ГЛАВА 28

Поддержка времени проектирования

К пользовательским элементам управления предъявляются два требования. Они должны взаимодействовать с веб-страницей (и вашим кодом) во время выполнения, и они должны взаимодействовать с Visual Studio во время проектирования. Эти две задачи взаимосвязаны, но они могут быть уточнены и настроены по отдельности. Некоторые из наиболее усовершенствованных элементов управления ASP.NET (вроде `Calendar` и `DataList`) предлагают впечатляющий набор удобств на этапе проектирования, включая возможность конфигурирования сложных свойств и применения тем одним щелчком мыши.

Возможно, вы уже знаете о том, что Visual Studio автоматически предоставляет пользовательским элементам управления высокую степень поддержки времени проектирования. Например, когда вы компилируете проект пользовательских элементов управления, ваши элементы добавляются в панель инструментов (Toolbox) автоматически. Вы можете затем перетаскивать их в свою веб-форму, конфигурировать их свойства, прикреплять обработчики событий и т.д. В зависимости от того, как реализована логика генерации, можно даже видеть представление времени проектирования HTML-кода этого элемента. В настоящей главе вы узнаете, как расширить эту поддержку времени проектирования.

Многие из приемов, с которыми вы ознакомитесь, представляют собой своего рода "украшательства", которые облегчают работу с пользовательскими элементами управления. Например, вы можете использовать поддержку времени выполнения для добавления описаний, которые появляются в окне Properties (Свойства) или выбора более представительного внешнего вида для вашего элемента в панели проектирования. Однако иногда поддержка времени проектирования просто необходима. Например, если вы создаете элемент управления, который представляет сложные объекты как свойства, и вы не предпринимаете никаких дополнительных шагов для добавления поддержки времени выполнения, то такой элемент будет вести себя неустойчиво в среде времени проектирования. Вы можете обнаружить, что свойства, которые вы устанавливаете в окне Properties, иногда сбрасываются или приводят к исчезновению дескрипторов вложенных дочерних элементов управления. Эти причуды являются следствием того, как ASP.NET сериализует свойства вашего элемента управления, и в этой главе вы узнаете, как бороться с подобными проявлениями.

Ключевые игроки

В .NET нет одного класса, обеспечивающего поддержку времени проектирования. Вместо того есть множество разных ингредиентов, участвующих в этом процессе. Часть из них перечислено ниже.

- *Атрибуты.* Вы применяете атрибуты к частям вашего элемента управления по нескольким причинам. Во-первых, эти атрибуты представляют информацию, которая будет использована для настройки отображения в окне Properties и пиктограммы в панели инструментов. Во-вторых, атрибуты конфигурируют сериализацию свойств. В-третьих, атрибуты позволяют прикреплять к вашему элементу другие компоненты времени проектирования, такие как конвертеры типов, редакторы типов и дизайнеры элементов.
- *Веб-ресурсы.* Веб-ресурсы касаются файлов, используемых пользовательским элементом управления, таких как графические изображения. Используя веб-ресурсы, вы можете встраивать эти файлы в сборку элемента управления и использовать специально форматированные URL для извлечения их при необходимости.
- *Конвертеры типов.* Конвертеры типов позволяют сложным и необычным типам данных преобразовываться в представление более распространенных типов и обратно. Например, если вы создаете редактор типов, позволяющий конвертировать пользовательский тип данных в строковое представлении и обратно, вы можете затем просматривать и редактировать свойство элемента управления, использующее этот тип данных, в окне Properties. Конвертеры типов могут также играть роль в коде сериализации, генерируя инициализирующий код, необходимый для воссоздания экземпляра сложного типа.
- *Редакторы типов.* Редакторы типов представляют графический интерфейс для установки значений сложных типов. Например, когда вы выбираете шрифт для веб-элемента управления, используя для этого раскрывающийся список наименований шрифтов в окне Properties, таким образом, активизируя редактор типа.
- *Дизайнеры элементов управления.* Дизайнеры элементов — "тяжеловесы" в мире разработки пользовательских элементов управления. Каждый элемент имеет свой дизайнер, управляющий его внешним видом и поведением во время проектирования. Вы можете использовать дизайнер пользовательского элемента управления для добавления таких прелестей, как "интеллектуальные" дескрипторы (smart tags — "смарт-тэги"), и представлять HTML-содержимое по умолчанию, которое Visual Studio должен применять для представления элемента на поверхности проектирования. Вы можете также использовать дизайнер для сокрытия свойств в вашем классе элемента во время проектирования, либо для добавления свойств, существующих только во время проектирования.

В настоящей главе мы пересмотрим несколько пользовательских элементов управления, которые были представлены в главе 27, и рассмотрим, как можно снарядить элемент управления специальной пиктограммой для панели инструментов и поддержкой свойств для окна Properties. Затем вы узнаете о том, как сформировать базовую сериализацию элемента с атрибутами и конвертерами типов. Кроме того, вы увидите, как усовершенствовать поддержку времени проектирования редакторами типов, облегчающими установку значений сложных свойств. И, наконец, вы увидите, как управлять дизайнером при выполнении трех распространенных задач: дальнейшей сериализации пользовательского элемента управления, изменению его HTML-представления по умолчанию, и добавления "интеллектуального" дескриптора с согласованными сокращениями для распространенных задач конфигурирования элемента.

Атрибуты времени проектирования

Первый уровень поддержки времени проектирования состоит из управляющих *атрибутов* — декларативных флагов, которые компилируются в метаданные сборки вашего пользовательского элемента управления. Атрибуты предоставляют вам возможность добавлять информацию, связанную с частью кода, без необходимости изменения этого кода или создания отдельного файла совершенно другого формата.

Атрибуты всегда помещаются в квадратных скобках перед элементом кода, который они модифицируют. Например, вот как можно добавить атрибут, который представляет описание свойства `Text` элемента управления:

```
[Description("Текст, который должен быть показан в элементе управления")]
public string Text
{ ... }
```

В данном случае атрибут `Description` *декорирует* свойство `Text`.

На заметку! Все атрибуты на самом деле являются классами. По соглашению имена этих классов оканчиваются на `Attribute`. Например, атрибут `Description` на самом деле представлен классом `DescriptionAttribute`. Хотя вы можете использовать полное имя этого класса, компилятор C# позволяет использовать удобное сокращение, пропуская завершающее слово `Attribute`.

В .NET атрибуты применяются для выполнения широкого диапазона задач. Ключевая деталь к пониманию атрибутов заключается в том, что они могут быть прочитаны и интерпретированы различными агентами. Например, вы можете добавлять атрибуты, которые представляют информацию CLR, компилятору или специальному инструменту. Эта глава в основном сосредоточена на атрибутах, представляющих информацию среде Visual Studio, сообщая ей о том, как работать с данным элементом управления во время проектирования. Позднее, в разделе "Сериализация кода" вы узнаете также о некоторых атрибутах, влияющих на то, как анализатор ASP.NET интерпретирует дескрипторы элементов управления в файле `.aspx`.

Совет. Как многие из классов, предназначенных для поддержки времени выполнения, большинство атрибутов, о которых вы узнаете из этой главы, находятся в пространстве имен `System.ComponentModel`. Прежде чем применить эти атрибуты, вы должны импортировать это пространство имен в файлы кода своих пользовательских элементов управления.

Окно Properties

Простейшее влияние атрибутов состоит в том, в каком виде ваш элемент управления появляется в окне Properties (Свойства). Например, вероятно, вы уже заметили, что базовое множество веб-элементов управления ASP.NET группируют свои свойства в нескольких категориях. При выборе свойства в окне Properties отображается краткое описание. Чтобы добавить эту информацию к вашему элементу управления, вы должны декорировать каждое из его свойств атрибутами `Category` и `Description`, как показано ниже:

```
[Category("Appearance")]
[Description("The text to be shown in the control")]
public string Text
{
  get {return (string)ViewState["Text"];
  set {ViewState["Text"] = value;}
}
```

Как видите, оба атрибута — `Category` и `Description` — принимают в качестве аргумента одну строку. На рис. 28.1 демонстрируется, что будет отображено при выборе свойства `Text` в окне Properties (Свойства).

