9
Design-Time Integration

A COMPONENT IS A NONVISUAL CLASS designed specifically to integrate with a design-time environment such as Visual Studio .NET. WinForms provides several standard components, and .NET lets you build your own, gaining a great deal of design-time integration with very little work.

On the other hand, with a bit more effort, you can integrate nonvisual components and controls very tightly into the design-time environment, providing a rich development experience for the programmer using your custom components and controls.

Components

Recall from Chapter 8: Controls that controls gain integration into VS.NET merely by deriving from the Control base class in the System.Windows.Forms namespace. That's not the whole story. What makes a control special is that it's one kind of *component*: a .NET class that integrates with a design-time environment such as VS.NET. A component can show up on the Toolbox along with controls and can be dropped onto any design surface. Dropping a component onto a design surface makes it available to set the property or handle the events in the Designer, just as a control is. Figure 9.1 shows the difference between a hosted control and a hosted component.

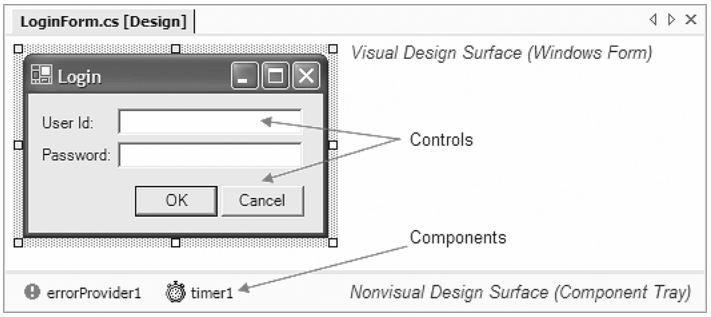

FIGURE 9.1: Locations of Components and Controls Hosted on a Form

Standard Components

It's so useful to be able to create instances of nonvisual components and use the Designer to code against them that WinForms comes with several components out of the box:

- **Standard dialogs.** The ColorDialog, FolderBrowserDialog, FontDialog, OpenFileDialog, PageSetupDialog, PrintDialog, PrintPreviewDialog, and SaveFileDialog classes make up the bulk of the standard components that WinForms provides. The printing-related components are covered in detail in Chapter 7: Printing.
- **Menus.** The MainMenu and ContextMenu components provide a form's menu bar and a control's context menu. They're both covered in detail in Chapter 2: Forms.
- **User information.** The ErrorProvider, HelpProvider, and ToolTip components provide the user with varying degrees of help in using a form and are covered in Chapter 2: Forms.
- **Notify icon.** The NotifyIcon component puts an icon on the shell's TaskBar, giving the user a way to interact with an application without the screen real estate requirements of a window. For an example, see Appendix D: Standard WinForms Components and Controls.
- **Image List.** The ImageList component keeps track of a developer-provided list of images for use with controls that need images when drawing. Chapter 8: Controls shows how to use them.

- **Timer.** The Timer component fires an event at a set interval measured in milliseconds.

Using Standard Components

What makes components useful is that they can be manipulated in the design-time environment. For example, imagine that you'd like a user to be able to set an alarm in an application and to notify the user when the alarm goes off. You can implement that using a Timer component. Dropping a Timer component onto a Form allows you to set the Enabled and Interval properties as well as handle the Tick event in the Designer, which generates code such as the following into InitializeComponent:

```
void InitializeComponent() {
  this.components = new Container();
  this.timer1 = new Timer(this.components);
  ...
  // timer1
  this.timer1.Enabled = true;
  this.timer1.Tick += new EventHandler(this.timer1_Tick);
  ...
}
```

As you have probably come to expect by now, the Designer-generated code looks very much like what you'd write yourself. What's interesting about this sample InitializeComponent implementation is that when a new component is created, it's put on a list with the other components on the form. This is similar to the Controls collection that is used by a form to keep track of the controls on the form.

After the Designer has generated most of the Timer-related code for us, we can implement the rest of the alarm functionality for our form:

```
DateTime alarm = DateTime.MaxValue; // No alarm

void setAlarmButton_Click(object sender, EventArgs e) {
  alarm = dateTimePicker1.Value;
}

// Handle the Timer's Tick event
void timer1_Tick(object sender, System.EventArgs e) {
  statusBar1.Text = DateTime.Now.TimeOfDay.ToString();
```

continues

```
  // Check to see whether we're within 1 second of the alarm
  double seconds = (DateTime.Now - alarm).TotalSeconds;
  if( (seconds >= 0) && (seconds <= 1) ) {
    alarm = DateTime.MaxValue; // Show alarm only once
    MessageBox.Show("Wake Up!");
  }
}
```

In this sample, when the timer goes off every 100 milliseconds (the default value), we check to see whether we're within 1 second of the alarm. If we are, we shut off the alarm and notify the user, as shown in Figure 9.2.

If this kind of single-fire alarm is useful in more than one spot in your application, you might choose to encapsulate this functionality in a custom component for reuse.

Custom Components

A component is any class that implements the IComponent interface from the System.ComponentModel namespace:

```
interface IComponent : IDisposable {
  ISite Site { get; set; }
  event EventHandler Disposed;
}

interface IDisposable {
  void Dispose();
}
```

A class that implements the IComponent interface can be added to the Toolbox[1] in VS.NET and dropped onto a design surface. When you drop a

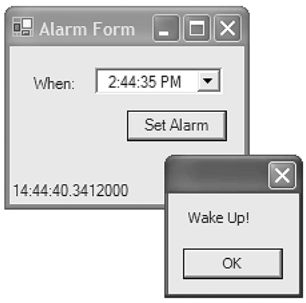

FIGURE 9.2: The Timer Component Firing Every 100 Milliseconds

component onto a form, it shows itself in a tray below the form. Unlike controls, components don't draw themselves in a region on their container. In fact, you could think of components as nonvisual controls, because, just like controls, components can be managed in the design-time environment. However, it's more accurate to think of controls as visual components because controls implement IComponent, which is where they get their design-time integration.

A Sample Component

As an example, to package the alarm functionality we built earlier around the Timer component, let's build an AlarmComponent class. To create a new component class, right-click on the project and choose Add | Add Component, enter the name of your component class, and press OK. You'll be greeted with a blank design surface, as shown in Figure 9.3.

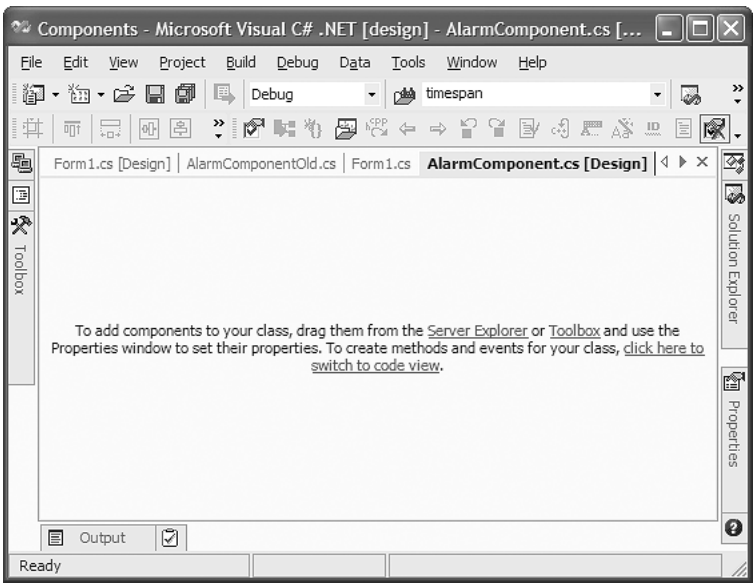

FIGURE 9.3: A New Component Design Surface

1. The same procedure for adding a custom control to the Toolbox in Chapter 8: Controls can be used to add a custom component to the Toolbox.

The design surface for a component is meant to host other components for use in implementing your new component. For example, we can drop our Timer component from the Toolbox onto the alarm component design surface. In this way, we can create and configure a timer component just as if we were hosting the timer on a form. Figure 9.4 shows the alarm component with a timer component configured for our needs.

Switching to the code view[2] for the component displays the following skeleton generated by the component project item template and filled in by the Designer for the timer:

```
using System;
using System.ComponentModel;
using System.Collections;
using System.Diagnostics;

namespace Components {
  /// <summary>
  /// Summary description for AlarmComponent.
  /// </summary>
```

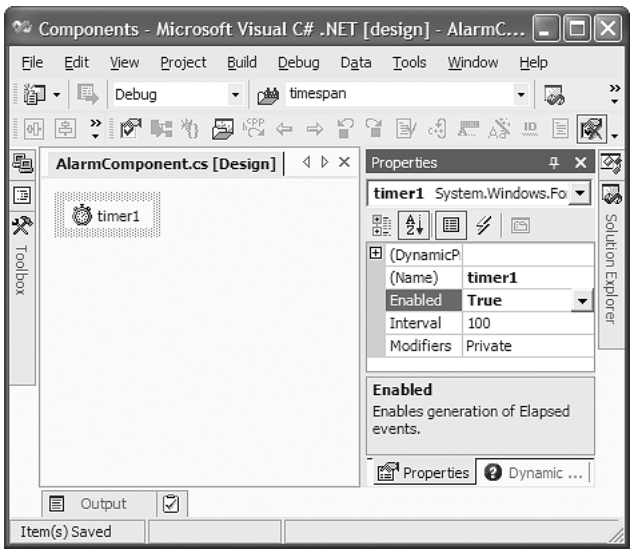

FIGURE 9.4: A Component Design Surface Hosting a Timer Component

2. You can switch to code view from designer view by choosing View | Code or pressing F7. Similarly, you can switch back by choosing View | Designer or by pressing Shift+F7.

```
public class AlarmComponent : System.ComponentModel.Component {
  private Timer timer1;
  private System.ComponentModel.IContainer components;

  public AlarmComponent(System.ComponentModel.IContainer container) {
    /// <summary>
    /// Required for Windows.Forms Class Composition Designer support
    /// </summary>
    container.Add(this);
    InitializeComponent();

    //
    // TODO: Add any constructor code after InitializeComponent call
    //
  }

  public AlarmComponent() {
    /// <summary>
    /// Required for Windows.Forms Class Composition Designer support
    /// </summary>
    InitializeComponent();

    //
    // TODO: Add any constructor code after InitializeComponent call
    //
  }

  #region Component Designer generated code
    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent() {
      this.components = new System.ComponentModel.Container();
      this.timer1 = new System.Windows.Forms.Timer(this.components);
      //
      // timer1
      //
      this.timer1.Enabled = true;
    }
  #endregion
  }
}
```

Notice that the custom component derives from the Component base class from the System.ComponentModel namespace. This class provides an implementation of IComponent for us.

After the timer is in place in the alarm component, it's a simple matter to move the alarm functionality from the form to the component by handling the timer's Tick event:

```
public class AlarmComponent : Component {
  ...
  DateTime alarm = DateTime.MaxValue; // No alarm
  public DateTime Alarm {
    get { return this.alarm; }
    set { this.alarm = value; }
  }

  // Handle the Timer's Tick event
  public event EventHandler AlarmSounded;

  void timer1_Tick(object sender, System.EventArgs e) {
    // Check to see whether we're within 1 second of the alarm
    double seconds = (DateTime.Now - this.alarm).TotalSeconds;
    if( (seconds >= 0) && (seconds <= 1) ) {
      this.alarm = DateTime.MaxValue; // Show alarm only once
      if( this.AlarmSounded != null ) {
        AlarmSounded(this, EventArgs.Empty);
      }
    }
  }
}
```

This implementation is just like what the form was doing before, except that the alarm date and time are set via the public Alarm property; when the alarm sounds, an event is fired. Now we can simplify the form code to contain merely an instance of the AlarmComponent, setting the Alarm property and handling the AlarmSounded event:

```
public class AlarmForm : Form {
  AlarmComponent alarmComponent1;
  ...
  void InitializeComponent() {
    ...
    this.alarmComponent1 = new AlarmComponent(this.components);
    ...
    this.alarmComponent1.AlarmSounded +=
      new EventHandler(this.alarmComponent1_AlarmSounded);
    ...
  }
}
```

```
void setAlarmButton_Click(object sender, EventArgs e) {
  alarmComponent1.Alarm = dateTimePicker1.Value;
}

void alarmComponent1_AlarmSounded(object sender, EventArgs e) {
  MessageBox.Show("Wake Up!");
}
```

In this code, the form uses an instance of AlarmComponent, setting the Alarm property based on user input and handling the AlarmSounded event when it's fired. The code does all this without any knowledge of the actual implementation, which is encapsulated inside the AlarmComponent itself.

