culture, value);
    }
}
```

The actual conversions are done in `ConvertTo` and `ConvertFrom`. They convert the string to and from a comma-separated list of the three coordinate values. Figure 7-4 shows this type converter in action on a property grid. It displays a component with a single property, `Point`, of type `ThreeDPoint`. (The full source code for these types is shown in Example 7-6 and Example 7-7. If you compile this code into a Class Library project, you will then be able to add it to your toolbox using the toolbox context menu's `Customize Toolbox` item: select the `.NET Framework Components` tab and then `Browse` for your component. Alternatively, you can drag your compiled DLL from a Windows Explorer window onto the toolbox.)

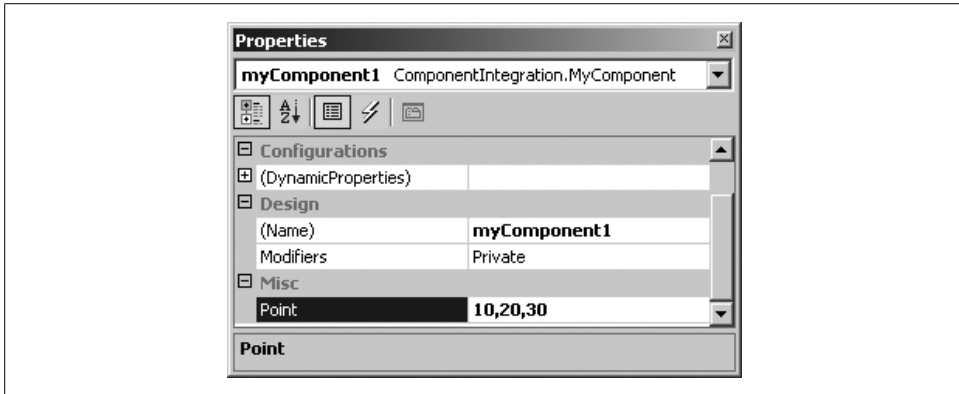


Figure 7-4. A type converter in action



If your type already has a suitable `Tostring` method, you do not need to override `CanConvertTo` and `ConvertTo` simply to support string conversion. `TypeConverter` provides default implementations of these methods that support string conversion by calling `Tostring` on the object. (As you will see shortly, you will normally want to override these methods to support code serialization. But even then, you can still defer to the base class for string conversions unless the type's `Tostring` method does not provide appropriate behavior.)

We can go one better than this and provide the same expandable editing that built-in classes such as `Size` and `Point` have. If we change the type converter in Example 7-8 so that its base class is `ExpandableObjectConverter`, the property grid will display an expandable version of the property, as Figure 7-5 shows.

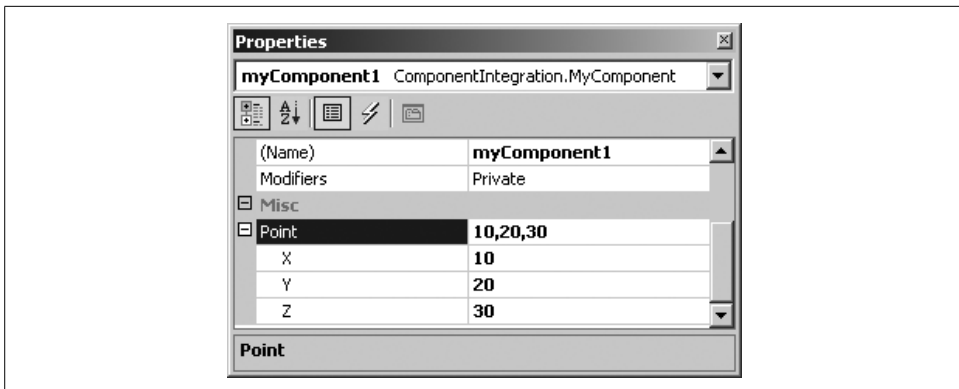


Figure 7-5. An expandable property

Unfortunately, if you try to use this type converter in Visual Studio .NET, you will discover that it has a serious shortcoming. The designer fails to save the edited val-

ues in the `InitializeComponent` method. Every time you reopen the form containing a component that uses this type, the property will have forgotten its value. The reason for this is that Visual Studio .NET does not know how to initialize new instances of our `ThreeDPoint` class. We must tell it how to do this by adding code serialization support to our type converter.

## Code serialization

For Visual Studio .NET to persist properties of a custom type in the `InitializeComponent` method, we must support an extra conversion in our type converter. The `CanConvertTo` and `ConvertTo` methods must support conversion to `InstanceDescriptor` (defined in the `System.ComponentModel.Design.Serialization` namespace).

The `InstanceDescriptor` class encapsulates instructions on how to create an instance of a particular type. We can use it in our type converter to tell Visual Studio .NET how to generate code to create a `ThreeDPoint` object. (We need to supply conversion only *to* `InstanceDescriptor`. Converting *from* an `InstanceDescriptor` back to our type is not needed—Visual Studio .NET just constructs the object according to the instructions in `InstanceDescriptor`.)

Example 7-9 shows the modified `CanConvertTo` and `ConvertTo` methods. When asked to convert to an `InstanceDescriptor`, the converter builds one, supplying a `ConstructorInfo` object (from the `System.Reflection` namespace) to indicate which constructor to use. It also supplies the parameters required by this constructor. With the type converter thus modified, Visual Studio .NET can now generate code for properties of type `ThreeDPoint`.

*Example 7-9. Type converter code serialization support*