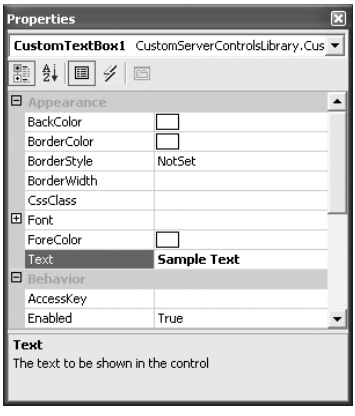

Рис. 28.1. Свойство с описанием

В табл. 28.1 перечислены ключевые атрибуты, которые влияют на способ отображения свойства в окне Properties.

Таблица 28.1. Атрибуты свойств элемента управления

Атрибут	Описание
`Browsable(true|false)`	Если равно `false`, то данное свойство не появится в окне Properties (хотя программист все равно сможет модифицировать код или вручную добавлять атрибут управляющего дескриптора, если вы предусмотрите для него процедуру `Set`). Одной из причин применения такого атрибута может быть сокрытие вычисляемых свойств или свойств времени выполнения, которые не могут изменяться во время проектирования.
`Category(string)`	Строка, указывающая категорию, под которой свойство появится в окне Properties. Если вы не специфицируете значение атрибута `Category`, свойство появится в окне `Properties` в категории `Default`.
`Description(string)`	Строка, указывающая описание, которое будет иметь свойство при выборе его в окне Properties.
`DefaultValue()`	Устанавливает значение по умолчанию, которое будет отображено для свойства в окне Properties. Значение по умолчанию — обычно начальное значение, в этом случае не нужно применять атрибут `DefaultValue`. Однако использование этого атрибута может иногда позволить генератору кода оптимизировать генерируемые им дескрипторы, пропуская информацию, если она соответствует установкам по умолчанию.
`Themeable(string)`	Все пользовательские элементы управления автоматически поддерживают темы (см. главу 16). Однако если вы не хотите, чтобы определенное свойство было сконфигурировано как часть темы, примените атрибут `Themeable` со значением `false`. В отличие от других атрибутов в этой таблице, атрибут `Themeable` определен в пространстве имен `System.Web.UI`.

Окончание табл. 28.1

Атрибут	Описание
`Localizable(true|false)`	Локализация включена для всех элементов управления и объектов. Если свойство локализовано, то Visual Studio позволит сохранять его значения в сопутствующей сборке. Если вам не нужна такая возможность или ваше свойство не зависит от текущих локальных установок, установите этот атрибут в `false`.
`ReadOnly(bool)`	При установке в `true`, это свойство во время проектирования доступно в окне Properties только для чтения.
`DesignOnly(bool)`	При установке в `true` это свойство доступно только во время проектирования. Обычно это используется со специальными свойствами, конфигурирующими поведение элемента управления во время проектирования и не соответствующими никакой "реальной" части информации об элементе управления.
`ImmutableObjext(bool)`	При установке в `true` для объектного свойства этот атрибут гарантирует, что все подсвойства этого объекта будут отображены с доступом только для чтения. Например, если вы примените это к свойству, использующему объект `Point`, то подсвойства `X` и `Y` будут доступны только для чтения.
`MergableProperty(bool)`	Конфигурирует поведение окна Properties, когда одновременно выбрано более одного экземпляра этого элемента управления. Если равно `false`, то свойство не отображается. Если равно `true` (по умолчанию), свойство может быть установлено для всех выбранных элементов управления сразу.
`ParenthesizePropertyName(bool)`	Если равно `true`, то Visual Studio отобразит скобки вокруг этого свойства в окне Properties (как это делается со свойством `ID`).
`Bindable(bool)`	Если равно `true`, Visual Studio отобразит это свойство в диалоговом окне DataBindings (Привязки данных) и позволит привязывать его к полю источника данных.
`RefreshProperties()`	Вы используете этот атрибут со значением из перечисления `RefreshProperties`. Оно специфицирует, должна ли остальная часть окна Properties обновляться при изменении этого свойства (например, если одна процедура свойства может изменить другое свойство).

Вы можете применить два атрибута, `DefaultEvent` и `DefaultProperty`, к объявлению вашего класса пользовательского элемента управления вместо специфического свойства. Вдобавок, атрибут `TagPrefix` используется на уровне сборки, и не присоединяется ни к какой конструкции кода. Эти атрибуты описаны в табл. 28.2.

Как известно из главы 27, каждый пользовательский элемент управления имеет префикс, который регистрируется директивой `Register` на странице `.aspx`. Visual Studio добавляет эту директиву автоматически, когда вы вставляете элемент управления. Если вы хотите настроить префикс, то можете использовать атрибут `TagPrefix`, который принимает два строковых параметра. Первая строка указывает пространство имен, в котором находится ваш элемент управления, а вторая — префикс дескриптора, который нужно использовать.

Рассмотрим пример, специфицирующий, что элементы управления в `CustomServer ControlsLibrary` должны использовать префикс дескриптора `apress`:

```
[assembly: System.Web.UI.TagPrefix("CustomServerControlsLibrary", "apress")]
```

Таблица 28.2. Атрибуты классов элемента управления и сборок

Атрибут	Описание
`DefaultEvent(string)`	Указывает имя события по умолчанию. Когда выполняется двойной щелчок на элементе управления в среде проектирования, Visual Studio автоматически добавляет обработчик события по умолчанию. (Если обработчика по умолчанию нет, двойной щелчок на элементе просто выбирает его.)
`DefaultProperty(string)`	Указывает имя свойства по умолчанию. `DefauleProperty` — это свойство, которое подсвечивается в окне Properties по умолчанию, когда элемент управления выбирается впервые. (Если свойства по умолчанию нет, никакое свойство не выбирается при первоначальном выборе элемента.)
`ControlValueProperty-Attribute(string)`	Указывает имя свойства, которое должно быть использовано по умолчанию во время привязки `ControlParameter` к этому элементу для использования в источнике данных (как описано в главе 9).
`NonVisualControl()`	Указывает, что данных элемент не имеет внешнего представления во время выполнения. Во время проектирования вы можете выбрать в меню команду View⇨Visual Aids⇨ ASP.NET Non-Visual Controls (Вид⇨Визуальные вспомогательные средства⇨Невизуальные элементы управления ASP.NET) для сокрытия всех элементов, имеющих атрибут `NonVisualControl`. Элементы управления — источники данных являются примером невизуальных элементов.
`TagPrefix(string, string)`	Ассоциирует пространство имен с префиксом, который будет использован при добавлении дескрипторов элементов управления к странице `.aspx`.
`ToolboxBitmap(type, string)`	Специфицирует битовую карту, которая будет показана для этого элемента управления, когда он добавляется в панель инструментов. По умолчанию Visual Studio использует пиктограмму в виде шестеренки.
`ToolboxData(string)`	Специфицирует дескриптор, который будет создан для данного элемента в файле `.aspx`, когда вы перетаскиваете его из панели инструментов. По умолчанию Visual Studio создает пустой дескриптор, включая только атрибуты `ID` и `runat`.

Атрибут сборки не должен помещаться внутри блока пространства имен. Вместо этого он должен определяться вне всех пространств имен, т.е. в глобальном контексте.

Теперь, если вы добавите элемент управления с именем класса `CustomTextBox` из пространства имен `CustomServerControlsLibrary`, то Visual Studio использует такой дескриптор:

```
<apress:CustomTextBox ... />
```

Если у вас есть элементы управления во многих пространствах имен, то вы должны использовать `TagPrefix` много раз — по одному для каждого пространства имен. Вы можете применять один и тот же префикс или же разные. Часто атрибут `TagPrefix` помещается в файл `AssemblyInfo.cs`.