Component Resource Management

Although components and controls are similar as far as their design-time interaction is concerned, they are not identical. The most obvious difference lies in the way they are drawn on the design surface. A less obvious difference is that the Designer does not generate the same hosting code for components that it does for controls. Specifically, a component gets extra code so that it can add itself to the container's list of components. When the container shuts down, it uses this list of components to notify all the components that they can release any resources that they're holding.

Controls don't need this extra code because they already get the Closed event, which is an equivalent notification for most purposes. To let the Designer know that it would like to be notified when its container goes away, a component can implement a public constructor that takes a single argument of type IContainer:

```
public AlarmComponent(IContainer container) {
  // Add object to container's list so that
  // we get notified when the container goes away
  container.Add(this);
  InitializeComponent();
}
```

Notice that the constructor uses the passed container interface to add itself as a container component. In the presence of this constructor, the Designer generates code that uses this constructor, passing it a container

for the component to add itself to. Recall that the code to create the AlarmComponent uses this special constructor:

```
public class AlarmForm : Form {
  IContainer components;
  AlarmComponent alarmComponent1;
  ...
  void InitializeComponent() {
    this.components = new Container();
    ...
    this.alarmComponent1 = new AlarmComponent(this.components);
    ...
  }
}
```

By default, most of the VS.NET-generated classes that contain components will notify each component in the container as part of the Dispose method implementation:

```
public class AlarmForm : Form {
  ...
  IContainer components;
  ...
  // Overridden from the base class Component.Dispose method
  protected override void Dispose( bool disposing ) {
    if( disposing ) {
      if (components != null) {
        // Call each component's Dispose method
        components.Dispose();
      }
    }
    base.Dispose( disposing );
  }
}
```

As you may recall from Chapter 4: Drawing Basics, the client is responsible for calling the Dispose method from the IDisposable interface. The IContainer interface derives from IDisposable, and the Container implementation of Dispose walks the list of components, calling IDisposable. Dispose on each one. A component that has added itself to the container can override the Component base class's Dispose method to catch the notification that is being disposed of:

```
public class AlarmComponent : Component {
  Timer timer1;
  IContainer components;
  ...
  void InitializeComponent() {
    this.components = new Container();
    this.timer1 = new Timer(this.components);
    ...
  }

  protected override void Dispose(bool disposing) {
    if( disposing ) {
      // Release managed resources
      ...

      // Let contained components know to release their resources
      if( components != null ) {
        components.Dispose();
      }
    }

    // Release unmanaged resources
    ...
  }
}
```

Notice that, unlike the method that the client container is calling, the alarm component's Dispose method takes an argument. The Component base class routes the implementation of IDisposable.Dispose() to call its own Dispose(bool) method, with the Boolean argument disposing set to true. This is done to provide optimized, centralized resource management.

A disposing argument of true means that Dispose was called by a client that remembered to properly dispose of the component. In the case of our alarm component, the only resources we have to reclaim are those of the timer component we're using to provide our implementation, so we ask our own container to dispose of the components it's holding on our behalf. Because the Designer-generated code added the timer to our container, that's all we need to do.

A disposing argument of false means that the client forgot to properly dispose of the object and that the .NET Garbage Collector (GC) is calling our object's finalizer. A *finalizer* is a method that the GC calls when it's

about to reclaim the memory associated with the object. Because the GC calls the finalizer at some indeterminate time—potentially long after the component is no longer needed (perhaps hours or days later)—the finalizer is a bad place to reclaim resources, but it's better than not reclaiming them at all.

The Component base class's finalizer implementation calls the Dispose method, passing a disposing argument of false, which indicates that the component shouldn't touch any of the managed objects it may contain. The other managed objects should remain untouched because the GC may have already disposed of them, and their state is undefined.

Any component that contains other objects that implement IDisposable, or handles to unmanaged resources, should implement the Dispose(bool) method to properly release those objects' resources when the component itself is being released by its container.

Design-Time Integration Basics

Because a component is a class that's made to be integrated into a design-time host, it has a life separate from the run-time mode that we normally think of for objects. It's not enough for a component to do a good job when interacting with a user at run time as per developer instructions; a component also needs to do a good job when interacting with the developer at design time.

Hosts, Containers, and Sites

In Visual Studio .NET, the Windows Forms Designer is responsible for providing design-time services during Windows Forms development. At a high level, these services include a form's UI and code views. The responsibility of managing integration between design-time objects and the designer is handled by the designer's internal implementation of IDesignerHost (from the System.ComponentModel.Design namespace). The designer host stores IComponent references to all design-time objects on the current form and also stores the form itself (which is also a component). This collection of components is available from the IDesignerHost interface through the Container property of type IContainer (from the System.ComponentModel namespace):

```
interface IContainer : IDisposable {
  ComponentCollection Components { get; }
  void Add(IComponent component);
  void Add(IComponent component, string name);
  void Remove(IComponent component);
}
```

This implementation of IContainer allows the designer host to establish a relationship that helps it manage each of the components placed on the form. Contained components can access the designer host and each other through their container at design time. Figure 9.5 illustrates this two-way relationship.

In Figure 9.5 you can see that the fundamental relationship between the designer host and its components is established with an implementation of the ISite interface (from the System.ComponentModel namespace):

```
interface ISite : IServiceProvider {
  IComponent Component { get; }
  IContainer Container { get; }
  bool DesignMode { get; }
  string Name { get; set; }
}
```

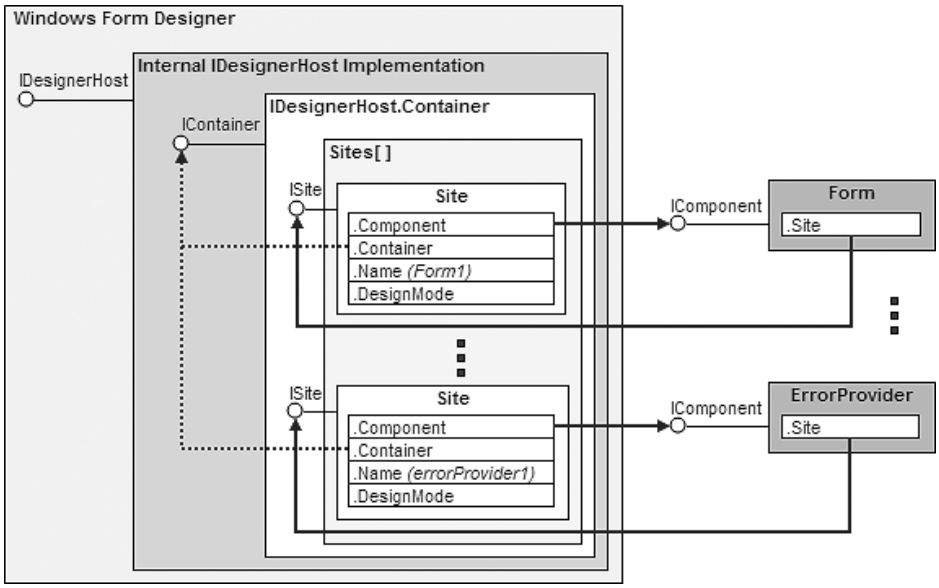

FIGURE 9.5: Design-Time Architecture

Internally, a container stores an array of sites. When each component is added to the container, the designer host creates a new site, connecting the component to its design-time container and vice versa by passing the ISite interface in the IComponent.Site property implementation:

```
interface IComponent : IDisposable {
  ISite Site { get; set; }
  event EventHandler Disposed;
}
```

The Component base class implements IComponent and caches the site's interface in a property. It also provides a helper property to go directly to the component's container without having to go first through the site:

```
class Component : MarshalByRefObject, IComponent, IDisposable {
  public IContainer Container { get; }
  public virtual ISite Site { get; set; }
  protected bool DesignMode { get; }
  protected EventHandlerList Events { get; }
}
```

The Component base class gives a component direct access to both the container and the site. A component can also access the Visual Studio .NET designer host itself by requesting the IDesignerHost interface from the container:

```
IDesignerHost designerHost = this.Container as IDesignerHost;
```

In Visual Studio .NET, the designer has its own implementation of the IDesignerHost interface, but, to fit into other designer hosts, it's best for a component to rely only on the interface and not on any specific implementation.

Debugging Design-Time Functionality

To demonstrate the .NET Framework's various design-time features and services, I've built a sample.[3] Because components and controls share the

3. While I use the term "I" to maintain consistency with the rest of the prose, it was actually Michael Weinhardt who built this sample as well as doing the initial research and even the initial draft of much of the material in this chapter. Thanks, Michael!

same design-time features and because I like things that look snazzy, I built a digital/analog clock control with the following public members:

```
public class ClockControl : Control {
  public ClockControl();
  public DateTime Alarm { get; set; }
  public bool IsItTimeForABreak { get; set; }
  public event AlarmHandler AlarmSounded;
  ...
}
```

Figure 9.6 shows the control in action.

When you build design-time features into your components,[4] you'll need to test them and, more than likely, debug them. To test run-time functionality, you simply set a breakpoint in your component's code and run a test application, relying on Visual Studio .NET to break at the right moment.

What makes testing design-time debugging different is that you need a design-time host to debug against; an ordinary application won't do. Because the hands-down hosting favorite is Visual Studio .NET itself, this means that you'll use one instance of Visual Studio .NET to debug another instance of Visual Studio .NET with a running instance of the component loaded. This may sound confusing, but it's remarkably easy to set up:

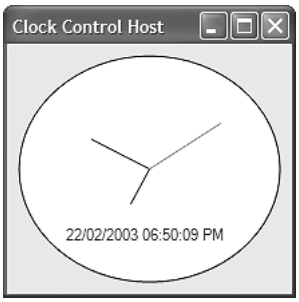

FIGURE 9.6: Snazzy Clock Control

4. Remember that nonvisual components as well as controls are components for the purposes of design-time integration.

1. Open the component solution to debug in one instance of Visual Studio .NET.
2. Set a second instance of Visual Studio .NET as your debug application by going to Project | Properties | Configuration Properties | Debugging and setting the following properties:
   a. Set Debug Mode to Program.
   b. Set Start Application to <your devenv.exe path>\devenv.exe.
   c. Set Command Line Arguments to <your test solution path>\yourTestSolution.sln.
3. Choose Set As StartUp Project on your component project.
4. Set a breakpoint in the component.
5. Use Debug | Start (F5) to begin debugging.

At this point, a second instance of Visual Studio.NET starts up with another solution, allowing you to break and debug at will, as illustrated in Figure 9.7.

The key to making this setup work is to have one solution loaded in one instance of VS.NET that starts another instance of VS.NET with a completely different solution to test your component in design mode.