```
public override bool CanConvertTo(ITypeDescriptorContext context,
    Type destinationType)
{
    if (destinationType == typeof(InstanceDescriptor)) return true;
    if (destinationType == typeof(string)) return true;
    return base.CanConvertTo(context, destinationType);
}

public override object ConvertTo(ITypeDescriptorContext context,
    System.Globalization.CultureInfo culture, object value,
    Type destinationType)
{
    if (destinationType == typeof(InstanceDescriptor))
    {
        Type[] ctorParamTypes = new Type[]
            { typeof(int), typeof(int), typeof(int) };
        ConstructorInfo ctor = typeof(ThreeDPoint).GetConstructor(ctorParamTypes);

        ThreeDPoint p = (ThreeDPoint) value;
```

*Example 7-9. Type converter code serialization support (continued)*

```
        object[] ctorParams = { p.X, p.Y, p.Z };

        return new InstanceDescriptor(ctor, ctorParams);
    }
    if (destinationType == typeof(string))
    {
        ThreeDPoint point = (ThreeDPoint) value;
        return string.Format("{0},{1},{2}", point.X, point.Y, point.Z);
    }
    return base.ConvertTo(context, culture, value, destinationType);
}
```

Example 7-10 shows some generated code from an `InitializeComponent` method.

*Example 7-10. Code generated based on an `InstanceDescriptor`*

```
//
// componentWith3D1
//
this.componentWith3D1.Point = new ThreeDPoint(10, 20, 30);
```

## Custom UI Type Editors

Visual Studio .NET will use type converters only for text-based property editing and code serialization. Some built-in types, such as `Color` or `DockStyle`, get a specialized user interface in the property grid as well as text support. If you would like to supply a graphical editing interface for your own property types, you can do so by supplying a UI type editor.



Any type or property is allowed to have both a type converter and a UI type editor. Supplying both gives developers who use your controls a choice—they can edit properties either as text or using the custom UI.

A UI type editor is similar to a type converter—it is a class associated with a custom type via an attribute and used by Visual Studio .NET in the property grid. The attribute for a UI type editor is the `Editor` attribute, defined in the `System.ComponentModel` namespace. As with a type converter, you may apply this attribute either to the custom type or to a property itself. If you apply this attribute to a property, the property's type doesn't even need to be a custom type—you can supply a custom editing UI for a built-in type if you want, as Example 7-11 shows.

*Example 7-11. A property with a custom UI type editor*

```
[Editor(typeof(ContrastEditor), typeof(UITypeEditor))]
public int Contrast
{
    get { return myContrast; }
    set { myContrast = value; }
```

*Example 7-11. A property with a custom UI type editor (continued)*

```
}  
private int myContrast;
```

As Example 7-11 shows, the `Editor` attribute requires you to indicate what sort of editor you are specifying as well as the editor's class—it is designed to allow multiple different kinds of editors to be associated with a property or type. In this case, we are specifying `UITypeEditor`. (In fact, with Visual Studio .NET 2003, custom UI type editors are the only kind of editor supported.) UI type editors must derive from the `UITypeEditor` class, which is defined in the `System.Drawing.Design` namespace. (The `ContrastEditor` is a fictional editor. Two possible implementations are shown later in Example 7-12 and Example 7-13.)

When we write the UI editor class itself, we have a choice as to the kind of user interface we can supply. We can either open a modal dialog or supply a pop-up user interface that will appear in the property grid itself. We indicate this by overriding the `GetEditStyle` method. This method returns a value from the `UITypeEditorEditStyle` enumeration, either `Modal` or `DropDown`. For either type of user interface, we must also override the `EditValue` method, which will be called when the user tries to edit the value.

*Example 7-12. A dialog custom UI type editor*

```
using System.Drawing.Design;  
using System.Windows.Forms;  
  
public class ContrastEditor : UITypeEditor  
{  
    public override UITypeEditorEditStyle GetEditStyle(  
        ITypeDescriptorContext context)  
    {  
        return UITypeEditorEditStyle.Modal;  
    }  
  
    public override object EditValue(ITypeDescriptorContext context,  
        IServiceProvider provider, object value)  
    {  
        DialogResult rc = MessageBox.Show("Maximum contrast?",  
            "Contrast", MessageBoxButtons.YesNoCancel);  
        if (rc == DialogResult.Yes)  
            return 100;  
        if (rc == DialogResult.No)  
            return 50;  
        return value;  
    }  
}
```

Example 7-12 shows a custom UI type editor that displays a simple message box. (Any modal dialog would do.) The value returned from `EditValue` will be written back to the property. The property grid indicates that a modal editor is available for

the property by putting a button with a ... label on the grid when the property is selected, as Figure 7-6 shows. It will call `EditValue` when the button is clicked.

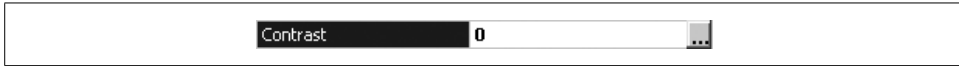


Figure 7-6. A property with a modal custom UI type editor

If you want to provide a drop-down editor user interface (such as the one supplied for the built-in `Color` type), the technique is slightly different. You must get Visual Studio .NET to open the window for you, so that it can be placed and sized correctly. The code for doing this is shown in Example 7-13.

Example 7-13. A drop-down custom UI type editor