По умолчанию, когда вы перетаскиваете пользовательский элемент из панели инструментов, Visual Studio создает дескриптор этого элемента с атрибутами `runat` и `ID`, как показано ниже:

```
<apress:CustomTextBox ID="CustomTextBox1" runat="server" />
```

Используя атрибут ToolboxData, вы можете применять различную разметку, которая может включать значения по умолчанию. Вот пример:

```
[ToolboxData(
  "<{0}:CustomTextBox ID=CustomTextBox1 Text=\"[Empty]\" runat=server />")]
```

Обратите внимание, что для представления префикса дескриптора используется указатель места заполнения {0}.

Когда вы специфицируете значения атрибутов с атрибутом ToolboxData, вы можете часто пропускать знаки кавычек (как с атрибутами ID и runat в данном примере). Однако кавычки необходимы, когда приходится иметь дело со строками, содержащими пробелы или специальные символы (как с атрибутом Text). Чтобы включить кавычки в строку ToolboxData, можно использовать управляющую последовательность \".

Теперь, когда вы перетащите CustomTextBox на форму, то получите следующий дескриптор:

```
<apress:CustomTextBox ID="CustomTextBox1" Text="[Empty]" runat="server" />
```

Памятуя обо всех этих деталях, вы готовы обеспечить ожидаемый уровень поддержки времени проектирования обычному пользовательскому элементу управления. Рассмотрим простой пример элемента управления CustomTextBox из предыдущей главы с полным комплектом атрибутов. Код опущен.

```
[assembly: System.Web.UI.TagPrefix("CustomServerControlsLibrary", "apress")]
namespace CustomServerControlsLibrary
{
  [DefaultProperty("Text")]
  [DefaultEvent("TextChanged")]
  [ToolboxData(
  "<{0}:CustomTextBox ID=\"CustomTextBox1\" Text=\"[Empty]\" runat=\"server\" />")]
  public class CustomTextBox : WebControl, IPostBackDataHandler
  {
     public CustomTextBox() : base(HtmlTextWriterTag.Input)
     { ... }
     [Category("Appearance")]
     [Description("The text to be shown in the control")]
     [DefaultValue("")]
     [MergableProperty(true)]
     public string Text
     { ... }
     protected override void AddAttributesToRender(HtmlTextWriter output)
     { ... }
     public bool LoadPostData(string postDataKey,
     NameValueCollection postData)
     { ... }
     public void RaisePostDataChangedEvent()
     { ... }
     public event EventHandler TextChanged;
     protected virtual void OnTextChanged(EventArgs e)
     { ... }
  }
}
```

Атрибуты и наследование

Когда вы наследуете элемент управления от базового класса, который имеет атрибуты времени проектирования, элемент управления наследует функциональность времени проектирования своего родителя — точно так же, как он наследует методы и свойст-

ва. Если реализация родительского класса атрибута времени выполнения существенна для вашего элемента управления, вы не обязаны повторно применять их.

Однако в некоторых случаях вы можете изменить поведение времени выполнения существующего свойства. В этом случае вы должны, прежде всего, переопределить свойство и затем повторно применить измененные атрибута либо добавить новые.

Большинство свойств базовых классов `WebControl` и `Control` помечены как виртуальные, что позволяет вам изменять их поведение. Например, если вы хотите сокрыть свойство `Height` пользовательского элемента управления, который унаследован от `WebControl` (возможно, потому, что оно вычисляется по содержимому, а не устанавливается разработчиком), то вы можете переопределить свойство `Height` и применить атрибут `Browseable`, как показано ниже:

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

Пиктограмма панели инструментов

Добавить пиктограмму в панель инструментов (Toolbox) на удивление просто. Все, что нужно для этого сделать — добавить битовую карту (bitmap) к проекту и выполнить следующие правила.

- Файл битовой карты должен иметь то же имя, что и класс вашего пользовательского элемента управления (но с расширением `.bmp`). Например, вы можете использовать карту по имени `CustomTextBox.bmp` для элемента `CustomTextBox`.
- Карта должна иметь размер 16×16 пикселей. Иначе она будет искажена, когда Visual Studio попытается масштабировать ее.
- Она должна использовать только 16 цветов.
- Как только вы добавите файл, воспользуйтесь окном Properties (Свойства) для установки свойства Build Action (Действие сборки) для нее в Embedded Resource (Встроенный ресурс).
- На рис. 28.2 показан требуемая пиктограмма для элемента управления `CustomTextBox`.
- Как только вы добавите правильную битовую карту и установите ее свойство Build Action в Embedded Resource, следующим шагом должна быть компиляция проекта. Однако вы не увидите ваши пользовательские пиктограммы во вкладке Toolbox, которую Visual Studio автоматически добавит к вашему проекту. Вы увидите вашу пиктограмму, только если вручную сконфигурируете Toolbox и добавите свой элемент управления к одной из ее постоянных вкладок. Чтобы сделать это, выполните щелчок правой кнопкой мыши на соответствующей вкладке, выберите команду Choose Items (Выберите элементы) в контекстном меню и затем перейдите к сборке, содержащей вашу библиотеку элементов управления (как описано в главе 27).

На рис. 28.3 представлен пример с двумя пользовательскими элементами управления. Один имеет обобщенную пиктограмму, изображающую шестерню, в то время как `CustomTextBox` использует специальную битовую карту, добавленную в проект этого элемента управления.

Между прочим, можно использовать пиктограмму панели инструментов, которая находится в файле, чье имя не совпадает с именем класса вашего элемента управления. В этом случае вам понадобится помощь атрибута `ToolboxBitmap`.

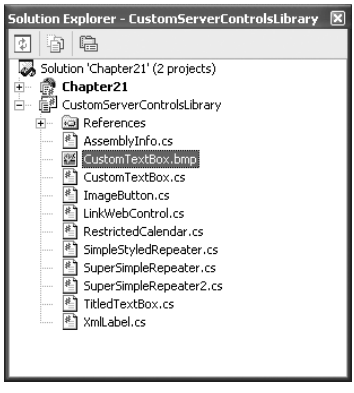

Рис. 28.2. Добавление битовой карты в панель инструментов

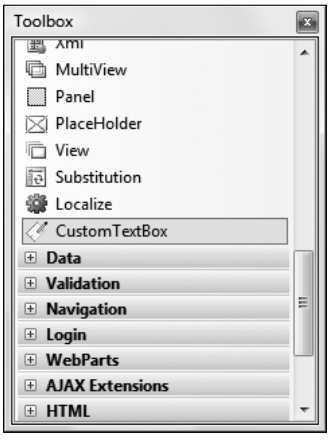

Рис. 28.3. Пользовательская битовая карта в панели инструментов

Например, следующий код конфигурирует элемент управления `CustomTextBox` для использования битовой карты по имени `CustomTextBox1.bmp`:

```
[ToolboxBitmap(typeof(CustomTextBox), "CustomTextBox1.bmp")]
public class CustomTextBox : WebControl, IPostBackDataHandler
{ ... }
```

Вы можете также использовать этот трюк для размещения битовых карт в отдельной вложенной папке вашего проекта. Например, вот как можно сослаться на битовую карту в папке `Images`:

```
[ToolboxBitmap(typeof(CustomTextBox), @"Images\CustomTextBox1.bmp")]
```

И, наконец, также можно позаимствовать битовые карты у основных элементов управления ASP.NET, применив код следующего типа:

```
[ToolboxBitmap(typeof(System.Web.UI.WebControls.TextBox))]
```

Если вы создаете простой элемент управления, все, что вам может понадобиться сделать — это добавить набор описательных свойств и пиктограмму для панели инструментов. Однако более сложные элементы часто требуют дополнительных компонентов. Они распространяются от характеристик сериализации (как создается дескриптор элемента управления, когда вы используете окно Properties) до визуальных конструкторов элемента управления (расширенных инструментов для настройки HTML времени проектирования, который генерирует ваш элемент управления). В остальной части настоящей главы будут рассмотрены эти темы.