The DesignMode Property

To change the behavior of your component at design time, often you need to know that you're running in a Designer. For example, the clock control uses a timer component to track the time via its Tick event handler:

```
public class ClockControl : Control {
  ...
  Timer timer = new Timer();
  ...
  public ClockControl() {
    ...
    // Initialize timer
    timer.Interval = 1000;
    timer.Tick += new System.EventHandler(this.timer_Tick);
    timer.Enabled = true;
  }
  ...
```

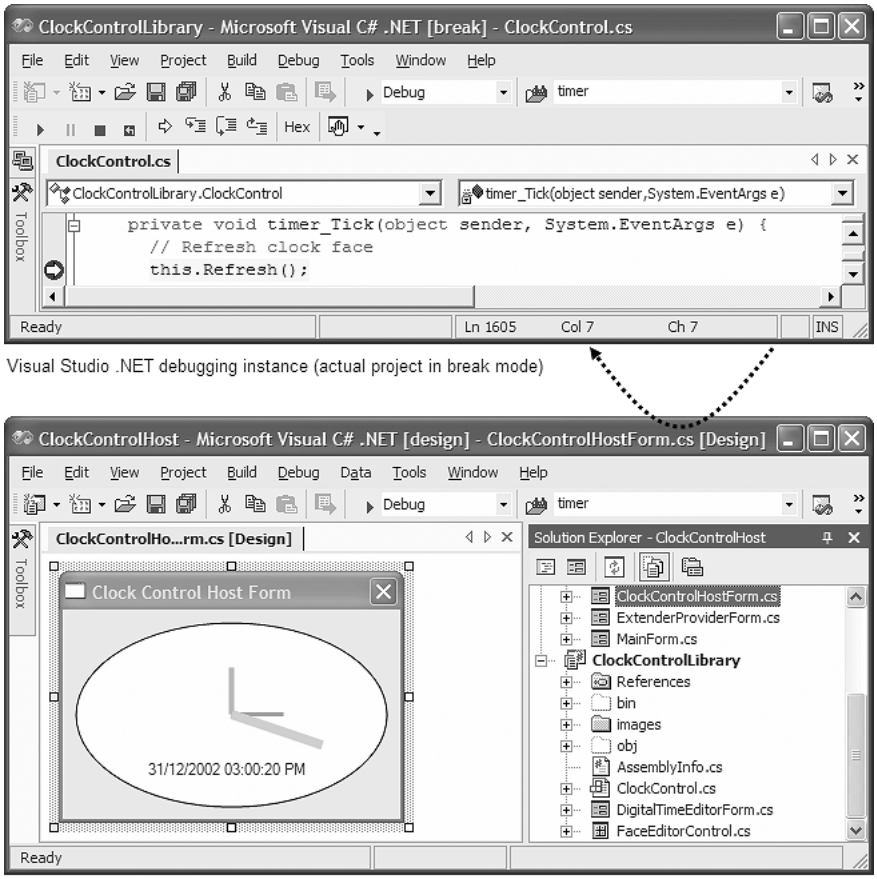

FIGURE 9.7: Design-Time Control Debugging

```
  void timer_Tick(object sender, EventArgs e) {
    // Refresh clock face
    this.Invalidate();
    ...
  }
}
```

Inspection reveals that the control is overly zealous in keeping time both at design time and at run time. Such code should really be executed at run time only. In this situation, a component or control can check the

DesignMode property, which is true only when it is executing at design time. The timer_Tick event handler can use DesignMode to ensure that it is executed only at run time, returning immediately from the event handler otherwise:

```
void timer_Tick(object sender, EventArgs e) {
  // Don't execute event if running in design time
  if( this.DesignMode ) return;
  this.Invalidate();
  ...
}
```

Note that the DesignMode property should not be checked from within the constructor or from any code that the constructor calls. A constructor is called before a control is sited, and it's the site that determines whether or not a control is in design mode. DesignMode will also be false in the constructor.

Attributes

Design-time functionality is available to controls in one of two ways: programmatically and declaratively. Checking the DesignMode property is an example of the programmatic approach. One side effect of using a programmatic approach is that your implementation takes on some of the design-time responsibility, resulting in a blend of design-time and run-time code within the component implementation.

The declarative approach, on the other hand, relies on attributes to request design-time functionality implemented somewhere else, such as the designer host. For example, consider the default Toolbox icon for a component, as shown in Figure 9.8.

If the image is important to your control, you'll want to change the icon to something more appropriate. The first step is to add a 16×16, 16-color icon or bitmap to your project and set its Build Action to Embedded

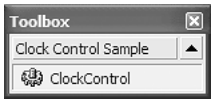

FIGURE 9.8: Default Toolbox Icon

Resource (embedded resources are discussed in Chapter 10: Resources). Then add the ToolboxBitmapAttribute to associate the icon with your component:

```
[ToolboxBitmapAttribute(
  typeof(ClockControlLibrary.ClockControl),
  "images.ClockControl.ico")]
public class ClockControl : Control {...}
```

The parameters to this attribute specify the use of an icon resource located in the "images" project subfolder.

You'll find that the Toolbox image doesn't change if you add or change ToolboxBitmapAttribute after the control has been added to the Toolbox. However, if your implementation is a component, its icon is updated in the component tray. One can only assume that the Toolbox is not under the direct management of the Windows Form Designer, whereas the component tray is. To refresh the Toolbox, remove your component and then add it again to the Toolbox. The result will be something like Figure 9.9.

You can achieve the same result without using ToolboxBitmapAttribute: Simply place a 16×16, 16-color bitmap in the same project folder as the component, and give it the same name as the component class. This is a special shortcut for the ToolboxBitmapAttribute only; don't expect to find similar shortcuts for other design-time attributes.

Property Browser Integration

No matter what the icon is, after a component is dragged from the Toolbox onto a form, it can be configured through the designer-managed Property Browser. The Designer uses reflection to discover which properties the design-time control instance exposes. For each property, the Designer calls the associated get accessor for its current value and renders both the

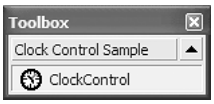

FIGURE 9.9: New and Improved Toolbox Icon

property name and the value onto the Property Browser. Figure 9.10 shows how the Property Browser looks for the basic clock control.

The System.ComponentModel namespace provides a comprehensive set of attributes, shown in Table 9.1, to help you modify your component's behavior and appearance in the Property Browser.

By default, public read and read/write properties—such as the Alarm property highlighted in Figure 9.10—are displayed in the Property Browser under the "Misc" category. If a property is intended for run time only, you can prevent it from appearing in the Property Browser by adorning the property with BrowsableAttribute:

```
[BrowsableAttribute(false)]
public bool IsItTimeForABreak {
  get { ... }
  set { ... }
}
```

With IsItTimeForABreak out of the design-time picture, only the custom Alarm property remains. However, it's currently listed under the Property Browser's Misc category and lacks a description. You can improve the situation by applying both CategoryAttribute and DescriptionAttribute:

```
[
  CategoryAttribute("Behavior"),
  DescriptionAttribute("Alarm for late risers")
]
public DateTime Alarm {
  get { ... }
  set { ... }
}
```

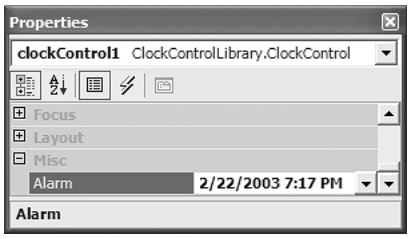

FIGURE 9.10: Visual Studio.NET with a Clock Control Chosen

TABLE 9.1: Design-Time Property Browser Attributes

Attribute	Description
AmbientValueAttribute	Specifies the value for this property that causes it to acquire its value from another source, usually its container (see the section titled Ambient Properties in Chapter 8: Controls).
BrowsableAttribute	Determines whether the property is visible in the Property Browser.
CategoryAttribute	Tells the Property Browser which group to include this property in.
DescriptionAttribute	Provides text for the Property Browser to display in its description bar.
DesignOnlyAttribute	Specifies that the design-time value of this property is serialized to the form's resource file. This attribute is typically used on properties that do not exist at run time.
MergablePropertyAttribute	Allows this property to be combined with properties from other objects when more than one are selected and edited.
ParenthesizePropertyNameAttribute	Specifies whether this property should be surrounded by parentheses in the Property Browser.
ReadOnlyAttribute	Specifies that this property cannot be edited in the Property Browser.

After adding these attributes and rebuilding, you will notice that the Alarm property has relocated to the desired category in the Property Browser, and the description appears on the description bar when you select the property (both shown in Figure 9.11). You can actually use CategoryAttribute to create new categories, but you should do so only if the existing categories don't suitably describe a property's purpose. Otherwise, you'll confuse users looking for your properties in the logical category.

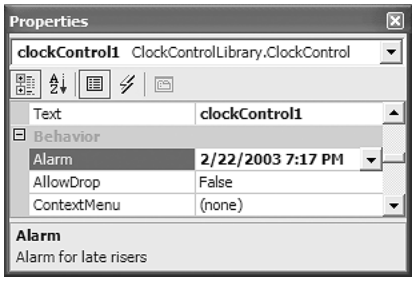

FIGURE 9.11: Alarm Property with CategoryAttribute and DescriptionAttribute Applied

In Figure 9.11, some property values are shown in boldface and others are not. Boldface values are those that differ from the property's default value, which is specified by DefaultValueAttribute:

```
 [
  CategoryAttribute("Appearance"),
  DescriptionAttribute("Whether digital time is shown"),
  DefaultValueAttribute(true)
]
public bool ShowDigitalTime {
  get { ... }
  set { ... }
}
```

Using DefaultValueAttribute also allows you to reset a property to its default value using the Property Browser, which is available from the property's context menu, as shown in Figure 9.12.

This option is disabled if the current property is already the default value. Default values represent the most common value for a property.

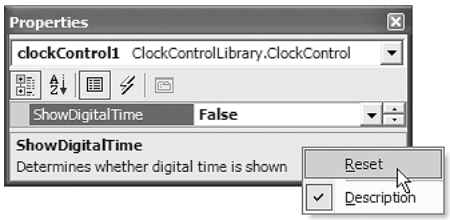

FIGURE 9.12: Resetting a Property to Its Default Value

Some properties, such as Alarm or Text, simply don't have a default that's possible to define, whereas others, such as Enabled and ControlBox, do.

Just like properties, a class can have defaults. You can specify a default event by adorning a class with DefaultEventAttribute:

```
[DefaultEventAttribute("AlarmSounded")]
class ClockControl : Control { ... }
```

Double-clicking the component causes the Designer to automatically hook up the default event; it does this by serializing code to register with the specified event in InitializeComponent and providing a handler for it:

```
class ClockControlHostForm : Form {
  ...
  void InitializeComponent() {
    ...
    this.clockControl1.AlarmSounded +=
      new AlarmHandler(this.clockControl1_AlarmSounded);
    ...
  }
  ...
  void clockControl1_AlarmSounded(
    object sender,
    ClockControlLibrary.AlarmType type) {
  }
  ...
}
```

You can also adorn your component with DefaultPropertyAttribute:

```
[DefaultPropertyAttribute("ShowDigitalTime")]
public class ClockControl : Windows.Forms.Control { ... }
```

This attribute causes the Designer to highlight the default property when the component's property is first edited, as shown in Figure 9.13.

Default properties aren't terribly useful, but setting the correct default event properly can save a developer's time when using your component.

Code Serialization

Whereas DefaultEventAttribute and DefaultPropertyAttribute affect the behavior only of the Property Browser, DefaultValueAttribute serves a dual purpose: It also plays a role in helping the Designer determine which

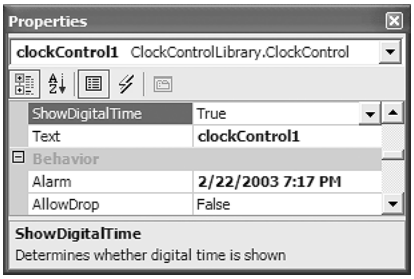

FIGURE 9.13: Default Property Highlighted in the Property Browser

code is serialized to InitializeComponent. Properties that don't have a default value are automatically included in InitializeComponent. Those that have a default value are included only if the property's value differs from the default. To avoid unnecessarily changing a property, your initial property values should match the value set by DefaultValueAttribute.

DesignerSerializationVisibilityAttribute is another attribute that affects the code serialization process. The DesignerSerializationVisibilityAttribute constructor takes a value from the DesignerSerializationVisibility enumeration:

```
enum DesignerSerializationVisibility {
  Visible, // initialize this property if nondefault value
  Hidden, // don't initialize this property
  Content // initialize sets of properties on a subobject
}
```

The default, Visible, causes a property's value to be set in InitializeComponent if the value of the property is not the same as the value of the default. If you'd prefer that no code be generated to initialize a property, use Hidden:

```
[
  DefaultValueAttribute(true),
  DesignerSerializationVisibilityAttribute(
    DesignerSerializationVisibility.Hidden)
]
public bool ShowDigitalTime {
  get { ... }
  set { ... }
}
```

You can use Hidden in conjunction with BrowsableAttribute set to false for run-time-only properties. Although BrowsableAttribute determines whether a property is visible in the Property Browser, its value may still be serialized unless you prevent that by using Hidden.