```
using System.Drawing.Design;
using System.Windows.Forms;
using System.Windows.Forms.Design;

public class ContrastEditor : UITypeEditor
{
    public override UITypeEditorEditStyle GetEditStyle(
        ITypeDescriptorContext context)
    {
        return UITypeEditorEditStyle.DropDown;
    }

    public override object EditValue(ITypeDescriptorContext context,
        IServiceProvider provider, object value)
    {
        IWindowsFormsEditorService wfes = provider.GetService(
            typeof(IWindowsFormsEditorService)) as
            IWindowsFormsEditorService;
        if (wfes != null)
        {
            TrackBar tb = new TrackBar();
            tb.Minimum = 0;
            tb.Maximum = 100;
            tb.Value = (int) value;
            tb.TickFrequency = 10;
            wfes.DropDownControl(tb);
            value = tb.Value;
        }
        return value;
    }
}
```

This code uses the `IServiceProvider` passed to `EditValue`. It asks it for the `IWindowsFormsEditorService` interface (which is defined in the `System.Windows.Forms.Design` namespace). This service provides the facility for opening a drop-down editor—we simply call the `DropDownControl` method on it, and it will open whichever

control we pass. It sets the size and location of the control so that it appears directly below the property when the drop-down arrow is clicked, as Figure 7-7 shows. (It will modify the control's width to be the same as the property grid's value column, but it will use whatever height you specify. Since we have not set the height in this example, we are simply getting the `TrackBar` control's default height.)

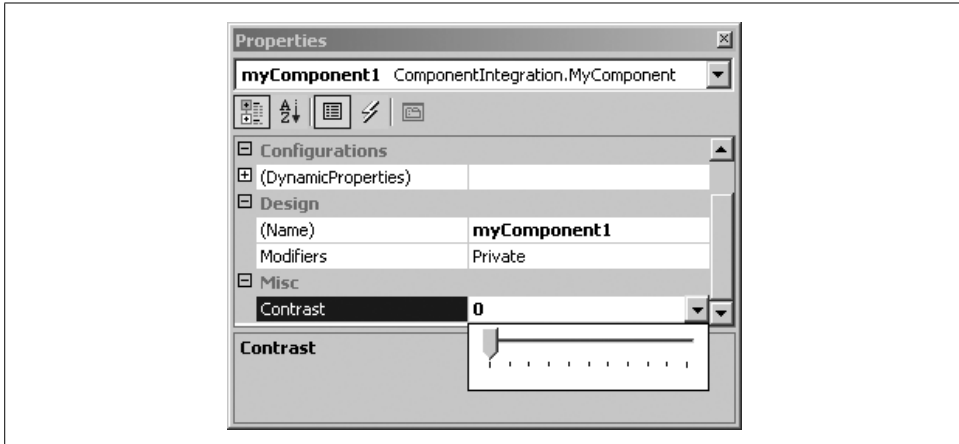


Figure 7-7. A drop-down UI type editor in action



Although this example uses one of the built-in controls, the `TrackBar`, you are free to use any control, including controls of your own devising. It is common practice to create a `UserControl` (a custom control built by composing several other controls) for a drop-down editor.

Custom UI type editors can also add a small graphic to the property grid, which will be displayed in the value field whether the user opens the custom editor or not. Several of the built-in types use this facility. For example, properties of type `Color` always show a small rectangle of the currently selected color in the grid. To supply a similar graphic of your own, you must override two methods: `GetPaintValueSupported` and `PaintValue`.

As Example 7-14 shows, the `GetPaintValueSupported` method is very simple. This will be called when the property is shown in the property grid and we return `true` to indicate that we would like to supply a graphic for the property. Visual Studio .NET will then call the `PaintValue` method, in which we draw the graphical representation of the value. The `PaintValueEventArgs` object supplies a `Graphics` object into which we draw the representation and a `Bounds` rectangle indicating how large the drawing should be.

#### Example 7-14. Adding a value graphic

```
// Add these using statements to Example 7-13.  
using System.Drawing;
```

*Example 7-14. Adding a value graphic (continued)*

```
using System.Drawing.Drawing2D;

// Add these methods to the ContrastEditor class from Example 7-13.
//
public override bool GetPaintValueSupported(ITypeDescriptorContext context)
{
    return true;
}

public override void PaintValue(System.Drawing.Design.PaintValueEventArgs e)
{
    Graphics g = e.Graphics;
    int contrast = (int) e.Value;

    int darkValue = ((100-contrast) * 127) / 100;
    int lightValue = 255 - darkValue;
    Color darkColor = Color.FromArgb(darkValue, darkValue, darkValue);
    Color lightColor = Color.FromArgb(lightValue, lightValue, lightValue);

    using (Brush fill = new LinearGradientBrush(
        e.Bounds, darkColor, lightColor,
        LinearGradientMode.BackwardDiagonal))
    {
        g.FillRectangle(fill, e.Bounds);
    }
}
```



The `Graphics` object supplied to the `PaintValue` method is the same one that the property grid uses to paint itself. This means that you should take care to leave it in the state that you found it. If you change anything such as the transform, or smoothing mode, you should save the state at the start of your method by calling `Save`, and restore it at the end using `Restore`. If you fail to do this, the property grid's appearance may be adversely affected.

Also, note that clip rectangle for the `Graphics` object is not quite set correctly. It is possible to draw slightly outside of the region specified by the `PaintValueEventArgs` object's `Bounds` property. (With the current implementation, you can draw anywhere in the cell showing your property's value.) You should therefore be careful not to draw anything outside of the region specified by `Bounds`.

Example 7-14 simply fills the available space with a rectangle painted with a gradient fill. When the property (which in this case is the `Contrast` property from Example 7-11) is at 100%, the fill will be high-contrast, ranging from black to white, as Figure 7-8 shows. When the contrast is 0%, the fill will be a uniform shade of gray.



Figure 7-8. A property with a custom value graphic

## Custom Component Designers

Type converters and custom UI editors enable us to provide specialized editing facilities for custom property types. But what if we are writing controls and want to be able to customize the way they are presented on forms? Visual Studio .NET even lets us provide custom editing facilities for controls hosted in the forms designer, by writing a *custom designer*.

A custom designer is a class that derives from `ComponentDesigner` (which is defined in the `System.ComponentModel.Design` namespace). Designers for nonvisual components derive directly from this class, but control designers derive from one of the two `ControlDesigner` classes. (The `System.Windows.Forms.Design` and `System.Web.UI.Design` namespaces each have a `ControlDesigner` class. These are used for Windows Forms and Web Forms designers, respectively.)



This separation of runtime and design-time elements allows you to place all of the design-time code into a separate component. This will mean that, at runtime, your component will not be carrying any unnecessary design-time baggage, making it slightly more memory-efficient.

Whether for Web Forms Controls, Windows Forms Controls, or plain components, designer classes have certain commonalities. They are associated with their components by applying the `Designer` attribute (in the `System.ComponentModel` namespace) to the component class. And although most of the integration features are specific to either Windows Forms or Web Forms, all designer classes can add extra menu items to the Visual Studio .NET context menu.

## Adding Menu Verbs

To add extra items to the context menu for a component in the forms designer, we must override the associated designer class's `Verbs` property. This property is of type `DesignerVerbCollection`, which is defined in the `System.ComponentModel.Design` namespace.

Example 7-15 shows a control designer with an example `Verbs` property.

*Example 7-15. Adding custom menu verbs*