Веб-ресурсы

Часто пользовательские элементы управления будут иметь другие ассоциированные с ними ресурсы, не связанные с кодом. Например, сюда могут входить файлы сценариев, таблицы стилей и графические образы, которые нужно использовать вместе с элементом управления. Это является дополнительной головной болью при развертывании, поскольку вы должны при этом копировать все эти ресурсные файлы в каждое веб-приложение, использующее ваш элемент управления. К счастью, ASP.NET предлагает новое решение в виде новой модели веб-ресурсов.

Создание ресурса

Чтобы создать веб-ресурс, начните с добавления файла ресурсов в проект пользовательского элемента управления. Затем в окне Properties (Свойства) измените значение свойства Build Action (Действие сборки) на Embedded Resource (Встроенный ресурс) вместо Content (Содержимое), как показано на рис. 28.4. Таким образом, файл будет встроен внутрь вашей компилированной сборки.

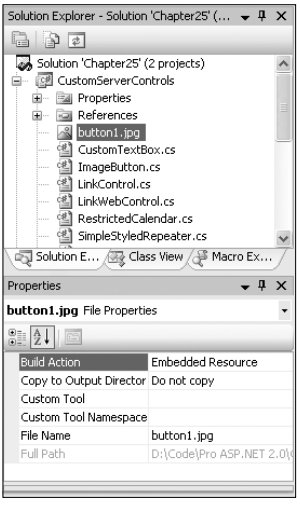

Рис. 28.4. Встроенный ресурс

Следующий шаг — сделать ресурс доступным через URL. Это работает за счет применения нового атрибута уровня сборки `WebResource`.

Например, предположим, что вы хотите обратиться к изображению кнопки для элемента управления `CustomImageButton`. Как только вы измените установку Build Action, вам понадобится добавить следующий атрибут к своему коду:

```
[assembly: WebResource(
  "CustomServerControls.button1.jpg", "image/jpeg")]
```

Атрибут `WebResource` принимает два параметра. Первый — полное имя встраиваемого ресурса, которое *должно* быть снабжено префиксом — названием пространства имен по умолчанию вашего проекта. (Эта деталь, как можно убедиться, щелкнув правой кнопкой мыши на вашем проекте в проводнике Solution Explorer и выбрав в контекстном меню команду Properties (Свойства), автоматически добавляется в начало имени ресурса.) Второй параметр — тип содержимого.

Выполнив эти шаги, последнее, что остается сделать — получить URL вашего встроенного ресурса. ASP.NET поддерживает его через новый обработчик `WebResource.axd`. Этот обработчик принимает запросы URL, извлекает соответствующий ресурс из соответствующей сборки и возвращает его содержимое. Другими словами, вам не нужно возиться с картинками и сценариями, потому что файл `WebResource.axd` может обслужить их, как нужно — прямо из сборки вашего пользовательского элемента управления.

Извлечение ресурса

Чтобы получит ресурс, вам нужно сообщить обработчику `WebResource.axd`, какой именно ресурс вам нужен, и какая сборка его содержит. Вы можете сгенерировать правильный URL, используя метод `Page.ClientScript.GetWebResourceUrl()`. Ниже приведен пример использования этой техники для захвата стандартной пиктограммы, связанной с элементом управления `CustomImageButton`:

```
protected override void OnInit(EventArgs e)
{
   ImageUrl = Page.ClientScript.GetWebResourceUrl(typeof(CustomImageButton),
      "CustomServerControls.button1.jpg");
}

public string ImageUrl
{
   get {return (string)ViewState["ImageUrl"];}
   set {ViewState["ImageUrl"] = value;}
}
```

На заметку! Кстати, веб-ресурсы автоматически принимают во внимание локализацию. Если у вас есть сопровождающая сборка ресурса со специфической локальной версией файла изображения, то именно эта версия и будет использована. Подробнее об этом читайте в главе 34.

Действительный URL выглядит примерно так:

```
WebResource.axd?a=CustomServerControls&r=button.jpg&t=632059604175183419
```

Строковые параметры запроса `a` и `r` специфицируют имена сборок и ресурсов соответственно. Параметр `t` — обходной путь для поддержки кэширования. По сути, `WebResource.axd` кэширует каждый запрошенный ресурс. Вот почему вы можете запрашивать одно и то же изображение сотни раз, не вызывая при этом дополнительной работы по извлечению ресурса из сборки. (Это особенно важно для обеспечения должного уровня производительности, когда используется сборка, включающая в себя огромное количество ресурсов.) Однако кэширование влечет за собой собственную проблему, а именно: вы не захотите повторно использовать кэшированный ресурс, если сборка содержит его более новую версию. Параметр `t` защищает от этого. Это — временная метка сборки. Когда заново строится сборка пользовательского элемента управления, сгенерированные URL будут иметь другое значение `t`. В результате браузер выполнит новый запрос, и файл `WebResource.axd` получит самое свежее содержимое.

Вы можете попробовать этот пример с элементом управления `CustomImageButton` в загружаемом коде для этой главы.

Подстановка текста

Обработчик `WebResource.axd` имеет в запасе еще один трюк. Вы можете установить булевский параметр `PerformSubstitution` в атрибуте `WebResource`, чтобы заставить его выполнять автоматическую подстановку. Это позволит вам создать встроенный ресурс, указывающий на *другие* встроенные ресурсы. Например, рассмотрим веб-элемент управления, использующий несколько HTML-файлов для предоставления вспомогательной информации. Вы можете применять гиперссылки для подключения одного HTML-файла к другому, но при этом встраивать их в виде веб-ресурсов.

Без автоматической подстановки вы не сможете создать ресурс, указывающий на другой ресурс, потому что не знаете имен ресурсов, которые сгенерирует ASP.NET. Но с автоматической подстановкой можно использовать оригинальное имя файла. Компилятор заменит вашу ссылку на имя файла правильным, автоматически сгенерированным именем ресурса. Например, предположим, что вы создаете файл по имени `CustomServerControls.Help.htm`, который содержит ссылки на другие ресурсы. Вот какой атрибут вам понадобится для файла `CustomServerControls.Help.htm`:

```
[assembly: WebResource("CustomServerControls.Help.htm", "text/html",
 PerformSubstituation=true)]
```

Теперь файл `WebResource.axd` будет сканировать текст вашего ресурсного файла (в данном случае — HTML-файла по имени `CustomServerControls.Help.htm`) и искать выражение в следующем формате:

```
<img src="WebResource(HelpTitle.gif) %>" alt="Help" />
```

И всякий раз, когда компилятор найдет ссылку вроде этой, он заменит ее URL-адресом соответствующего веб-ресурса (в данном случае — URL, указывающим на `HelpTitle.gif`). Вот пример:

```
<img src=
 "WebResource.axd?a=CustomServerControls&r=HelpTitle.gif&t=632059604175183419"
 alt="Help" />
```

По завершении трансформации модифицированная версия `CustomServerControls.Help.htm` будет встроена в сборку в виде ресурса.

На заметку! Автоматическая подстановка работает только с ресурсами, основанными на тексте.

Сериализация кода

Когда вы конфигурируете свойства элемента управления в окне Properties (Свойства), Visual Studio должна иметь возможность создавать и модифицировать соответствующий дескриптор элемента управления в файле `.aspx`. Этот процесс называется *сериализацией кода*, и часто работает автоматически. Однако вы можете столкнуться с проблемой, если используете свойства, которые сами являются сложными типами, либо создадите шаблонный элемент управления или же элемент управления, который поддерживает дочерние элементы.