By default, properties that maintain a collection of custom types cannot be serialized to code. Such a property is implemented by the clock control in the form of a "messages to self" feature, which captures a set of messages and displays them at the appropriate date and time. To enable serialization of a collection, you can apply DesignerSerializationVisibility. Content to instruct the Designer to walk into the property and serialize its internal structure:

```
[
  CategoryAttribute("Behavior"),
  DescriptionAttribute ("Stuff to remember for later"),
  DesignerSerializationVisibilityAttribute
       (DesignerSerializationVisibility.Content)
]
public MessageToSelfCollection MessagesToSelf {
  get { ... }
  set { ... }
}
```

The generated InitializeComponent code for a single message looks like this:

```
void InitializeComponent() {
  ...
  this.clockControl1.MessagesToSelf.AddRange(
  new ClockControlLibrary.MessageToSelf[] {
    new ClockControlLibrary.MessageToSelf(
      new System.DateTime(2003, 2, 22, 21, 55, 0, 0), "Wake up")});
  ...
}
```

This code also needs a "translator" class to help the Designer serialize the code to construct a MessageToSelf type. This is covered in detail in the section titled "Type Converters" later in this chapter.

Host Form Integration

While we're talking about affecting code serialization, there's another trick that's needed for accessing a component's hosting form. For example, consider a clock control and a clock component, both of which offer the ability to place the current time in the hosting form's caption. Each needs to acquire a reference to the host form to set the time in the form's Text property. The control comes with native support for this requirement:

```
Form hostingForm = this.Parent as Form;
```

Unfortunately, components do not provide a similar mechanism to access their host form. At design time, the component can find the form in the designer host's Container collection. However, this technique will not work at run time because the Container is not available at run time. To get its container at run time, a component must take advantage of the way the Designer serializes code to the InitializeComponent method. You can write code that takes advantage of this infrastructure to seed itself with a reference to the host form at design time and run time. The first step is to grab the host form at design time using a property of type Form:

```
Form hostingForm = null;

[BrowsableAttribute(false)]
public Form HostingForm {
  // Used to populate InitializeComponent at design time
  get {
    if( (hostingForm == null) && this.DesignMode ) {
     // Access designer host and obtain reference to root component
      IDesignerHost designer =
        this.GetService(typeof(IDesignerHost)) as IDesignerHost;
      if( designer != null ) {
        hostingForm = designer.RootComponent as Form;
      }
    }
    return hostingForm;
  }

  set {...}
}
```

The HostingForm property is used to populate the code in InitializeComponent at design time, when the designer host is available. Stored in

the designer host's RootComponent property, the root component represents the primary purpose of the Designer. For example, a Form component is the root component of the Windows Forms Designer. DesignerHost.RootComponent is a helper function that allows you to access the root component without enumerating the Container collection. Only one component is considered the root component by the designer host. Because the HostingForm property should go about its business transparently, you should decorate it with BrowsableAttribute set to false, thereby ensuring that the property is not editable from the Property Browser.

Because HostForm is a public property, the Designer retrieves HostForm's value at design time to generate the following code, which is needed to initialize the component:

```
void InitializeComponent() {
  ...
  this.myComponent1.HostingForm = this;
  ...
}
```

At run time, when InitializeComponent runs, it will return the hosting form to the component via the HostingForm property setter:

```
Form hostingForm = null;

[BrowsableAttribute(false)]
public Form HostingForm {
  get { ... }

  // Set by InitializeComponent at run time
  set {
    if( !this.DesignMode ) {
      // Don't change hosting form at run time
      if( (hostingForm != null) && (hostingForm != value) ) {
        throw new
          InvalidOperationException
            ("Can't set HostingForm at run time.");
      }
    }
    else hostingForm = value;
  }
}
```

In this case, we're using our knowledge of how the Designer works to trick it into handing our component a value at run-time that we pick at design-time.

Batch Initialization

As you may have noticed, the code that eventually gets serialized to InitializeComponent is laid out as an alphanumerically ordered sequence of property sets, grouped by object. Order isn't important until your component exposes range-dependent properties, such as Min/Max or Start/Stop pairs. For example, the clock control also has two dependent properties: PrimaryAlarm and BackupAlarm (the Alarm property was split into two for extra sleepy people).

Internally, the clock control instance initializes the two properties 10 minutes apart, starting from the current date and time:

```
DateTime primaryAlarm = DateTime.Now;
DateTime backupAlarm = DateTime.Now.AddMinutes(10);
```

Both properties should check to ensure that the values are valid:

```
public DateTime PrimaryAlarm {
  get { return primaryAlarm; }
  set {
    if( value >= backupAlarm )
      throw new ArgumentOutOfRangeException
           ("Primary alarm must be before Backup alarm");
    primaryAlarm = value;
  }
}

public DateTime BackupAlarm {
  get { return backupAlarm; }
  set {
    if( value < primaryAlarm )
      throw new ArgumentOutOfRangeException
           ("Backup alarm must be after Primary alarm");
    backupAlarm = value;
  }
}
```

With this dependence checking in place, at design time the Property Browser will show an exception in an error dialog if an invalid property is entered, as shown in Figure 9.14.

This error dialog is great at design time, because it lets the developer know the relationship between the two properties. However, there's a problem when the properties are serialized into InitializeComponent alphabetically:

```
void InitializeComponent() {
  ...
  // clockControl1
  this.clockControl1.BackupAlarm =
       new System.DateTime(2003, 11, 24, 13, 42, 47, 46);
  ...
  this.clockControl1.PrimaryAlarm =
       new System.DateTime(2003, 11, 24, 13, 57, 47, 46);
  ...
}
```

Notice that even if the developer sets the two alarms properly, as soon as BackupAlarm is set and is checked against the default value of PrimaryAlarm, a run-time exception will result.

To avoid this, a component must be notified when its properties are being set from InitializeComponent in "batch mode" so that they can be validated all at once at the end. Implementing the ISupportInitialize interface (from the System.ComponentModel namespace) provides this capability, with two notification methods to be called before and after initialization:

```
public interface ISupportInitialize {
  public void BeginInit();
```

continues

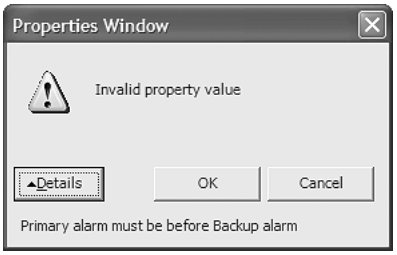

FIGURE 9.14: Invalid Value Entered into the Property Browser

```
  public void EndInit();
}
```

When a component implements this interface, calls to BeginInit and EndInit are serialized to InitializeComponent:

```
void InitializeComponent() {
  ...
  ((System.ComponentModel.ISupportInitialize)
       (this.clockControl1)).BeginInit();
  ...
  // clockControl1
  this.clockControl1.BackupAlarm =
       new System.DateTime(2003, 11, 24, 13, 42, 47, 46);
  ...
  this.clockControl1.PrimaryAlarm =
       new System.DateTime(2003, 11, 24, 13, 57, 47, 46);
  ...
  ((System.ComponentModel.ISupportInitialize)
       (this.clockControl1)).EndInit();
  ...
}
```

The call to BeginInit signals the entry into initialization batch mode, a signal that is useful for turning off value checking:

```
public class ClockControl : Control, ISupportInitialize {
  ...
  bool initializing = false;
  ...
  void BeginInit() { initializing = true; }
  ...
  public DateTime PrimaryAlarm {
    get { ... }
    set {
      if( !initializing ) { /* check value */ }
      primaryAlarm = value;
    }
  }

  public DateTime BackupAlarm {
    get { ... }
    set {
      if( !initializing ) { /* check value */ }
      backupAlarm = value;
    }
  }
}
```

Placing the appropriate logic into EndInit performs batch validation:

```
public class ClockControl : Control, ISupportInitialize {
  void EndInit() {
    if( primaryAlarm >= backupAlarm )
      throw new ArgumentOutOfRangeException
           ("Primary alarm must be before Backup alarm");
  }
  ...
}
```

EndInit also turns out to be a better place to avoid the timer's Tick event, which currently fires once every second during design time. Although the code inside the Tick event handler doesn't run at design time (because it's protected by a check of the DesignMode property), it would be better not to even start the timer at all until run time. However, because DesignMode can't be checked in the constructor, a good place to check it is in the EndInit call, which is called after all properties have been initialized at run time or at design time:

```
public class ClockControl : Control, ISupportInitialize {
  ...
  void EndInit() {
    ...
    if( !this.DesignMode ) {
      // Initialize timer
      timer.Interval = 1000;
      timer.Tick += new System.EventHandler(this.timer_Tick);
      timer.Enabled = true;
    }
  }
}
```

The Designer and the Property Browser provide all kinds of design-time help to augment the experience of developing a component, including establishing how a property is categorized and described to the developer and how it's serialized for the InitializeComponent method.

Extender Property Providers

So far the discussion has focused on the properties implemented by a control for itself. TimeZoneModifier, an example of such a property, allows the

clock control to be configured to display the time in any time zone. One way to use this feature is to display the time in each time zone where your organization has offices. If each office were visually represented with a picture box, you could drag one clock control for each time zone onto the form, manually adjusting the TimeZoneModifier property on each clock control. The result might look like Figure 9.15.

This works quite nicely but could lead to real estate problems, particularly if you have one clock control for each of the 24 time zones globally and, consequently, 24 implementations of the same logic on the form. If you are concerned about resources, this also means 24 system timers. Figure 9.16 shows what this might look like.

Another approach is to have a single clock control and update its TimeZoneModifier property with the relevant time zone from the Click event of each picture box. This is a cumbersome approach because it requires developers to write the code associating a time zone offset with each control, a

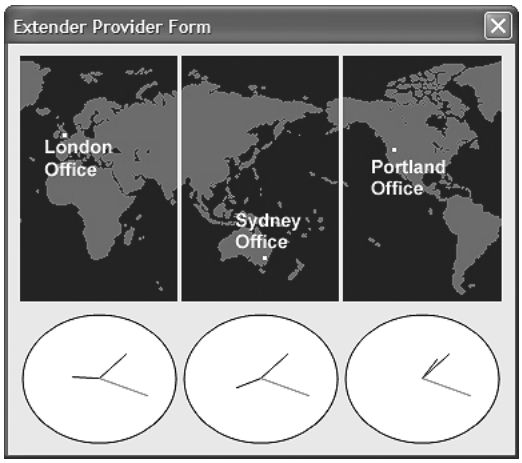

FIGURE 9.15: Form with Multiple Time Zones

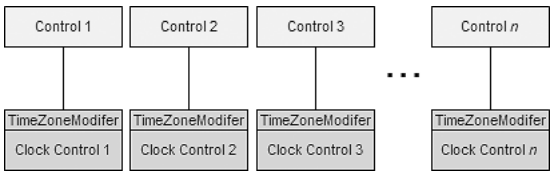

FIGURE 9.16: One Provider Control for Each Client Control

situation controls are meant to help avoid. Figure 9.17 illustrates this approach.

A nicer way to handle this situation is to provide access to a single implementation of the clock control without forcing the developer to write additional property update code. .NET offers extender property support to do just this, allowing components to extend property implementations to other components.

Logically, an *extender property* is a property provided by an extender component, like the clock control, on other components in the same container, like picture boxes. Extender properties are useful whenever a component needs data from a set of other components in the same host. For example, WinForms itself provides several extender components, including ErrorProvider, HelpProvider, and ToolTip. In the case of the ToolTip component, it makes a lot more sense to set the ToolTip property on each control on a form than it does to try to set tooltip information for each control using an editor provided by the ToolTip component itself.

In our case, by implementing TimeZoneModifier as an extender property, we allow each picture box control on the form to get its own value, as shown in Figure 9.18.