```
public class MyComponentDesigner : ComponentDesigner
{
    public override DesignerVerbCollection Verbs
    {
```

Example 7-15. Adding custom menu verbs (continued)

```
    get
    {
        DesignerVerb[] verbs = new DesignerVerb[]
        {
            new DesignerVerb("Add Widget",
                new EventHandler(OnAddWidget)),
            new DesignerVerb("Remove Widget",
                new EventHandler(OnRemoveWidget))
        };
        return new DesignerVerbCollection(verbs);
    }
}

private void OnAddWidget (object sender, EventArgs e)
{
    MyComponent ctl = (MyComponent) this.Component;
    . . .
}

private void OnRemoveWidget(object sender, EventArgs e)
{
    MyComponent ctl = (MyComponent) this.Component;
    . . .
}
}
```

The easiest way to build a `DesignerVerbsCollection` is to construct one from an array of `DesignerVerb` objects. Each `DesignerVerb` is relatively simple—it simply needs the text that will appear on the menu and a delegate referring to the event handler that should be called when the relevant menu item is clicked. So when you right-click on an item with this custom designer, Visual Studio .NET will show a context menu with extra Add Widget and Remove Widget menu items, as Figure 7-9 shows. It will call our `OnAddWidget` or `OnRemoveWidget` method, respectively, when these menu items are selected. (The component being edited can be retrieved from the `ComponentDesigner` base class property `Component`, as Example 7-15 shows.)

Any menu verbs added like this will also appear in the property grid. Visual Studio .NET adds an extra panel to the grid and shows verbs there using a hyperlink style (a blue, underlined word), as Figure 7-10 shows.

## Windows Forms Control Designers

Windows Forms custom control designers are essentially specialized component designers. They can provide extra menu items, just like a normal component designer. They can also modify how resizing and positioning are handled, paint adornments (such as extra handles) on your control, and manage mouse clicks in the Visual Studio .NET Windows Forms designer.

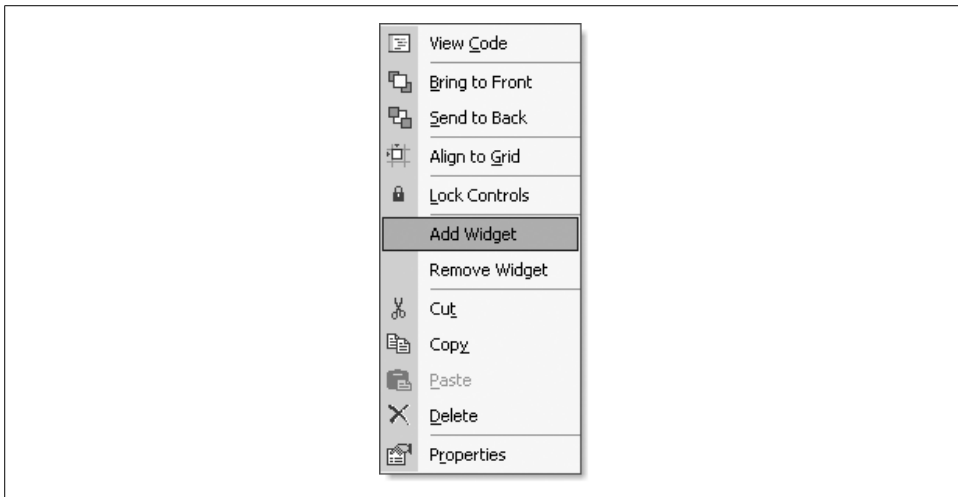


Figure 7-9. Visual Studio .NET context menu with custom items

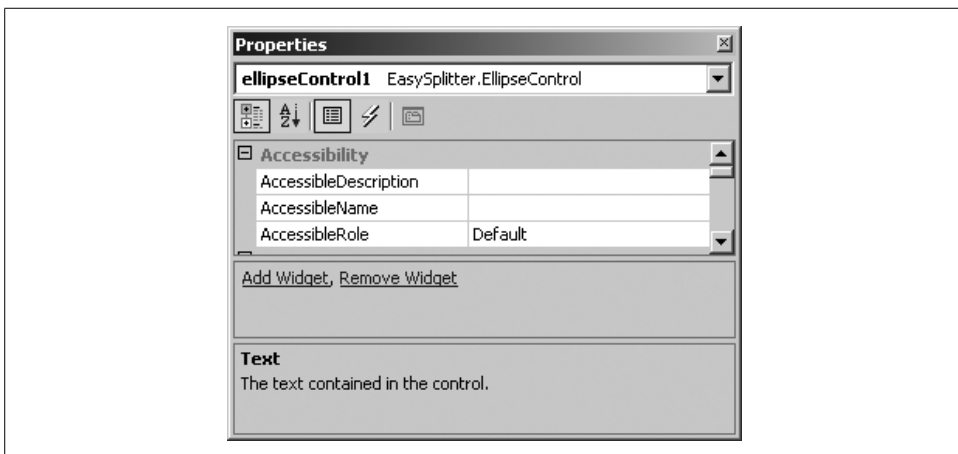


Figure 7-10. Custom verbs in the property grid

Example 7-16 shows a control with a custom designer, specified with the Designer attribute. The designer class itself must derive from the ControlDesigner class. (ControlDesigner itself derives from ComponentDesigner.) We choose which methods to override in the designer class based on which aspects of the control’s design-time functionality we would like to customize.

*Example 7-16. A Windows Forms control with a custom designer*