В следующих разделах вы узнаете о различных ингредиентах, которые влияют на сериализацию элемента управления, и какие изменения следует внести для решения распространенных проблем.

Конвертеры типов

Окно Properties прозрачно работает с обычными типами данных. Строковые данные не представляют собой проблемы, но окно Properties также может преобразовывать строки в числовые типы. Например, если взглянуть на свойство `Width` (ширина) элемента управления, можно увидеть такое значение, как `50 px`. Вы можете вводить любые символы в это поле, но если вы попытаетесь зафиксировать изменения (нажав <Enter> или перейдя к другому полю), введя символы, которые не могут быть интерпретированы правильно, такое изменение будет отклонено.

Подобного рода поведение возможно благодаря *конвертерам типов* (type converters) — специализированным классам, которые предназначены для единственной цели — преобразовывать специализированные типы данных в строковое представление и обратно. Большая часть типов данных ядра .NET имеют конвертеры типов по умолчанию, которые работают исключительно хорошо. (Вы можете найти эти конвертеры типов в пространстве имен `System.ComponentModel`.) Однако если вы создадите собственную структуру классов и используете их в качестве свойств, то вам, вероятно, захочется создать и пользовательские конвертеры типов, которые позволят с ними работать в окне Properties.

Элемент управления с объектными свойствами

Следующий пример использует элемент управления `RichLabel`, который представляет собой слегка усовершенствованную версию элемента `XmlLabel`, показанного в главе 27. Разница в том, что в то время как `XmlLabel` предназначен только для отображения XML-документов, `RichLabel` предназначен для поддержки содержимого различного рода.

По сути, `RichLabel` может поддерживать содержимое любого типа, который определен в следующем перечислении `RichLabelTextType`. В этом простом примере перечисление `RichLabelTextType` включает только два выбора: `Xml` (который использует тот же код, что и `XmlLabel`) и `Html` (который трактуется как текст и не требует никакой дополнительной обработки). Однако вы можете легко добавить код генерации для разных типов текста.

```
public enum RichLabelTextType
{Xml, Html}
```

RichLabel также позволяет вам выбрать дескриптор, который вы хотите использовать для форматирования важных деталей (наподобие XML-дескрипторов в режиме генерации XML). Это работает через другой класс — RichLabelFormattingOptions. В этом классе определены два свойства: Type (которое содержит значение из перечисления RichLabelFormattingOptions) и HighlightTag (которое хранит имя дескриптора в виде строки — например, b для дескриптора <b>, включающего форматирование полужирным).

```
[Serializable()]
public class RichLabelFormattingOptions
{
 private RichLabelTextType type;
 public RichLabelTextType Type
 {
  get {return type;}
  set {type = value;}
 }
 private string highlightTag;
 public string HighlightTag
 {
  get {return highlightTag;}
  set {highlightTag = value;}
 }
 public RichLabelFormattingOptions(RichLabelTextType type,
 string highlightTag)
 {
  this.highlightTag = highlightTag;
  this.type = type;
 }
}
```

Класс RichLabel включает свойство Format, которое представляет экземпляр пользовательского класса RichLabelFormattingOptions. Логика генерации элемента управления RichLabel использует эту информацию для настройки генерируемого HTML.

Рассмотрим код элемента управления RichLabel:

```
[DefaultProperty("RichText")]
public class RichLabel : WebControl
{
 public RichLabel()
 {
  Text = "";
  // По умолчанию текст XML с дескрипторами, сформатированными полужирным.
  Format = new RichLabelFormattingOptions(RichLabelTextType.Xml, "b");
 }
 [Category("Appearance")]
 [Description("The content that will be displayed.")]
 public string Text
 {
  get {return (string)ViewState["Text"];}
  set {ViewState["Text"] = value;}
 }
 [Category("Appearance")]
 [Description("Options for configuring how text is rendered.")]
 public RichLabelFormattingOptions Format
 {
  get {return (RichLabelFormattingOptions)ViewState["Format"];}
  set {ViewState["Format"] = value;}
 }
```

```
protected override void RenderContents(HtmlTextWriter output)
{
 string convertedText = "";
 switch (Format.Type)
 {
   case RichLabelTextType.Xml:
      // Найти и выделить дескрипторы XML.
      convertedText = RichLabel.ConvertXmlTextToHtmlText(
      Text, Format.HighlightTag);
      break;
   case RichLabelTextType.Html:
      // Оставить текст в том виде, в каком он есть.
      convertedText = Text;
      break;
 }
 output.Write(convertedText);
}
public static string ConvertXmlTextToHtmlText(string inputText,
  string highlightTag)
{
 // (Код опущен.)
}
}
```

Возможен и альтернативный дизайн. Например, вы можете добавить эти два фрагмента информации (`Type` и `HighlightTag`) как отдельные свойства класса `RichLabel` — в этом случае вам не понадобится предпринимать никаких дополнительных шагов для обеспечения правильной сериализации. Однако вы можете решить сгруппировать взаимосвязанные свойства вместе, применяя пользовательский класс, и на то имеется множество причин. Возможно, вы хотите получить возможность повторно использовать класс `RichLabelFormattingOptions`, чтобы специфицировать опции форматирования текста для других элементов управления. Или же, может быть, вы захотите создать более сложный элемент управления, который будет принимать несколько разных кусков текста и конвертировать их, используя независимые установки `RichLabelFormattingOptions`. В обеих ситуациях будет удобно сгруппировать свойства, используя класс `RichLabelFormattingOptions`.

Однако элемент `RichLabel` работает не слишком хорошо с Visual Studio. Когда вы попытаетесь модифицировать этот элемент управления во время проектирования, то немедленно столкнетесь с проблемой. Окно Properties не позволит редактировать свойство `RichLabel.Format`. Вместо этого оно показывает пустое поле редактирования, в которое вы ничего не можете ввести. Чтобы решить эту проблему, вам потребуется создать пользовательский конвертер типа, как будет объяснено в следующем разделе.

Создание пользовательского конвертера типа

Пользовательский конвертер типа — это класс, который может преобразовывать ваш собственный типа данных (в данном случае — класс `RichLabelFormattingOptions`) в строку и обратно. В следующем примере мы рассмотрим такой класс `RichLabelFormatting OptionsConverter`.

Первый шаг состоит в том, чтобы создать пользовательский класс, унаследованный от базового класса `TypeConverter`, как показано ниже:

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

По соглашению имя класса-конвертера типа состоит из имени преобразуемого класса, за которым следует слово `Converter`.

Создав конвертер типа, вы должны переопределить несколько методов.

- **`CanConvertFrom()`**. Этот метод проверяет тип данных и возвращает `true`, если конвертер типа может выполнять преобразование этого типа данных в пользовательский тип.
- **`ConvertFrom()`**. Этот метод выполняет преобразование из переданного типа данных в пользовательский тип.
- **`ConvertTo()`**. Этот метод выполняет преобразование из пользовательского типа данных в указанный.