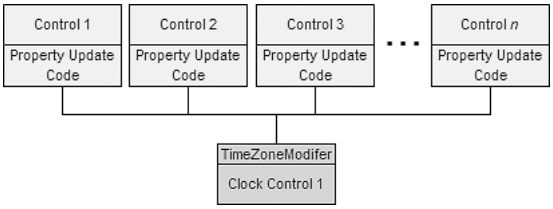

FIGURE 9.17: One Provider Control for All Client Controls, Accessed with Code

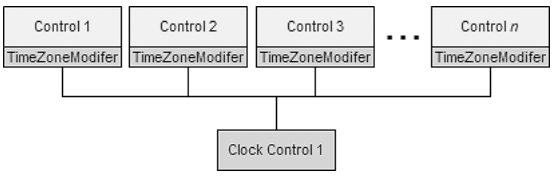

FIGURE 9.18: One Provider Control for All Client Controls, Accessed with a Property Set

Exposing an extender property from your control requires that you first use ProvidePropertyAttribute to declare the property to be extended:

```
[ProvidePropertyAttribute("TimeZoneModifier", typeof(PictureBox))]
public class ClockControl : Control { ... }
```

The first parameter to the attribute is the name of the property to be extended. The second parameter is the "receiver" type, which specifies the type of object to extend, such as PictureBox. Only components of the type specified by the receiver can be extended. If you want to implement a more sophisticated algorithm, such as supporting picture boxes and panels, you must implement the IExtenderProvider CanExtend method:

```
class ClockControl : ..., IExtenderProvider {
  bool IExtenderProvider.CanExtend(object extendee) {
    // Don't extend self
    if( extendee == this ) return false;

    // Extend suitable controls
    return( (extendee is PictureBox) ||
            (extendee is Panel) );
  }
  ...
}
```

As you saw in Figure 9.18, the provider supports one or more extendee controls. Consequently, the provider control must be able to store and distinguish one extendee's property value from that of another. It does this in the Get<PropertyName> and Set<PropertyName> methods, in which PropertyName is the name you provided in ProvidePropertyAttribute. Then GetTimeZoneModifier simply returns the property value when requested by the Property Browser:

```
public class ClockControl : Control, IExtenderProvider {
  // Mapping of components to numeric timezone offsets
  HashTable timeZoneModifiers = new HashTable();

  public int GetTimeZoneModifier(Control extendee) {
    // Return component's timezone offset
    return int.Parse(timeZoneModifiers[extendee]);
  }
  ...
}
```

SetTimeZoneModifier has a little more work to do. Not only does it put the property value into a new hash table for the extendee when provided, but it also removes the hash table entry when the property is cleared. Also, with the sample TimeZoneModifier property, you need to hook into each extendee control's Click event, unless the control isn't using the extender property. SetTimeZoneModifier is shown here:

```
class ClockControl : ..., IExtenderProvider {
  HashTable timeZoneModifiers = new HashTable();
  ...
  public void SetTimeZoneModifier(Control extendee, object value) {
    // If property isn't provided
    if( value == null ) {
      // Remove it
      timeZoneModifiers.Remove(extendee);
      if( !this.DesignMode ) {
        extendee.Click -= new EventHandler(extendee_Click);
      }
    }
    else {
      // Add the offset as an integer
      timeZoneModifiers[extendee] = int.Parse(value);
      if( !this.DesignMode ) {
        extendee.Click += new EventHandler(extendee_Click);
      }
    }
  }
}
```

As with other properties, you can affect the appearance of an extender property in the Property Browser by adorning the Get<PropertyName> method with attributes:

```
class ClockControl : ..., IExtenderProvider {
  [
    Category("Behavior"),
    Description("Sets the timezone difference from the current time"),
    DefaultValue("")
  ]
  public int GetTimeZoneModifier(Control extendee) { ... }
  ...
}
```

These attributes are applied to the extendee's Property Browser view.

With all this in place, you can compile your extender component to see the results. Extended properties will appear in the extendee component's properties with the following naming format:

```
<ExtendedPropertyName> on <ExtenderProviderName>
```

Figure 9.19 shows the TimeZoneModifier extender property behaving like any other property on a PictureBox control.

If a property is set and is not the default value, it is serialized to InitializeComponent(), as a SetTimeZoneModifier method call, and grouped with the extendee component:

```
void InitializeComponent() {
  ...
  this.clockControl1.SetTimeZoneModifier(this.pictureBox1, -11);
  ...
}
```

Extender properties allow a component to add to the properties of other components in the same host. In this way, the developer can keep the data with the intuitive component, which is not necessarily the component that provides the service.

Type Converters

When you select a component on a design surface, the entries in the Property Browser are rendered from the design-time control instance. When you edit properties in the Property Browser, the component's design-time instance is updated with the new property values. This synchronicity isn't

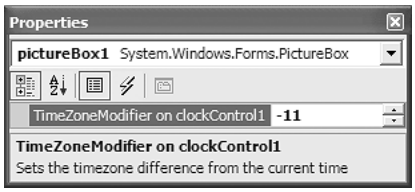

FIGURE 9.19: Extended Property in Action

as straightforward as it seems, however, because the Property Browser displays properties only as text, even though the source properties can be of any type. As values shuttle between the Property Browser and the design-time instance, they must be converted back and forth between the string type and the type of the property.

Enter the *type converter*, the translator droid of .NET, whose main goal in life is to convert between types. For string-to-type conversion, a type converter is used for each property displayed in the Property Browser, as shown in Figure 9.20.

.NET offers the TypeConverter class (from the System.ComponentModel namespace) as the base implementation type converter. .NET also gives you several derivations—including StringConverter, Int32Converter, and DateTimeConverter—that support conversion between common .NET types. If you know the type that needs conversion at compile time, you can create an appropriate converter directly:

```
// Type is known at compile time
TypeConverter converter = new Int32Converter();
```

Or, if you don't know the type that needs conversion until run time, let the TypeDescriptor class (from the System.ComponentModel namespace) make the choice for you:

```
// Don't know the type before run time
object myData = 0;
TypeConverter converter = TypeDescriptor.GetConverter(myData.GetType());
```

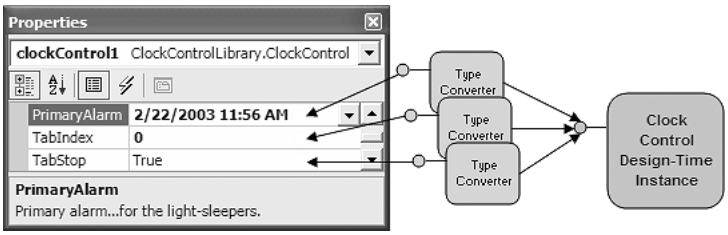

FIGURE 9.20: The Property Browser and Design-Time Conversion

The TypeDescriptor class provides information about a particular type or object, including methods, properties, events, and attributes. TypeDescriptor.GetConverter evaluates a type to determine a suitable TypeConverter based on the following:

1. Checking whether a type is adorned with an attribute that specifies a particular type converter.
2. Comparing the type against the set of built-in type converters.
3. Returning the TypeConverter base if no other type converters are found.

Because the Property Browser is designed to display the properties of any component, it can't know specific property types in advance. Consequently, it relies on TypeDescriptor.GetConverter to dynamically select the most appropriate type converter for each property.

After a type converter is chosen, the Property Browser and the design-time instance can perform the required conversions, using the same fundamental steps as those shown in the following code:

```
// Create the appropriate type converter
object myData = 0;
TypeConverter converter = TypeDescriptor.GetConverter(myData.GetType());

// Can converter convert int to string?
if( converter.CanConvertTo(typeof(string)) ) {
  // Convert it
  object intToString = converter.ConvertTo(42, typeof(string));
}

// Can converter convert string to int?
if( converter.CanConvertFrom(typeof(string)) ) {
  // Convert it
  object stringToInt = converter.ConvertFrom("42");
}
```

When the Property Browser renders itself, it uses the type converter to convert each design-time instance property to a string representation using the following steps:

1. CanConvertTo: Can you convert from the design-time property type to a string?
2. ConvertTo: If so, please convert property value to string.

The string representation of the source value is then displayed at the property's entry in the Property Browser. If the property is edited and the value is changed, the Property Browser uses the next steps to convert the string back to the source property value:

1. CanConvertFrom: Can you convert back to the design-time property type?
2. ConvertFrom: If so, please convert string to property value.

Some intrinsic type converters can do more than just convert between simple types. To demonstrate, let's expose a Face property of type ClockFace, allowing developers to decide how the clock is displayed, including options for Analog, Digital, or Both:

```
public enum ClockFace {
  Analog = 0,
  Digital = 1,
  Both = 2
}

class ClockControl : Control {
  ClockFace face = ClockFace.Both;
  public ClockFace Face {
    get { … }
    set { … }
  }
  ...
}
```

TypeDescriptor.GetConverter returns an EnumConverter, which contains the smarts to examine the source enumeration and convert it to a drop-down list of descriptive string values, as shown in Figure 9.21.

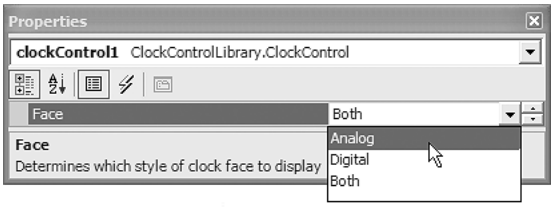

FIGURE 9.21: Enumeration Type Displayed in the Property Browser via EnumConverter

Custom Type Converters

Although the built-in type converters are useful, they aren't enough if your component or control exposes properties based on custom types, such as the clock control's HourHand, MinuteHand, and SecondHand properties, shown here:

```
public class Hand {
  public Hand(Color color, int width) {
    this.color = color;
    this.width = width;
  }
  public Color Color {
    get { return color; }
    set { color = value; }
  }
  public int Width {
    get { return width; }
    set { width = value; }
  }
  Color  color = Color.Black;
  int    width = 1;
}

public class ClockControl : Control {
  public Hand HourHand { ... }
  public Hand MinuteHand { ... }
  public Hand SecondHand { ... }
}
```

The idea is to give developers the option to pretty up the clock's hands with color and width values. Without a custom type converter,[5] the unfortunate result is shown in Figure 9.22.

5. Be careful when you use custom types for properties. If the value of the property is null, you won't be able to edit it in the Property Browser at all.

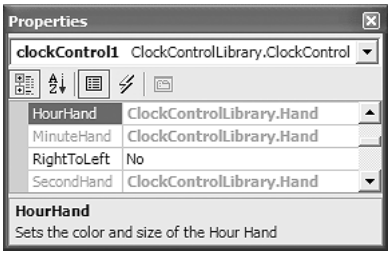

FIGURE 9.22: Complex Properties in the Property Browser

Just as the Property Browser can't know which types it will be displaying, .NET can't know which custom types you'll be developing. Consequently, there aren't any type of converters capable of handling them. However, you can hook into the type converter infrastructure to provide your own. Building a custom type converter starts by deriving from the TypeConverter base class:

```
public class HandConverter : TypeConverter { ... }
```

To support conversion, HandConverter must override CanConvertFrom, ConvertTo, and ConvertFrom:

```
public class HandConverter : TypeConverter {
  public override bool
    CanConvertFrom(
      ITypeDescriptorContext context, Type sourceType) {...}

  public override object
    ConvertFrom(
      ITypeDescriptorContext context,
      CultureInfo info,
      object value) {...}

  public override object
    ConvertTo(
      ITypeDescriptorContext context,
      CultureInfo culture,
      object value,
      Type destinationType) {...}
}
```

CanConvertFrom lets clients know what it can convert from. In this case, HandConverter reports that it can convert from a string type to a Hand type:

```
public override bool CanConvertFrom(
  ITypeDescriptorContext context, Type sourceType) {

  // We can convert from a string to a Hand type
  if( sourceType == typeof(string) ) { return true; }
  return base.CanConvertFrom(context, sourceType);
}
```

Whether the string type is in the correct format is left up to ConvertFrom, which actually performs the conversion. HandConverter expects a multivalued string. It splits this string into its atomic values and then uses it to instantiate a Hand object:

```
public override object ConvertFrom(
  ITypeDescriptorContext context, CultureInfo info, object value) {

    // If converting from a string
    if( value is string ) {
      // Build a Hand type
      try {
        // Get Hand properties
        string propertyList = (string)value;
        string[] properties = propertyList.Split(';');
        return new Hand(Color.FromName(properties[0].Trim()),
                        int.Parse(properties[1]));
      }
      catch {}
      throw new ArgumentException("The arguments were not valid.");
    }
    return base.ConvertFrom(context, info, value);
  }
  ...
}
```

ConvertTo converts from a Hand type back to a string:

```
public override object ConvertTo(
  ITypeDescriptorContext context,
  CultureInfo culture,
  object value,
  Type destinationType) {
```

```
  // If source value is a Hand type
  if( value is Hand ) {
    // Convert to string
    if( (destinationType == typeof(string)) ) {
      Hand hand = (Hand)value;
      string color = (hand.Color.IsNamedColor ?
                      hand.Color.Name :
                      hand.Color.R + ", " +
                        hand.Color.G + ", " +
                        hand.Color.B);
      return string.Format("{0}; {1}", color, hand.Width.ToString());
    }
  }
  return base.ConvertTo(context, culture, value, destinationType);
}
```

You may have noticed that HandConverter doesn't implement a CanConvertTo override. The base implementation of TypeConverter.CanConvertTo returns a Boolean value of true when queried for its ability to convert to a string type. Because this is the right behavior for HandConverter (and for most other custom type converters), there's no need to override it.