```
[Designer(typeof(MyControlDesigner))]
public class MyControl :
    System.Windows.Forms.Control
{
    . . .
}
```

## Resizing and moving

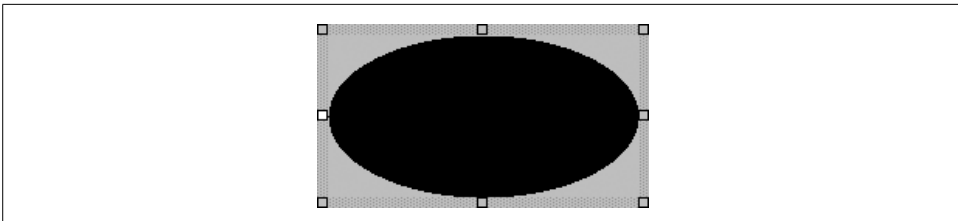
The forms designer will automatically provide all controls with an outline allowing them to be moved and resized. However, this is not always appropriate—some controls need to have a fixed size. (For example, the `TabPage` control's size and position are always determined by its parent `TabControl`.) Visual Studio .NET therefore lets us specify whether our control should be movable and which edges should be resizable. We simply override the `SelectionRules` property in our designer class, returning the required combination of bits from the `SelectionRules` enumeration (defined in the `System.Windows.Forms.Design` namespace).

Example 7-17 specifies `SelectionRules.Visible`, meaning that the resize/move outline should be displayed; it also indicates that the lefthand side of the outline should be resizable, with `SelectionRules.LeftSizeable`. (So, this particular control will not be vertically resizable. It cannot be moved either—you must specify `SelectionRules.Moveable` to enable that.) The default implementation of `SelectionRules` returns `SelectionRules.AllSizable | SelectionRules.Moveable | SelectionRules.Visible`.

*Example 7-17. Modifying support for moving and resizing*

```
public override SelectionRules SelectionRules
{
    get
    {
        return SelectionRules.Visible |
            SelectionRules.LeftSizeable;
    }
}
```

Figure 7-11 shows how the control with the designer class in Example 7-17 will look in the forms designer. Notice that all of the resize handles are gray, with the exception of the one halfway up the lefthand side, which is white. (Visual Studio .NET also uses the mouse cursor to indicate which edges can be resized. In this example, a resize cursor will appear only when the mouse is over the handle halfway up the lefthand side.) Resizing with all of the other handles has been disabled because we told Visual Studio .NET that the control cannot be moved, and only the lefthand side can be resized. VS.NET colors handles that cannot be moved gray.



*Figure 7-11. A control with one resizable edge*

## Adornments

Sometimes it is useful to add extra visual features to a control at design time, to allow developers to change properties visually. The outline and handles that Visual Studio .NET adds to controls to enable them to be moved and resized are an example of this. With a custom designer class, it is possible to add further such adornments of your own.

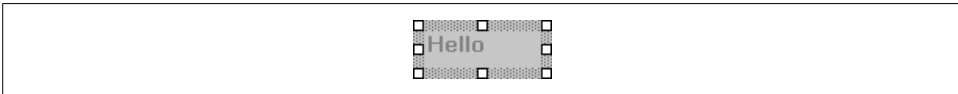
We could simply modify our control's `OnPaint` method to draw adornments at design time. (It is possible to detect that a control is hosted in a designer by examining the `Control` class's `DesignMode` property.) However, adornments are normally drawn only when the control is selected, and it is hard to detect this in `OnPaint`. Moreover, to do this would defeat the ability to separate runtime and design-time facets into separate components. Fortunately, Visual Studio .NET gives us an opportunity to paint adornments in our designer class. It will call the `OnPaintAdornments` method when the control is selected.

Example 7-18 illustrates the basic principle.

*Example 7-18. Drawing custom adornments*