Напомним, что ключевая задача конвертера типа — выполнять преобразования между вашим пользовательским типом и его строковым представлением. В этом примере используется строковое представление, которое включает оба значения объекта `RichLabelFormattingOptions`, разделенные запятой и пробелом, с взятым в угловые скобки именем дескриптора. Вот как выглядит формат строки:

```
Type Name, <HighlightTag>
```

А вот пример форматирования XML с дескриптором <b>:

```
Xml, <b>
```

Имея это в виду, вы можете создать два вспомогательных метода в классе-конвертере для выполнения такого преобразования. Первый — метод `ToString()` — строит необходимое строковое представление:

```
private string ToString(object value)
{
   RichLabelFormattingOptions format = (RichLabelFormattingOptions)value;
   return String.Format("{0}, <{1}>", format.Type, format.HighlightTag);
}
```

Вторая часть — метод `FromString()` — декодирует строковое представление. Если строка не имеет нужного формата, код `FromString()` возбуждает исключение. В противном случае он возвращает новый экземпляр объекта.

```
private RichLabelFormattingOptions FromString(object value)
{
 string[] values = ((string)value).Split(',');
 if (values.Length != 2)
  throw new ArgumentException("Невозможно конвертировать значение");
 try
 {
  // Конвертирует имя перечислимого значения в соответствующее перечислимое
  // значение (которое на самом деле является целочисленной константой).
  RichLabelTextType type = (RichLabelTextType)Enum.Parse(
     typeof(RichLabelTextType), values[0], true);
  // Удалить пробелы и угловые скобки вокруг имени дескриптора.
  string tag = values[1].Trim(new char[]{' ','<','>'});
  return new RichLabelFormattingOptions(type, tag);
 }
 catch
 {
  throw new ArgumentException("Невозможно конвертировать значение");
 }
}
```

Перед попыткой преобразования из строки в объект `RichLabelFormattingOptions` окно Properties первым делом запрашивает метод `CanConvertFrom()`. Если он возвращает значение `true`, то вызывается метод `ConvertFrom()`. Все, что должен сделать метод `CanConvertFrom()` — проверить, что передано значение типа строки, как показано ниже:

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

Метод `ConvertFrom()` вызывает преобразование с помощью метода `FromString()`, показанного ранее:

```
public override object ConvertFrom(ITypeDescriptorContext context,
   CultureInfo culture, object value)
{
 if (value is string)
 {
  return FromString(value);
 }
 else
 {
  return base.ConvertFrom(context, culture, value);
 }
}
```

На заметку! Хорошая объектно-ориентированная практика предусматривает предоставление базовому классу возможности обработать сообщение, которое вы не собираетесь поддерживать. В данном случае любые запросы на преобразование нераспознанного типа передаются базовому классу.

Тот же процесс выполняется в обратном порядке, когда осуществляется преобразование объекта `RichLabelFormattingOptions` в строку. Во-первых, `Properties` вызывает метод `CanConvertTo()`. Если он возвращает `true`, то следующий шаг для окна Properties вызвать метода `ConvertTo()`. Вот необходимый код:

```
public override bool CanConvertTo(ITypeDescriptorContext context,
   Type destinationType)
{
 if (destinationType == typeof(string))
 {
  return true;
 }
 else
 {
  return base.CanConvertTo(context, destinationType);
 }
}
public override object ConvertTo(ITypeDescriptorContext context,
   CultureInfo culture, object value, Type destinationType)
{
 if (destinationType == typeof(string))
 {
  return ToString(value);
 }
 else
```

```
 {
  return base.ConvertTo(context, culture, value, destinationType);
 }
}
```

Теперь, когда вы имеет полнофункциональный конвертер типа, следующий шаг — прикрепить его к соответствующему свойству.

Прикрепление конвертера типа

Прикрепить конвертер типа можно двумя способами. Можно добавить атрибут `TypeConverter` к соответствующему классу (в данном случае к `RichLabelFormattingOptions`), как показано ниже:

```
[TypeConverter(typeof(RichLabelFormattingOptionsConverter))]
[Serializable()]
public class RichLabelFormattingOptions
{ ... }
```

Таким образом, всякий раз, когда экземпляр этого класса используется для элемента управления, Visual Studio знает, что нужно использовать ваш конвертер типа.

Альтернативно вы можете прикрепить конвертер типа непосредственно к свойству в вашем пользовательском элементе управления, как показано ниже:

```
public class RichLabel : WebControl
{
    [TypeConverter(typeof(RichLabelFormattingOptionsConverter))]
    public RichLabelFormattingOptions Format
    { ... }
    ...
}
```

Этот подход наиболее оправдан, когда вы используете обобщенный тип данных (такой как строка) и хотите настроить его поведение только для данного случая.

Теперь вы можете перекомпилировать код и попытаться использовать элемент управления `RichLabel` на простой веб-странице. Когда вы выберете `RichLabel`, то увидите текущее значение свойства `RichLabel.Format` в окне Properties, показанном на рис. 28.5, и сможете редактировать его вручную.

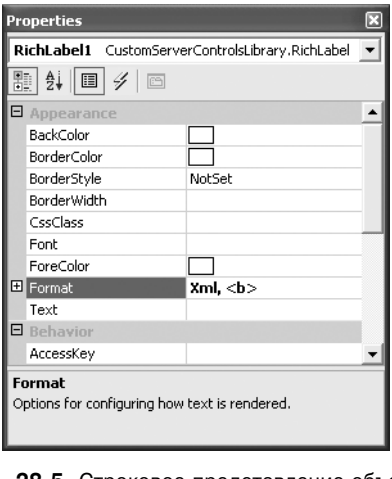

Рис. 28.5. Строковое представление объекта `RichLabelFormattingOptions`

На заметку! При изменении таких деталей, как конвертер типа, визуальный конструктор элемента управления и его построители, внесенные изменения не появляются немедленно в среде проектирования после перекомпиляции. Вместо этого может понадобиться закрыть решение с тестовой веб-страницей и затем открыть его повторно.

Конечно, если только вы не введете корректное строковое представление, то получите сообщение об ошибке, и внесенное изменение будет отклонено. Другими словами, показанный здесь пользовательский конвертер типа предоставляет вам возможность специфицировать объект `RichLabelFormattingOptions` в виде строки, но этот процесс, конечно, не слишком дружественен к пользователю. В следующем разделе показано, как можно повысить этот уровень поддержки.

`ExpandableObjectConverter`

Веб-элементы управления ASP.NET поддерживают множество объектных свойств. Лучшим примером является свойство `Font`, которое ссылается на объект `FontInfo`, обладающий такими свойствами, как `Bold`, `Italic`, `Name` и т.д. Когда вы устанавливаете свойство `Font` в окне Properties, вам не приходится набирать всю эту информацию в единственной корректно форматированной строке. Вместо этого вы можете развернуть свойство `Font`, щелкнув на значке плюс (+), и затем отредактировать свойства `FontInfo` индивидуально.

Вы можете включить такой же способ редактирования со своими собственными типами объектов — для этого потребуется просто создать пользовательский конвертер типа, унаследованный от класса `ExpandableObjectConverter`, вместо базового класса `TypeConverter`. Например, вы можете взять `RichLabelFormattingOptionsConverter`, разработанный в предыдущем разделе, и изменить его следующим образом:

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

После этого можно специфицировать свойство `Format`, либо вводя его в строке, либо развернув его и модифицировав одно из двух его подсвойств. На рис. 28.6 демонстрируется намного более удобный интерфейс, который вы увидите в окне Properties.

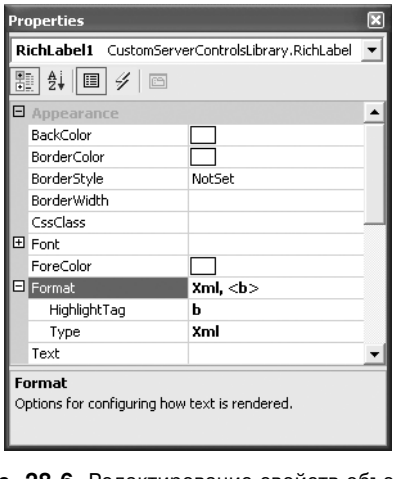

Рис. 28.6. Редактирование свойств объекта `RichLabelFormattingOptions`

На первый взгляд это выглядит хорошо, но все еще имеет несколько шероховатостей. Одна проблема состоит в том, что когда вы изменяете подсвойство (Type или HighlightTag), то строковое представление, показанное в поле Format, не обновляется немедленно. Чтобы решить эту проблему, вам нужно применить атрибуты NotifyParentProperty и RefreshProperties к свойствам класса RichLabelFormattingOptions. В то же время вы можете пожелать добавить атрибут Description для настройки текста, который будет появляться в окне Properties для этого подсвойства.