When the Property Browser looks for a custom type converter, it looks at each property for a TypeConverterAttribute:

```
public class ClockControl : Control {
  ...
  [ TypeConverterAttribute (typeof(HandConverter)) ]
  public Hand HourHand { ... }

  [TypeConverterAttribute (typeof(HandConverter)) ]
  public Hand MinuteHand { ... }

  [TypeConverterAttribute (typeof(HandConverter)) ]
  public Hand SecondHand { ... }
  ...
}
```

However, this is somewhat cumbersome, so it's simpler to decorate the type itself with TypeConverterAttribute:

```
[ TypeConverterAttribute(typeof(HandConverter)) ]
public class Hand { ... }
```

continues

```
public class ClockControl : Control {
  ...
  public Hand HourHand { ... }
  public Hand MinuteHand { ... }
  public Hand SecondHand { ... }
  ...
}
```

Figure 9.23 shows the effect of the custom HandConverter type converter.

Expandable Object Converter

Although using the UI shown in Figure 9.23 is better than not being able to edit the property at all, there are still ways it can be improved. For instance, put yourself in a developer's shoes. Although it might be obvious what the first part of the property is, it's disappointing not to be able to pick the color from one of those pretty drop-down color pickers. And what is the second part of the property meant to be? Length, width, degrees, something else?

As an example of what you'd like to see, the Font type supports browsing and editing of its subproperties, as shown in Figure 9.24.

This ability to expand a property of a custom type makes it a lot easier to understand what the property represents and what sort of values you need to provide. To allow subproperty editing, you simply change the base type from TypeConverter to ExpandableObjectConverter (from the System.ComponentModel namespace):

```
public class HandConverter : ExpandableObjectConverter { ... }
```

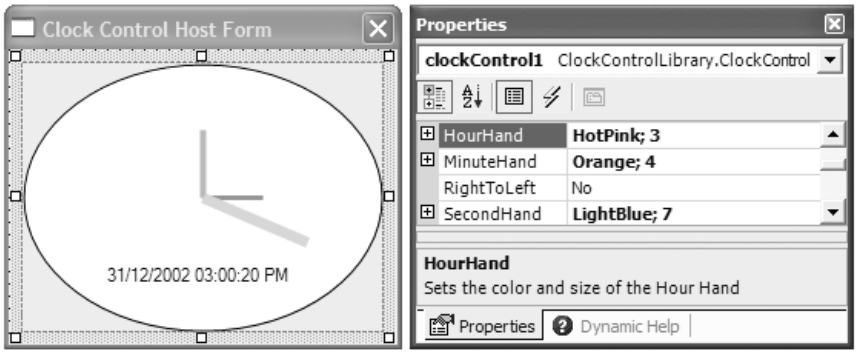

FIGURE 9.23: HandConverter in Action (See Plate 23)

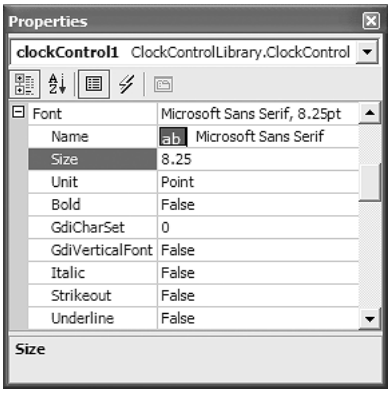

FIGURE 9.24: Expanded Property Value

This change gives you multivalue and nested property editing support, as shown in Figure 9.25.

Although you don't have to write any code to make this property expandable, you must write a little code to fix an irksome problem: a delay in property updating. In expanded mode, a change to the root property value is automatically reflected in the nested property value list. This occurs because the root property entry refers to the design-time property instance, whereas its nested property values refer to the design-time instance's properties directly, as illustrated in Figure 9.26.

When the root property is edited, the Property Browser calls HandConverter.ConvertFrom to convert the Property Browser's string entry to a new SecondHand instance, and that results in a refresh of the Property Browser. However, changing the nested values only changes the current instance's property values, rather than creating a new instance, and that doesn't result in an immediate refresh of the root property.

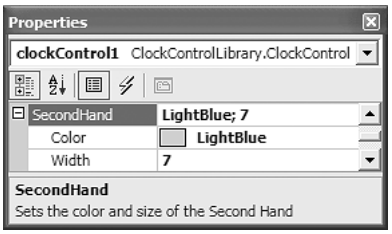

FIGURE 9.25: HandConverter Derived from ExpandableObjectConverter

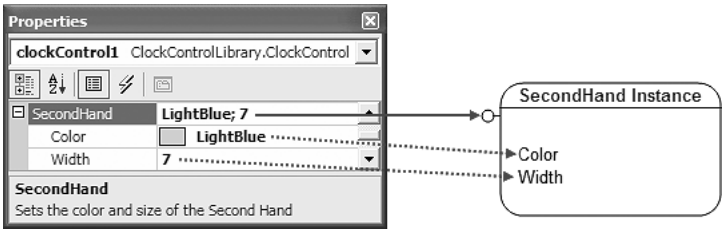

FIGURE 9.26: Relationship between Root and Nested Properties and Design-Time Property Instance

TypeConverters offer a mechanism you can use to force the creation of a new instance whenever instance property values change, something you achieve by overriding GetCreateInstanceSupported and CreateInstance. The GetCreateInstanceSupported method returns a Boolean indicating whether this support is available and, if it is, calls CreateInstance to implement it:

```
public class HandConverter : ExpandableObjectConverter {
  public override bool
  GetCreateInstanceSupported(
    ITypeDescriptorContext context) {

    // Always force a new instance
    return true;
  }

  public override object
  CreateInstance(
    ITypeDescriptorContext context, IDictionary propertyValues) {

    // Use the dictionary to create a new instance
    return new Hand(
      (Color)propertyValues["Color"],
      (int)propertyValues["Width"]);
  }
  ...
}
```

If GetCreateInstanceSupported returns true, then CreateInstance will be used to create a new instance whenever any of the subproperties of an expandable object are changed. The propertyValues argument to CreateInstance provides a set of name/value pairs for the current values of the object's subproperties, and you can use them to construct a new instance.

Custom Type Code Serialization with TypeConverters

Although the Hand type now plays nicely with the Property Browser, it doesn't yet play nicely with code serialization. In fact, at this point it's not being serialized to InitializeComponent at all. To enable serialization of properties exposing complex types, you must expose a public ShouldSerialize<PropertyName> method that returns a Boolean:

```
public class ClockControl : Control {
  public Hand SecondHand { ... }
  bool ShouldSerializeSecondHand() {
    // Only serialize nondefault values
    return(
      (secondHand.Color != Color.Red) || (secondHand.Width != 1) );
  }
  ...
}
```

Internally, the Designer looks for a method named ShouldSerialize <PropertyName> to ask whether the property should be serialized. From the Designer's point of view, it doesn't matter whether your ShouldSerialize<PropertyName> is public or private, but choosing private removes it from client visibility.

To programmatically implement the Property Browser reset functionality, you use the Reset<PropertyName> method:

```
public Hand SecondHand { ... }

void ResetSecondHand() {
  SecondHand = new Hand(Color.Red, 1);
}
```

Implementing ShouldSerialize lets the design-time environment know whether the property should be serialized, but you also need to write custom code to help assist in the generation of appropriate InitializeComponent code. Specifically, the Designer needs an *instance descriptor*, which provides the information needed to create an instance of a particular type. The code serializer gets an InstanceDescriptor object for a Hand by asking the Hand type converter:

```
public class HandConverter : ExpandableObjectConverter {
  public override bool
```

continues

```
    CanConvertTo(
      ITypeDescriptorContext context, Type destinationType) {

    // We can be converted to an InstanceDescriptor
    if( destinationType == typeof(InstanceDescriptor) ) return true;
    return base.CanConvertTo(context, destinationType);
  }

  public override object
    ConvertTo(
      ITypeDescriptorContext context, CultureInfo culture,
      object value, Type destinationType) {

    if( value is Hand ) {
      // Convert to InstanceDescriptor
      if( destinationType == typeof(InstanceDescriptor) ) {
        Hand     hand = (Hand)value;
        object[] properties = new object[2];
        Type[]   types = new Type[2];

        // Color
        types[0] = typeof(Color);
        properties[0] = hand.Color;

        // Width
        types[1] = typeof(int);
        properties[1] = hand.Width;

        // Build constructor
        ConstructorInfo ci = typeof(Hand).GetConstructor(types);
        return new InstanceDescriptor(ci, properties);
      }
      ...
    }
   return base.ConvertTo(context, culture, value, destinationType);
  }
  ...
}
```

To be useful, an instance descriptor requires two pieces of information. First, it needs to know what the constructor looks like. Second, it needs to know which property values should be used if the object is instantiated. The former is described by the ConstructorInfo type, and the latter is simply an array of values, which should be in constructor parameter order. After the control is rebuilt and assuming that ShouldSerialize<PropertyName> permits, all Hand type properties will be serialized using the information provided by the HandConverter-provided InstanceDescriptor:

```
public class ClockControlHostForm : Form {
  ...
  void InitializeComponent() {
    ...
    this.clockControl1.HourHand =
      new ClockControlLibrary.Hand(System.Drawing.Color.Black, 2);
    ...
  }
}
```

Type converters provide all kinds of help for the Property Browser and the Designer to display, convert, and serialize properties of custom types for components that use such properties.

UI Type Editors

ExpandableObjectConverters help break down a complex multivalue property into a nested list of its atomic values. Although this technique simplifies editing of a complicated property, it may not be suitable for other properties that exhibit the following behavior:

- Hard to construct, interpret, or validate, such as a regular expression
- One of a list of values so large it would be difficult to remember all of them
- A visual property, such as a ForeColor, that is not easily represented as a string

Actually, the ForeColor property satisfies all three points. First, it would be hard to find the color you wanted by typing comma-separated integers like 33, 86, 24 or guessing a named color, like PapayaWhip. Second, there are a lot of colors to choose from. Finally, colors are just plain visual.

In addition to supporting in-place editing in the Property Browser, properties such as ForeColor help the developer by providing an alternative UI-based property-editing mechanism. You access this tool, shown in Figure 9.27, from a drop-down arrow in the Property Browser.

The result is a prettier, more intuitive way to select a property value. This style of visual editing is supported by the *UI type editor*, a design-time feature that you can leverage to similar effect. There are two types of "editor"

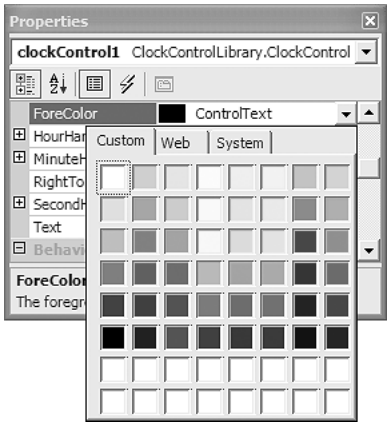

FIGURE 9.27: Color Property Drop-Down UI Editor

you can choose from: modal or drop-down. Drop-down editors support single-click property selection from a drop-down UI attached to the Property Browser. This UI might be a nice way to enhance the clock control's Face property, allowing developers to visualize the clock face style as they make their selection, shown in Figure 9.28.