```
protected override void OnPaintAdornments(PaintEventArgs pe)
{
    pe.Graphics.DrawString("Hello", Control.Font, Brushes.Red, 0, 0);
}
```

The results are shown in Figure 7-12. Normally, you would draw an adornment that reflected some aspect of the component's control, of course. But as this example shows, you draw adornments in just the same way that you draw in a normal `OnPaint` method—simply use the `Graphics` object supplied in the `PaintEventArgs` object.



*Figure 7-12. A custom adornment*



Many controls draw extra grab handles as adornments. For example, a control that shows rotated text might want to allow the angle to be controlled with a draggable handle. The `System.Windows.Forms.ControlPaint` class provides a method for doing this: `DrawGrabHandle`. This allows the size of the grab handles to be specified. To be consistent with Visual Studio .NET you should use `7x7`.

Visual Studio .NET provides extra visual feedback for its adornments—whenever the mouse moves over a grab handle or control outline, the mouse cursor changes. You can do the same thing for your adornments. If the mouse pointer is over your control, Visual Studio .NET will call your designer class's `OnSetCursor` method every

time it moves. You can write code in here to detect whether the cursor is over any of your grab handles (or other adornments) and set the cursor. Just set the `Cursor` class's `Current` property. Unfortunately, `OnSetCursor` is not passed the cursor's current position, so you must retrieve that from the `Cursor` class and map the coordinates to your control's coordinate space, using the technique shown in Example 7-19.

*Example 7-19. Modifying the cursor*

```
protected override void OnSetCursor()
{
    Point cp = Control.PointToClient(Cursor.Position);
    if (IsPointOverAnAdornment(cp))
    {
        Cursor.Current = Cursors.SizeWE;
    }
    else
        base.OnSetCursor();
}

private bool IsPointOverAnAdornment(Point p)
{
    . . . Do hit testing
}
```



You must call the base class's `OnSetCursor` method if you do not set the cursor yourself. Otherwise, the default cursor will not be restored when the mouse moves away from one of your adornments.

Most adornments are designed for clicking on and dragging. (Especially those drawn with `ControlPaint.DrawGrabHandle`.) You will, therefore, usually want to handle mouse input if you draw any adornments.

## Handling mouse input

Visual Studio .NET will notify a designer class of certain types of mouse activity. It presumes that controls will typically be interested in drag operations—the three methods it calls to indicate mouse activity are `OnMouseDownBegin`, `OnMouseDownMove`, and `OnMouseDownEnd`. Override these to be notified when the mouse button is first pressed, when the mouse moves while the button is pressed, and when the button is released, respectively.

All three methods are passed the current mouse position as a pair of integers. However, despite what the documentation claims, these are screen coordinates, so, as with `OnSetCursor`, you must use `Control.PointToClient` to map them back into your control's coordinate space.



You should always call the base class implementations of these methods unless you handle them completely yourself. You should always call the base `OnMouseDragEnd` method in any case. If you fail to call the base class's `OnMouseDragEnd`, the forms designer will be left in a state in which the mouse stops working correctly, as shown in Example 7-2.

## Example Windows Forms Control with Designer

This section presents a complete example of a custom Windows Forms control with an associated designer class to illustrate all of the points raised in the previous section. The control is a directional label control. It is similar to the built-in `Label` class, except it allows text to be displayed at any angle. Figure 7-13 shows an application using this control.



Figure 7-13. *DirectionalLabel* control

The source for the `DirectionalLabel` control is shown in Example 7-20. The structure of the class is fairly straightforward. It begins with a constructor. The `OnPaint` method follows—this contains the code that draws the rotated text. After the redraw code are two properties, `Origin` and `Direction`. These set the start position of the text and the direction in which it should be drawn. These properties have been annotated with the `Category` and `Description` attributes, to make sure that they are displayed correctly in the property grid. These properties also provide change notifications (through `OnOriginChanged` and `OnDirectionChanged` methods and associated events).

Because the `Origin` and `Direction` properties use the `Point` and `Size` types, respectively, it is not possible to use the `DefaultValue` attribute. (Attributes must be initialized with constant values. Here, the default values are `new Point(0,0)`, and `new Size(30,0)`. You cannot construct an attribute with these values.) These properties, therefore, have corresponding `ShouldSerialize` methods. This enables Visual Studio .NET to know whether the properties currently have their default values or not despite the absence of the `DefaultValue` attribute.

The control's appearance depends on several properties. As well as using the `Origin` and `Direction` properties, the redraw code in `OnPaint` uses the standard `Text`, `Font`, `ForeColor`, and `BackColor` properties. The control needs to be redrawn whenever any of these properties changes, so the control ends with a series of change handlers, all of which call `Invalidate` to redraw the control. Example 7-20 shows the source code for this control.

*Example 7-20. DirectionalLabel control class*

```
using System;
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Text;
using System.Windows.Forms;

[ToolboxBitmap(typeof(DirectionalLabel))]
[Designer(typeof(DirectionalLabelDesigner))]
public class DirectionalLabel : Control
{
    public DirectionalLabel()
    {
        // Enable double-buffering - reduces flicker when the
        // user adjusts the control in the designer.
        SetStyle(ControlStyles.AllPaintingInWmPaint |
            ControlStyles.DoubleBuffer | ControlStyles.UserPaint, true);
    }

    protected override void OnPaint(PaintEventArgs pe)
    {
        Graphics g = pe.Graphics;
        float angle = (float) (Math.Atan2(Direction.Height, Direction.Width) /
            Math.PI * 180.0);

        g.TranslateTransform(Origin.X, Origin.Y);
        g.RotateTransform(angle);
        g.TextRenderingHint = TextRenderingHint.AntiAlias;
        using (Brush b = new SolidBrush(ForeColor))
        {
            g.DrawString(Text, Font, b, 0, 0);
        }

        base.OnPaint(pe);
    }

    [Category("Appearance")]
    [Description("The starting point (top left) of the label's text")]
    public Point Origin
    {
        get
        {
            return originVal;
        }
        set
        {
            if (value != originVal)
            {
                originVal = value;
                OnOriginChanged(EventArgs.Empty);
            }
        }
    }
}
```

Example 7-20. *DirectionalLabel* control class (continued)