Ниже показан измененный код класса RichLabelFormattingOptions.

```
public class RichLabelFormattingOptions
{
  private RichLabelTextType type;
  [RefreshProperties(RefreshProperties.Repaint)]
  [NotifyParentProperty(true)]
  [Description("Type of content supplied in the text property")]
  public RichLabelTextType Type
  {
    get {return type;}
    set {type = value;}
  }
  private string highlightTag;
  [RefreshProperties(RefreshProperties.Repaint)]
  [NotifyParentProperty(true)]
  [Description("The HTML tag you want to use to mark up highlighted portions.")]
  public string HighlightTag
  {
    get {return highlightTag;}
    set {highlightTag = value;}
  }
  public RichLabelFormattingOptions(RichLabelTextType type,
  string highlightTag)
  {
    this.highlightTag = highlightTag;
    this.type = type;
  }
}
```

Это решает проблемы синхронизации и редактирования, но не устраняет *всех* шероховатостей. Проблема в том, что хотя вы и можете редактировать свойство RichLabel.Format, установленная информация не сохраняется в дескрипторе элемента управления. Это значит, что изменения, проведенные во время проектирования, по сути, игнорируются. Чтобы решить эту проблему, вам придется погрузиться немного глубже в процесс сериализации элементов управления, осуществляемый .NET, что и описано в следующем разделе.

Атрибуты сериализации

Сериализацией свойств элемента управления в файле .aspx можно управлять с помощью атрибутов.

Вам нужно рассмотреть два ключевых атрибута — DesignerSerializationVisibility и PersistenceMode. Атрибут DesignerSerializationVisibility определяет сериализуемость свойства.

У вас есть три выбора (как определено в перечислении DesignerSerializationVisibility).

- **Visible**. Это значение по умолчанию. Оно специфицирует, что свойство должно быть сериализовано, и работает для простых типов данных (таких как строки, даты и перечисления), а также для числовых типов.

- **Content**. Это сериализует полное содержимое объекта. Данное значение можно использовать для сериализации сложных типов с множеством свойств — вроде коллекций.
- **Hidden**. Это указывает, что свойство вообще не подлежит сериализации. Например, вы можете использовать это для предотвращения сериализации вычисляемых значений.

Атрибут PersistenceMode позволяет задать, *как* сериализуется свойство. Вам доступны следующие выборы (как определено в перечислении PersistenceMode).

- **Attribute**. Значение по умолчанию. Свойство будет сериализовано как атрибут HTML элемента управления.
- **InnerProperty**. Свойство будет сохранено как вложенный дескриптор в элементе управления. Это — предпочтительная установка для генерации сложных вложенных иерархий объектов. Примерами могут служить элементы Calendar и DataList.
- **InnerDefaultProperty**. Свойство будет сохранено внутри управляющего дескриптора. Это будет единственное содержимое управляющего дескриптора. Пример — свойство Text элемента управления Label. При использовании свойства по умолчанию имя свойства не появляется во вложенном содержимом.
- **EncodedInnerDefaultProperty**. То же самое, что и InnerDefaultProperty, за исключением того, что содержимое будет закодировано HTML перед сохранением.

Чтобы понять, как работают эти разные опции, стоит рассмотреть несколько примеров. Выбор PersistenceMode.Attribute — опция по умолчанию, которую вы уже видели в базовом наборе управляющих дескрипторов ASP.NET. Если вы комбинируете этот атрибут с DesignerSerializationVisibility.Content в свойстве, чей тип содержит подсвойства, ASP.NET использует синтаксис прохода по объекту (object-walker syntax), в форме Свойство-Подсвойство="Значение". Вы можете видеть это на примере свойства Font, как показано здесь:

```
<apress:ctrl Font-Size="8pt" Font-Names="Tahoma" Bold="True" ... />
```

С другой стороны, рассмотрим, что происходит, если вы создаете пользовательский элемент управления, который переопределяет поведение сохранения свойство Font для использования PersistenceMode.InnerProperty, как показано ниже:

```
[PersistenceMode(PersistenceMode.InnerProperty)]
public override FontInfo Font
{
   get {return base.Font;}
}
```

Теперь сохраняемый код свойства Font принимает такую форму:

```
<apress:ctrl ... >
   <Font Size="8pt" Names="Tahoma" Bold="True"></Font>
</apress:ctrl>
```

Чтобы позволить RichLabel корректно сериализовать его свойство Format, необходимо применить оба атрибута — PersistenceMode и DesignerSerializationVisibility. Атрибут DesignerSerializationVisibility будет специфицировать Content, потому что свойство Format — это сложный объект. Атрибут PersistenceMode будет специфицировать InnerProperty, что сохранит информацию свойства Format как отдельный вложенный дескриптор. Вот как вы должны применить эти два атрибута для свойства RichLabelFormattingOptions.Format:

```
[TypeConverter(typeof(RichLabelFormattingOptionsConverter))]
[DesignerSerializationVisibility(DesignerSerializationVisibility.Content)]
[PersistenceMode(PersistenceMode.InnerProperty)]
public RichLabelFormattingOptions Format
{
   get {return (RichLabelFormattingOptions)ViewState["Format"];}
   set {ViewState["Format"] = value;}
}
```

Теперь, когда вы конфигурируете свойство `Format` в окне Properties, ASP.NET создаст дескриптор следующей формы:

```
<apress:RichLabel id="RichLabel1" runat="server">
   <Format Type="Xml" HighlightTag="b"></Format>
</apress:RichLabel>
```

В конечном итоге получаем превосходно работающий элемент управления `RichLabel` — и когда он помещается на веб-страницу во время выполнения, так и когда разработчик использует его во время проектирования.

Еще двумя свойствами, связанными с сериализацией, которые вы будете применять на уровне класса, являются `PersistChildren` и `ParseChildren`. Оба атрибута управляют тем, как ASP.NET обращается с вложенными дескрипторами и как поддерживает дочерние элементы. Когда `PersistChildren` установлено в `true`, дочерние элементы сохраняются как вложенные дескрипторы. Когда `PersistChildren` равно `false`, любые вложенные дескрипторы означают свойства. `ParseChildren` играет ту же роль при чтении управляющих дескрипторов. Когда `ParseChildren` равно `true`, анализатор ASP.NET интерпретирует все вложенные дескрипторы как свойства, а не элементы управления.

При наследовании от класса `WebControl` по умолчанию `PersistChildren` имеет значение `false`, а `ParseChildren` — `true`, и в этом случае любые вложенные дескрипторы трактуются как значения свойств. Если вы хотите, чтобы дочернее содержимое трактовалось как дочерние элементы управления в иерархии элементов, вы должны явно установить `PersistChildren` в `true`, а `ParseChildren` — в `false`. Поскольку элемент управления `RichLabel` не предназначен для включения вложенных элементов, этот шаг не требуется, — значения по умолчанию вам подходят.

`RichLabel` — не единственный элемент управления, который нуждается в атрибутах сериализации. Чтобы успешно использовать шаблонные элементы управления, описанные в главе 27 (наподобие `SuperSimpleRepeater`), все шаблонные свойства должны использовать сериализацию `PersistenceMode.InnerProperty`.

Рассмотрим пример корректно сконфигурированного шаблонного свойства:

```
[PersistenceMode(PersistenceMode.InnerProperty)]
[TemplateContainer(typeof(SimpleRepeaterItem))]
public ITemplate ItemTemplate
{
   get {return itemTemplate;}
   set {itemTemplate=value;}
}
```

В противном случае, когда вы установите другие свойства элемента управления, шаблонное содержимое будет стерто.