You begin implementing a custom UI editor by deriving from the UITypeEditor class (from the System.Drawing.Design namespace):

```
public class FaceEditor : UITypeEditor { ... }
```

The next step requires you to override the GetEditStyle and EditValue methods from the UITypeEditor base class:

```
public class FaceEditor : UITypeEditor {
  public override UITypeEditorEditStyle GetEditStyle(
```

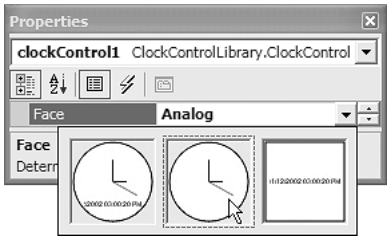

FIGURE 9.28: Custom View Drop-Down UI Editor

```
    ITypeDescriptorContext context)
  {...}

  public override object EditValue(
    ITypeDescriptorContext context,
    IServiceProvider provider,
    object value)
   {...}
}
```

As with type converters, the appropriate UI type editor, provided by the GetEditor method of the TypeDescription class, is stored with each property. When the Property Browser updates itself to reflect a control selection in the Designer, it queries GetEditStyle to determine whether it should show a drop-down button, an open dialog button, or nothing in the property value box when the property is selected. This behavior is determined by a value from the UITypeEditorEditStyle enumeration:

```
enum UITypeEditorEditStyle {
  DropDown, // Display drop-down UI
  Modal, // Display modal dialog UI
  None, // Don't display a UI
}
```

Not overriding GetEditStyle is the same as returning UITypeEditorEditStyle.None, which is the default edit style. To show the drop-down UI editor, the clock control returns UITypeEditorEditStyle.DropDown:

```
public class FaceEditor : UITypeEditor {
  public override UITypeEditorEditStyle GetEditStyle(
    ITypeDescriptorContext context) {

    if( context != null ) return UITypeEditorEditStyle.DropDown;
    return base.GetEditStyle(context);
  }
  ...
}
```

ITypeDescriptorContext is passed to GetEditStyle to provide contextual information regarding the execution of this method, including the following:

- The container and, subsequently, the designer host and its components

- The component design-time instance being shown in the Property Browser
- A PropertyDescriptor type describing the property, including the TypeConverter and UITypeEditor assigned to the component
- A PropertyDescriptorGridEntry type, which is a composite of the PropertyDescriptor and the property's associated grid entry in the Property Browser

Whereas GetEditStyle is used to initialize the way the property behaves, EditValue actually implements the defined behavior. Whether the UI editor is drop-down or modal, you follow the same basic steps to edit the value:

1. Access the Property Browser's UI display service, IWindowsForms-EditorService.
2. Create an instance of the editor UI implementation, which is a control that the Property Browser will display.
3. Pass the current property value to the UI editor control.
4. Ask the Property Browser to display the UI editor control.
5. Choose the value and close the UI editor control.
6. Return the new property value from the editor.

Drop-Down UI Type Editors

Here's how the clock control implements these steps to show a drop-down editor for the Face property:

```
public class FaceEditor : UITypeEditor {
  ...
  public override object EditValue(
    ITypeDescriptorContext context,
    IServiceProvider provider,
    object value) {

    if( (context != null) && (provider != null) ) {
      // Access the Property Browser's UI display service
      IWindowsFormsEditorService editorService =
        (IWindowsFormsEditorService)
          provider.GetService(typeof(IWindowsFormsEditorService));
```

```
      if( editorService!= null ) {
        // Create an instance of the UI editor control
        FaceEditorControl dropDownEditor =
          new FaceEditorControl(editorService);

        // Pass the UI editor control the current property value
        dropDownEditor.Face = (ClockFace)value;

        // Display the UI editor control
        editorService.DropDownControl(dropDownEditor);

        // Return the new property value from the UI editor control
        return dropDownEditor.Face;
      }
    }
    return base.EditValue(context, provider, value);
  }
}
```

When it comes to displaying the UI editor control, you must play nicely in the design-time environment, particularly regarding UI positioning in relation to the Property Browser. Specifically, drop-down UI editors must appear flush against the bottom of the property entry and must be sized to the width of the property entry.

To facilitate this, the Property Browser exposes a service, an implementation of the IWindowsFormsEditorService interface, to manage the loading and unloading of UI editor controls as well as their positioning inside the development environment. The FaceEditor type references this service and calls its DropDownControl method to display the FaceEditorControl, relative to Property's Browser edit box. When displayed, FaceEditorControl has the responsibility of capturing the user selection and returning control to EditValue with the new value. This requires a call to IWindowsFormsEditorService.CloseDropDown from FaceEditorControl, something you do by passing to FaceEditorControl a reference to the IWindowsFormsEditorService interface:

```
public class FaceEditorControl : UserControl {
  ClockFace face = ClockFace.Both;
  IWindowsFormsEditorService editorService = null;
  ...
  public FaceEditorControl(IWindowsFormsEditorService editorService) {
    ...
```

continues

```
    this.editorService = editorService;
  }

  public ClockFace Face {
    get { ... }
    set { ... }
  }

  void picBoth_Click(object sender, System.EventArgs e) {
    face = ClockFace.Both;

    // Close the UI editor control upon value selection
    editorService.CloseDropDown();
  }

  void picAnalog_Click(object sender, System.EventArgs e) {
    face = ClockFace.Analog;

    // Close the UI editor control upon value selection
    editorService.CloseDropDown();
  }

  void picDigital_Click(object sender, System.EventArgs e) {
    face = ClockFace.Digital;

    // Close the UI editor control upon value selection
    editorService.CloseDropDown();
  }
  ...
}
```

The final step is to associate FaceEditor with the Face property by adorning the property with EditorAttribute:

```
[
  CategoryAttribute("Appearance"),
  DescriptionAttribute("Which style of clock face to display"),
  DefaultValueAttribute(ClockFace.Both),
  EditorAttribute(typeof(FaceEditor), typeof(UITypeEditor))
]
public ClockFace Face { ... }
```

Now FaceEditor is in place for the Face property. When a developer edits that property in Propery Browser, it will show a drop-down arrow and the FaceEditorControl as the UI for the developer to use to choose a value of the ClockFace enumeration.

Modal UI Type Editors

Although drop-down editors are suitable for single-click selection, there are times when unrestricted editing is required. In such situations, you would use a modal UITypeEditor implemented as a modal form. For example, the clock control has a digital time format sufficiently complex to edit with a separate dialog outside the Property Browser:

```
public class ClockControl : Control {
  ...
  string digitalTimeFormat = "dd/MM/yyyy hh:mm:ss tt";
  ...
  [
    CategoryAttribute("Appearance"),
    DescriptionAttribute("The digital time format, ..."),
    DefaultValueAttribute("dd/MM/yyyy hh:mm:ss tt"),
  ]
  public string DigitalTimeFormat {
    get { return digitalTimeFormat; }
    set {
      digitalTimeFormat = value;
      this.Invalidate();
    }
  }
}
```

Date and Time format strings are composed of a complex array of format specifiers that are not easy to remember and certainly aren't intuitive in a property browser, as shown in Figure 9.29.

Modal UITypeEditors are an ideal way to provide a more intuitive way to construct hard-to-format property values. By providing a custom form, you give developers whatever editing experience is the most conducive for

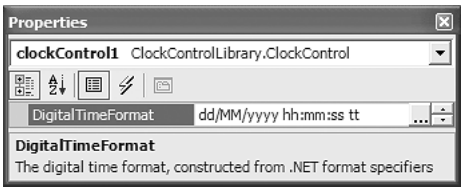

FIGURE 9.29: The DigitalTimeFormat Property

that property type. Figure 9.30 illustrates how the Digital Time Format Editor dialog makes it easier to edit the clock control's DigitTimeFormat property.

A modal UITypeEditor actually requires slightly different code from that of its drop-down counterpart. You follow the same logical steps as with a drop-down editor, with three minor implementation differences:

- Returning UITypeEditorEditStyle.Modal from UITypeEditor.GetEditStyle
- Calling IWindowsFormsEditorService.ShowDialog from EditValue to open the UI editor dialog
- Not requiring an editor service reference to be passed to the dialog, because a Windows Form can close itself

The clock control's modal UI type editor is shown here:

```
public class DigitalTimeFormatEditor : UITypeEditor {
  public override UITypeEditorEditStyle GetEditStyle(
    ITypeDescriptorContext context) {
    if( context != null ) {
      return UITypeEditorEditStyle.Modal;
    }
    return base.GetEditStyle(context);
  }

  public override object EditValue(
    ITypeDescriptorContext context,
    IServiceProvider provider,
    object value) {
```

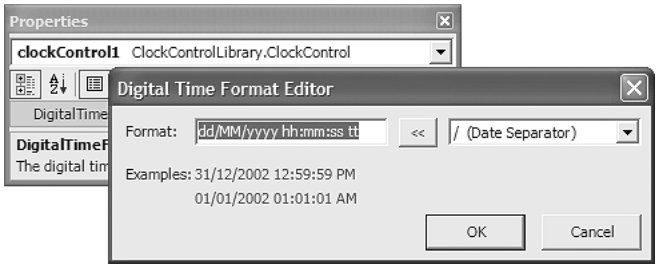

FIGURE 9.30: Custom DigitalTimeFormat Modal UI Editor

```
    if( (context != null) && (provider != null) ) {
      // Access the Property Browser's UI display service
      IWindowsFormsEditorService editorService =
        (IWindowsFormsEditorService)
          provider.GetService(typeof(IWindowsFormsEditorService));

      if( editorService != null ) {
        // Create an instance of the UI editor form
        DigitalTimeFormatEditorForm modalEditor =
          new DigitalTimeFormatEditorForm();

        // Pass the UI editor dialog the current property value
        modalEditor.DigitalTimeFormat = (string)value;

        // Display the UI editor dialog
        if( editorService.ShowDialog(modalEditor) == DialogResult.OK ) {
          // Return the new property value from the UI editor form
          return modalEditor.DigitalTimeFormat;
        }
      }
    }
    return base.EditValue(context, provider, value);
  }
}
```

At this point, normal dialog activities (as covered in Chapter 3: Dialogs) apply for the UI editor's modal form:

```
public class DigitalTimeFormatEditorForm : Form {
  ...
  string digitalTimeFormat = "dd/MM/yyyy hh:mm:ss tt";

  public string DigitalTimeFormat {
    get { return digitalTimeFormat; }
    set { digitalTimeFormat = value; }
  }
  ...
  void btnOK_Click(object sender, System.EventArgs e) {
    DialogResult = DialogResult.OK;
    digitalTimeFormat = txtFormat.Text;
  }
  ...
}
```

Again, to associate the new UI type editor with the property requires applying the EditorAttribute:

```
[
  CategoryAttribute("Appearance"),
  DescriptionAttribute("The digital time format, ..."),
  DefaultValueAttribute("dd/MM/yyyy hh:mm:ss tt"),
  EditorAttribute(typeof(DigitalTimeFormatEditor), typeof(UITypeEditor))
]
public string DigitalTimeFormat { ... }
```

After EditorAttribute is applied, the modal UITypeEditor is accessed via an ellipsis-style button displayed in the Property Browser, as shown in Figure 9.31.

UI type editors allow you to provide a customized editing environment for the developer on a per-property basis, whether it's a drop-down UI to select from a list of possible values or a modal dialog to provide an entire editing environment outside the Property Browser.

Custom Designers

So far, you have seen how properties are exposed to the developer at design time, and you've seen some of the key infrastructure provided by .NET to improve the property-editing experience, culminating in UITypeEditor. Although the focus has been on properties, they aren't the only aspect of a control that operates differently in design-time mode compared with run-time mode. In some situations, a control's UI might render differently between these modes.

For example, the Splitter control displays a dashed border when its BorderStyle is set to BorderStyle.None. This design makes it easier for developers to find this control on the form's design surface in the absence of a visible border, as illustrated in Figure 9.32.

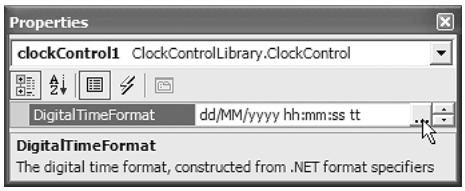

FIGURE 9.31: Accessing a Modal UITypeEditor

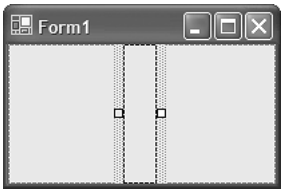

FIGURE 9.32: Splitter Dashed Border When BorderStyle Is None

Because BorderStyle.None means "don't render a border at run time," the dashed border is drawn only at design time for the developer's benefit. Of course, if BorderStyle is set to BorderStyle.FixedSingle or BorderStyle.Fixed3D, the dashed border is not necessary, as illustrated by Figure 9.33.

What's interesting about the splitter control is that the dashed border is not actually rendered from the control implementation. Instead, this work is conducted on behalf of them by a *custom designer*, another .NET design-time feature that follows the tradition, honored by type converters and UI type editors, of separating design-time logic from the control.