```
    }  
  }  
}  
private Point originVal = new Point(0, 0);  
  
public event EventHandler OriginChanged;  
protected virtual void OnOriginChanged(EventArgs e)  
{  
    if (OriginChanged != null)  
        OriginChanged(this, e);  
    Invalidate();  
}  
  
public bool ShouldSerializeOrigin()  
{  
    return Origin != new Point(0, 0);  
}  
  
[Category("Appearance")]  
[Description("The direction in which the text will be drawn")]  
public Size Direction  
{  
    get  
    {  
        return directionVal;  
    }  
    set  
    {  
        if (value != directionVal)  
        {  
            directionVal = value;  
            OnDirectionChanged(EventArgs.Empty);  
        }  
    }  
}  
private Size directionVal = new Size(30, 0);  
  
public event EventHandler DirectionChanged;  
protected virtual void OnDirectionChanged(EventArgs e)  
{  
    if (DirectionChanged != null)  
        DirectionChanged(this, e);  
    Invalidate();  
}  
  
public bool ShouldSerializeDirection()  
{  
    return Direction != new Size(30, 0);  
}
```

*Example 7-20. DirectionalLabel control class (continued)*

```
protected override void OnForeColorChanged(System.EventArgs e)
{
    Invalidate();
    base.OnForeColorChanged(e);
}

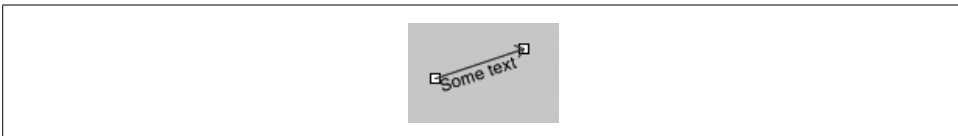
protected override void OnBackColorChanged(System.EventArgs e)
{
    Invalidate();
    base.OnBackColorChanged(e);
}

protected override void OnFontChanged(System.EventArgs e)
{
    Invalidate();
    base.OnFontChanged(e);
}

protected override void OnTextChanged(System.EventArgs e)
{
    Invalidate();
    base.OnTextChanged(e);
}
}
```

The control has had the `ToolboxBitmap` attribute applied. This means that the custom embedded bitmap will be used when the control is displayed in a Visual Studio .NET toolbox. (You can add a control to a toolbox either by dragging the DLL from a Windows Explorer window onto the toolbox or by using the toolbox's customization menu option.)

The control also has the `Designer` attribute, indicating that it has an associated designer class. The designer allows the position and direction of the text to be adjusted in the Visual Studio .NET Forms Editor using a pair of grab handles, as shown in Figure 7-14. These grab handles have an arrow drawn between them to make it clear in which direction the text will be displayed. Either grab handle can be moved with the mouse at design time. The designer class that supplies this editing facility, `DirectionalLabelDesigner`, is shown in Example 7-21.



*Figure 7-14. The DirectionalLabel at design time*

*Example 7-21. The direction label control's designer class*

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
using System.Windows.Forms.Design;
using System.ComponentModel;
using System.ComponentModel.Design;

public class DirectionalLabelDesigner : ControlDesigner
{
    public override void Initialize(IComponent component)
    {
        base.Initialize(component);

        selectionService = GetService(typeof(ISelectionService))
            as ISelectionService;
        if (selectionService != null)
        {
            selectionService.SelectionChanged +=
                new EventHandler(OnSelectionChanged);
        }
    }
    private ISelectionService selectionService;

    private void OnSelectionChanged(object sender, EventArgs e)
    {
        Control.Invalidate();
    }

    protected override void OnPaintAdornments(PaintEventArgs pe)
    {
        DirectionalLabel label = (DirectionalLabel) Control;
        if (selectionService != null)
        {
            if (selectionService.GetComponentSelected(label))
            {
                // Paint grab handles.

                Graphics g = pe.Graphics;
                Rectangle handle = GetHandle(label.Origin);
                ControlPaint.DrawGrabHandle(g, handle, true, true);
                handle = GetHandle(label.Origin + label.Direction);
                ControlPaint.DrawGrabHandle(g, handle, true, true);

                // Paint a line with an arrow—this makes it
                // more clear which grab handle is which.
                //
                // The built-in line caps are a bit small, so we'll
                // draw our own arrow on the end. The easiest way
                // to do this is to translate and rotate the transform.
            }
        }
    }
}
```

Example 7-21. The direction label control's designer class (continued)

```
        float angle = (float) (Math.Atan2(label.Direction.Height,
            label.Direction.Width) / Math.PI * 180.0);
        g.TranslateTransform(label.Origin.X, label.Origin.Y);
        g.RotateTransform(angle);

        float distance = (float) Math.Sqrt(
            label.Direction.Width * label.Direction.Width +
            label.Direction.Height * label.Direction.Height);

        g.SmoothingMode = SmoothingMode.AntiAlias;
        using (Pen p = new Pen(Color.Blue))
        {
            g.DrawLine(p, 0, 0, distance, 0);
            g.DrawLine(p, distance, 0, distance - 5, -4);
            g.DrawLine(p, distance, 0, distance - 5, 4);
        }
    }
}

// Get a standard-sized grab handle rectangle centered on
// the specified point.
private Rectangle GetHandle(Point pt)
{
    Rectangle handle = new Rectangle(pt, new Size(7, 7));
    handle.Offset(-3, -3);
    return handle;
}

protected override void OnSetCursor()
{
    // Get mouse cursor position relative to
    // the control's coordinate space.

    DirectionalLabel label = (DirectionalLabel) Control;
    Point p = label.PointToClient(Cursor.Position);

    // Display a resize cursor if the mouse is
    // over a grab handle; otherwise show a
    // normal arrow.

    if (GetHandle(label.Origin).Contains(p) ||
        GetHandle(label.Origin + label.Direction).Contains(p))
    {
        Cursor.Current = Cursors.SizeAll;
    }
    else
    {
        Cursor.Current = Cursors.Default;
    }
}
```

Example 7-21. The direction label control's designer class (continued)

```
// Drag handling state and methods.

private bool dragging = false;
private bool dragDirection;
private Point dragOffset;

protected override void OnMouseDownBegin(int x, int y)
{
    DirectionalLabel label = (DirectionalLabel) Control;
    Point p = label.PointToClient(new Point(x, y));

    bool overOrigin = GetHandle(label.Origin).Contains(p);
    bool overDirection = GetHandle(label.Origin + label.Direction).Contains(p);
    if (overOrigin || overDirection)
    {
        dragging = true;
        dragDirection = overDirection;
        Point current = dragDirection ?
            (label.Origin + label.Direction) :
            label.Origin;
        dragOffset = current - new Size(p);
    }
    else
    {
        dragging = false;
        base.OnMouseDownBegin(x, y);
    }
}

protected override void OnMouseDownMove(int x, int y)
{
    if (dragging)
    {
        DirectionalLabel label = (DirectionalLabel) Control;
        Point p = label.PointToClient(new Point(x, y));

        Point current = p + new Size(dragOffset);
        if (dragDirection)
        {
            label.Direction = new Size(current) - new Size(label.Origin);
        }
        else
        {
            label.Origin = current;
        }
    }
    else
    {
        base.OnMouseDownMove(x, y);
    }
}
```

Example 7-21. The direction label control's designer class (continued)

```
protected override void OnMouseDragEnd(bool cancel)
{
    if (dragging)
    {
        // Update property via PropertyDescriptor to
        // make sure that VS.NET notices.

        DirectionalLabel label = (DirectionalLabel) Control;
        if (dragDirection)
        {
            Size d = label.Direction;
            PropertyDescriptor pd =
                TypeDescriptor.GetProperties(label)["Direction"];
            pd.SetValue(label, d);
        }
        else
        {
            Point o = label.Origin;
            PropertyDescriptor pd =
                TypeDescriptor.GetProperties(label)["Origin"];
            pd.SetValue(label, o);
        }
        dragging = false;
    }

    // Always call base class.
    base.OnMouseDragEnd(cancel);
}
}
```

The grab handle and line adornments are drawn only when the control is selected, so the class starts with code that causes the control to be redrawn each time a selection change event occurs. This is followed by the `OnPaintAdornments` method, which renders the grab handles and the line.

The remaining code handles mouse input. `OnSetCursor` is used to display the resize cursor whenever the mouse is over one of the grab handles. The remaining three methods update the appropriate properties when a drag operation occurs. The only surprising code here is the use of the `PropertyDescriptor` class in `OnMouseDragEnd`. Without this code in place, Visual Studio .NET does not notice when a drag operation causes a control's property to change. However, if we update the property through a `PropertyDescriptor`, it will detect the change and save the modified property in the form's `InitializeComponent` method.

## Web Forms Control Designers

A Web Forms Control custom designer is a class derived from the `ControlDesigner` class defined in the `System.Web.UI.Design` namespace. This class derives from

ComponentDesigner and inherits the standard designer features such as the ability to add extra context menu items. As Example 7-22 shows, a web control designer is associated with a control in exactly the same way as any other component designer. It can also control its resizability in the designer, and it can influence the way it appears in Visual Studio .NET's design-time HTML view.

*Example 7-22. A Web Forms control with a custom designer*

```
[Designer(typeof(MyControlDesigner))]  
public class MyControl :  
    System.Web.UI.Control  
{  
    . . .  
}
```

## Resizing

A Web Forms control has less power than a Windows Forms control over the way in which it can be resized. With Web Forms, it is a yes/no choice—we can override the AllowResize property and return a Boolean indicating whether we want the control to be resizable in the designer.

## Design-time rendering

When your control is hosted in the designer, Visual Studio .NET will create an instance of it and ask it to render itself in the normal way. This means that the control will look the same at design time as it does at runtime. Most of the time, this will be the behavior that you require. However, sometimes you will want to provide a different appearance at design time. For example, your control may not be visible at runtime, in which case it is useful to be able to make something appear in the designer so that developers can see and select your control.

To modify the control's design-time appearance, override the GetDesignTimeHtml method in the designer class. This method returns a string, which should be HTML. Although you can return whatever you like here, the ControlDesigner class provides a protected method called CreatePlaceholderDesignTimeHtml that will generate a placeholder for you. Example 7-23 shows how to use this. It just generates a gray box containing the specified text.

*Example 7-23. Providing design-time HTML*

```
public class WebControlDesigner : System.Web.UI.Design.ControlDesigner  
{  
    public override string GetDesignTimeHtml()  
    {  
        return CreatePlaceholderDesignTimeHtml("My control");  
    }  
}
```

There is another popular use of the `GetDesignTimeHtml` method. Data-bound controls might be invisible unless they have some information to display. It is common practice for such controls to preload some fake data at design time so as to be visible. You can do this in the `GetDesignTimeHtml` method and then call the base class's implementation to get your control to render itself as usual.

## Conclusion

Visual Studio .NET automates a great many of the design-time features of components—all public properties are intrinsically editable, well-known property types get special-purpose editing user interfaces, components derived from certain well-known base classes (e.g., `Controls`) get further special design-time support. However, VS.NET also allows component authors to enhance design-time behavior. This can range from simply adding attributes to categorize properties and events to providing full custom editing for controls or their properties.