Custom designers are not the same as designer hosts or the Windows Forms Designer, although a strong relationship exists between designers and designer hosts. As every component is sited, the designer host creates at least one matching designer for it. As with type converters and UI type editors, the TypeDescriptor class does the work of creating a designer in the CreateDesigner method. Adorning a type with DesignerAttribute ties it to the specified designer. For components and controls that don't possess their own custom designers, .NET provides ComponentDesigner and

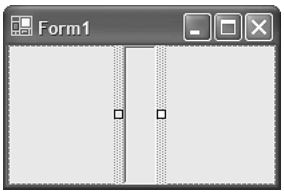

FIGURE 9.33: Splitter with BorderStyle.Fixed3D

ControlDesigner, respectively, both of which are base implementations of IDesigner:

```
public interface IDesigner : IDisposable {
  public void DoDefaultAction();
  public void Initialize(IComponent component);
  public IComponent Component { get; }
  public DesignerVerbCollection Verbs { get; }
}
```

For example, the clock face is round at design time when the clock control either is Analog or is Analog and Digital. This makes it difficult to determine where the edges and corners of the control are, particularly when the clock is being positioned against other controls. The dashed border technique used by the splitter would certainly help, looking something like Figure 9.34.

Because the clock is a custom control, its custom designer will derive from the ControlDesigner base class (from the System.Windows.Forms.Design namespace):

```
public class ClockControlDesigner : ControlDesigner { ... }
```

To paint the dashed border, ClockControlDesigner overrides the Initialize and OnPaintAdornments methods:

```
public class ClockControlDesigner : ControlDesigner {
  ...
```

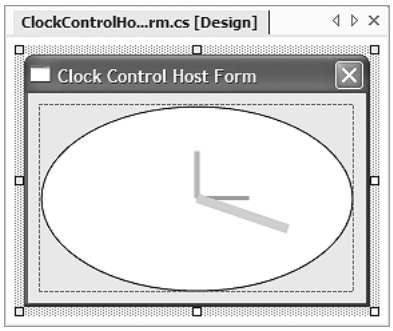

FIGURE 9.34: Border Displayed from ClockControlDesigner

```
  public override void Initialize(IComponent component) { ... }
  protected override void OnPaintAdornments(PaintEventArgs e) { ... }
  ...
}
```

Initialize is overridden to deploy initialization logic that's executed as the control is being sited. It's also a good location to cache a reference to the control being designed:

```
public class ClockControlDesigner : ControlDesigner {
  ClockControl clockControl = null;
  public override void Initialize(IComponent component) {
    base.Initialize(component);

    // Get clock control shortcut reference
    clockControl = (ClockControl)component;
  }
  ...
}
```

You could manually register with Control.OnPaint to add your design-time UI, but you'll find that overriding OnPaintAdornments is a better option because it is called only after the control's design-time or run-time UI is painted, letting you put the icing on the cake:

```
public class ClockControlDesigner : ControlDesigner {
  ...
  protected override void OnPaintAdornments(PaintEventArgs e) {
    // Let the base class have a crack
    base.OnPaintAdornments(e);

    // Don't show border if it does not have an Analog face
    if( clockControl.Face == ClockFace.Digital ) return;

    // Draw border
    Graphics g = e.Graphics;
    using( Pen pen = new Pen(Color.Gray, 1) ) {
      pen.DashStyle = DashStyle.Dash;
      g.DrawRectangle(
        pen, 0, 0, clockControl.Width - 1, clockControl.Height - 1);
    }
  }
  ...
}
```

Adding DesignerAttribute to the ClockControl class completes the association:

```
[ DesignerAttribute(typeof(ClockControlDesigner)) ]
public class ClockControl : Control { ... }
```

Design-Time-Only Properties

The clock control is now working as shown in Figure 9.34. One way to improve on this is to make it an option to show the border, because it's a feature that not all developers will like. Adding a design-time-only ShowBorder property will do the trick, because this is not a feature that should be accessible at run time. Implementing a design-time-only property on the control itself is not ideal because the control operates in both design-time and run-time modes. Designers are exactly the right location for design-time properties.

To add a design-time-only property, start by adding the basic property implementation to the custom designer:

```
public class ClockControlDesigner : ControlDesigner {
  ...
  bool showBorder = true;
  ...
  protected override void OnPaintAdornments(PaintEventArgs e) {
    ...
    // Don't show border if hidden or
    // does not have an Analog face
    if( (!showBorder) ||
        (clockControl.Face == ClockFace.Digital) ) return;
    ...
  }

  // Provide implementation of ShowBorder to provide
  // storage for created ShowBorder property
  bool ShowBorder {
    get { return showBorder; }
    set {
      showBorder = value;
      clockControl.Refresh();
    }
  }
}
```

This isn't enough on its own, however, because the Property Browser won't examine a custom designer for properties when the associated component is selected. The Property Browser gets its list of properties from TypeDescriptor's GetProperties method (which, in turn, gets the list of properties using .NET reflection). To augment the properties returned by the TypeDescriptor class, a custom designer can override the PreFilterProperties method:

```
public class ClockControlDesigner : ControlDesigner {
  ...
  protected override void PreFilterProperties(
    IDictionary properties) {

    // Let the base have a chance
    base.PreFilterProperties(properties);

    // Create design-time-only property entry and add it to
    // the Property Browser's Design category
    properties["ShowBorder"] = TypeDescriptor.CreateProperty(
      typeof(ClockControlDesigner),
      "ShowBorder",
      typeof(bool),
      CategoryAttribute.Design,
      DesignOnlyAttribute.Yes);
  }
  ...
}
```

The properties argument to PreFilterProperties allows you to populate new properties by creating PropertyDescriptor objects using the TypeDescriptor's CreateProperty method, passing the appropriate arguments to describe the new property. One of the parameters to TypeDescriptor.CreateProperty is DesignOnlyAttribute.Yes, which specifies design-time-only usage. It also physically causes the value of ShowBorder to be persisted to the form's resource file rather than to InitializeComponent, as shown in Figure 9.35.

If you need to alter or remove existing properties, you can override PostFilterProperties and act on the list of properties after TypeDescriptor has filled it using reflection. Pre/Post filter pairs can also be overridden for methods and events if necessary. Figure 9.36 shows the result of adding the ShowBorder design-time property.

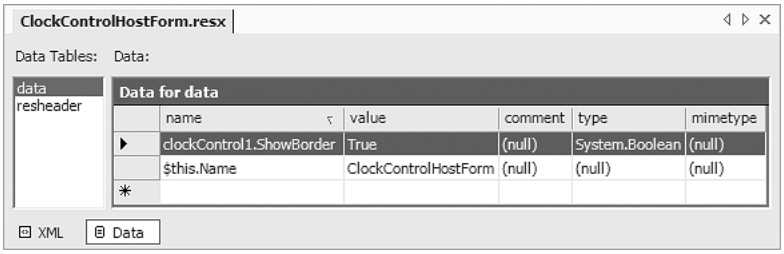

FIGURE 9.35: ShowBorder Property Value Serialized to the Host Form's Resource File

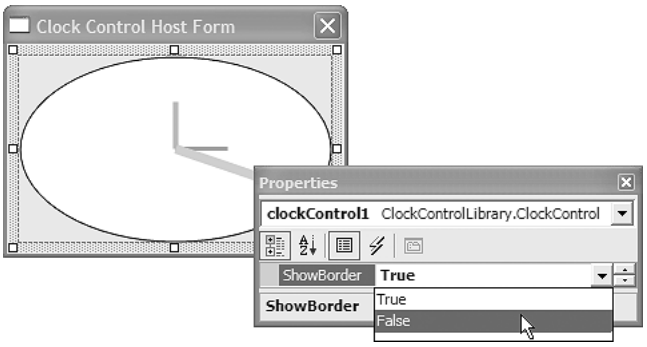

FIGURE 9.36: ShowBorder Option in the Property Browser

Design-Time Context Menu Verbs

To take the design-time-only property even further, it's possible to add items to a component's design-time context menu. These items are called *verbs*, and ShowBorder would make a fine addition to our clock control's verb menu.

Adding to the verb menu requires that we further augment the custom designer class:

```
public class ClockControlDesigner : ControlDesigner {
  ...
  public override DesignerVerbCollection Verbs {
    get {
      // Return new list of context menu items
      DesignerVerbCollection verbs = new DesignerVerbCollection();
      showBorderVerb =
        new DesignerVerb(
```

```
            GetVerbText(),
            new EventHandler(ShowBorderClicked));
        verbs.Add(showBorderVerb);
        return verbs;
      }
    }
    ...
}
```

The Verbs override is queried by the Designer shell for a list of DesignerVerbs to insert into the component's context menu. Each DesignerVerb in the DesignerVerbCollection takes a string name value plus the event handler that responds to verb selection. In our case, this is ShowBorderClicked:

```
public class ClockControlDesigner : ControlDesigner {
  ...
  void ShowBorderClicked(object sender, EventArgs e) {
    // Toggle property value
    ShowBorder = !ShowBorder;
  }
  ...
}
```

This handler simply toggles the ShowBorder property. However, because the verb menu for each component is cached, it takes extra code to show the current state of the ShowBorder property in the verb menu:

```
public class ClockControlDesigner : ControlDesigner {
  ...
  bool ShowBorder {
    get { return showBorder; }

    set {
      // Change property value
      PropertyDescriptor property =
        TypeDescriptor.GetProperties(typeof(ClockControl))["ShowBorder"];
      this.RaiseComponentChanging(property);
      showBorder = value;
      this.RaiseComponentChanged(property, !showBorder, showBorder);

      // Toggle Show/Hide Border verb entry in context menu
      IMenuCommandService   menuService =
        (IMenuCommandService)this.GetService
             (typeof(IMenuCommandService));
      if( menuService != null ) {
```

continues

```
        // Re-create Show/Hide Border verb
        if( menuService.Verbs.IndexOf(showBorderVerb) >= 0 ) {
          menuService.Verbs.Remove(showBorderVerb);
          showBorderVerb =
            new DesignerVerb(
              GetVerbText(),
              new EventHandler(ShowBorderClicked));
          menuService.Verbs.Add(showBorderVerb);
        }
      }

      // Update clock UI
      clockControl.Invalidate ();
    }
  }
  ...
}
```

ShowBorder now performs two distinct operations. First, the property value is updated between calls to RaiseComponentChanging and RaiseComponentChanged, helper functions that wrap calls to the designer host's IComponentChangeService. The second part of ShowBorder re-creates the Show/Hide Border verb to reflect the new property value. This manual intervention is required because the Verbs property is called only when a component is selected on the form. In our case, "Show/Hide Border" could be toggled any number of times after the control has been selected.

Fortunately, after the Verbs property has delivered its DesignerVerbCollection payload to the Designer, it's possible to update it via the designer host's IMenuCommandService. Unfortunately, because the Text property is read-only, you can't implement a simple property change. Instead, the verb must be re-created and re-associated with ShowBorderClicked every time the ShowBorder property is updated.

On top of adding Show/Hide Border to the context menu, .NET throws in a clickable link for each verb, located on the Property Browser above the property description bar. Figure 9.37 illustrates all three options, including the original editable property.

Custom designers allow you to augment an application developer's design-time experience even further than simply adding the effects to the Property Browser. Developers can change how a control renders itself, controlling the properties, methods, and events that are available at design time and augmenting a component's verbs.

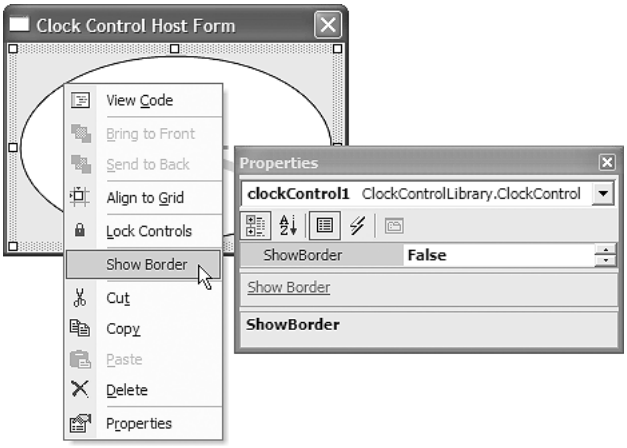

FIGURE 9.37: ShowBorder Option in the Property Browser and the Context Menu

Where Are We?

Although components (and, by association, controls) gain all kinds of integration into a .NET design-time environment with very little work, .NET also provides a rich infrastructure to augment the design-time experience for your custom components.