Редакторы типов

До сих пор вы видели, как конвертеры типов могут преобразовывать различные типы данных в представление, подходящее для окна Properties. Однако некоторые типы данных вообще не полагаются на редактирование строк. Например, если вам нужно перечислимое значение (вроде `BorderStyle`), вы можете выбрать из раскрывающегося

списка все значения перечисления. Еще более впечатляет выбор цвета из раскрывающейся палитры цветов. А некоторые свойства имеют возможность вообще "отрываться" от окна Properties (Свойства). Одним из примеров может служить свойство `Columns` элемента `GridView`. Если вы щелкаете на кнопке с троеточием рядом с именем свойства, появляется диалоговое окно, в котором можно конфигурировать коллекцию столбцов, используя богатый пользовательский интерфейс.

Эти свойства полагаются на пользовательский интерфейс *редакторов типов*. Перед редакторами типов в жизни стоит одна задача: они генерируют пользовательские интерфейсы, которые позволяют вам более удобно устанавливать свойства элементов управления. Некоторые типы данных (такие как коллекции, перечисления и цвета) автоматически ассоциируются с развитыми редакторами типов. В других случаях вы можете пожелать создать свои собственные классы редакторов типов "с нуля". Все пользовательские интерфейсы редакторов типов находятся в пространстве имен `System.Drawing.Design`.

Точно так же, как обстоят дела с конвертерами типов (и почти со всем в расширяемой архитектуре поддержки времени проектирования .NET), создание нового редактора типа включает наследование от базового класса (в данном случае — от `UITypeEditor`) и переопределение требуемых членов. Методы, которые можно переопределить, включают следующие.

- **`GetEditStyle()`**. Специфицирует, является ли редактор типа `DropDown` (представляющим список специально отображаемых вариантов выбора), `Modal` (представляющим диалоговое окно для выбора свойства) или `None` (нет поддержки редактирования).
- **`EditValue()`**. Этот метод вызывается, когда свойство редактируется (например, в окне Properties выполняется щелчок на кнопке с многоточием рядом с именем свойства). В общем, здесь вы должны создавать специальное диалоговое окно для редактирования свойства.
- **`GetPaintValueSupported()`**. Используйте этот метод, чтобы вернуть `true`, если вы представляете реализацию `PaintValue()`.
- **`PaintValue()`**. Вызывается для рисования пиктограммы, представляющей значение в таблице свойств. Например, используется для создания цветной рамки для свойства цвета.

Код пользовательского интерфейса редакторов типов не особенно сложен, но от веб-разработчиков потребует определенных усилий. Дело в том, что он предусматривает использование другой платформы интерфейсов пользователей .NET, а именно — Windows Forms. Хотя тема Windows Forms выходит за рамки настоящей книги, многому вы можете научиться из небольшого базового примера. На рис. 28.7 показан пользовательский элемент управления для редактирования цвета, который позволяет с помощью ползунков независимо устанавливать различные компоненты цвета. При этом полученный в результате цвет отображается в рамке в нижней части элемента.

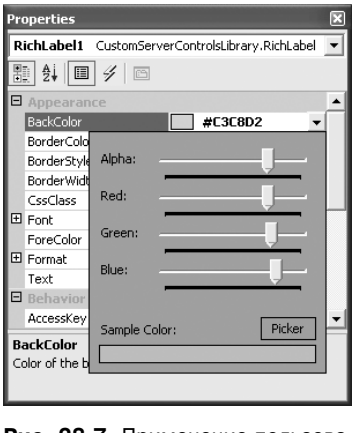

Рис. 28.7. Применение пользовательского редактора типа

Код самого элемента управления (`ColorTypeEditor Control`) здесь не показан, но вы можете обратиться к загружаемым примерам к этой главе, чтобы изучить его подробно.

Тем не менее, ниже показан полный код редактора типа, который использует этот элемент управления.

```
public class ColorTypeEditor : UITypeEditor
{
 public override UITypeEditorEditStyle GetEditStyle(
  ITypeDescriptorContext context)
 {
  // Этот редактор появляется по щелчку на стрелке вниз.
  return UITypeEditorEditStyle.DropDown;
 }
 public override object EditValue(ITypeDescriptorContext context,
  IServiceProvider provider, object value)
 {
  IWindowsFormsEditorService srv = null;
  // Получить от поставщика службу редактора, которая
  // понадобится для создания раскрывающегося окна.
  if (provider != null)
     srv = (IWindowsFormsEditorService)
            provider.GetService(typeof(IWindowsFormsEditorService));
  if (srv != null)
  {
    // Создать экземпляр пользовательского элемента
    // Windows Forms для выбора цвета.
    // Передать текущее значение цвета.
    ColorTypeEditorControl editor =
       new ColorTypeEditorControl((System.Drawing.Color)value,
       context.Instance as WebControl);
    // Показать элемент.
    srv.DropDownControl(editor);
    // Вернуть информацию о выбранном цвете.
    return editor.SelectedColor;
  }
  else
  {
    // Вернуть текущее значение.
    return value;
  }
 }
 public override bool GetPaintValueSupported(ITypeDescriptorContext context)
 {
  // Этот редактор типа будет генерировать пиктограмму.
  return true;
 }
 public override void PaintValue(PaintValueEventArgs e)
 {
  // Заполнить цветом левый прямоугольник.
  WebControl control = e.Context.Instance as WebControl;
  e.Graphics.FillRegion(new SolidBrush(control.BackColor), new Region(e.Bounds));
 }
}
```

Чтобы создать этот пример, вам нужно добавить ссылку на сборку `System.Windows.Forms`. Также не помешает импортировать пространство имен `System.Windows.Forms.Design`, содержащее интерфейс `IWindowsFormsEditorService`.

Чтобы использовать этот редактор типа, его нужно присоединить к свойству, имеющему тип данных `Color`. Большинство веб-элементов управления уже включают цветовые свойства, но вы можете переопределить одно из них и применить новый атрибут `Editor`.

Рассмотрим пример прикрепления редактора типа к свойству `BackColor` элемента управления `RichLabel`:

```
[Editor(typeof(ColorTypeEditor), typeof(UITypeEditor))]
public override Color BackColor
{
  get {return base.BackColor;}
  set {base.BackColor = value;}
}
```

Визуальные дизайнеры элементов управления

Дизайнер элемента управления влияет на его поведение и внешний вид во время проектирования. Каждый элемент управления использует готовый дизайнер из .NET Framework, в зависимости от класса, от которого он наследуется. Однако вы вольны переопределять эту деталь и конфигурировать свой элемент управления на использование специального дизайнера элементов. Чтобы создать специальный дизайнер элементов, вы наследуете его класс от `System.Web.UI.Design.ControlDesigner`.

Из всех вещей, касающихся времени проектирования, которые мы рассмотрим в этой главе, дизайнеры элементов управления наиболее сложны. В следующих разделах вы увидите, как выполнять две распространенные задачи с помощью специальных дизайнеров элементов управления. Во-первых, вы используете дизайнер элемента для настройки HTML времени проектирования для элемента. Затем вы примените дизайнер элемента для создания "интеллектуального" дескриптора. Эти два примера потребуют написания существенного куска кода, но, тем не менее, лишь вскользь затронут полный набор средств, которые может использовать опытный разработчик элементов. Например, вы можете применять специальный дизайнер элементов для дальнейшей настройки сериализации вложенного кода разметки вашего элемента, для фильтрации свойств, которые вы не хотите показывать во время проектирования, для создания свойств только времени проектирования либо предоставления средств редактирования шаблонов. Больше информации на эту тему вы найдете в документации MSDN или в специальных книгах, посвященных серверным элементам управления.

HTML времени проектирования

Возможно, вы заметили, что пользовательские элементы управления трактуются не одинаково на поверхности проектирования. ASP.NET старается отобразить их реалистичное представление времени проектирования, запуская логику визуализации, но бывают исключения. Например, композитные и шаблонные элементы управления вообще не визуализируются во время проектирования, а это означает, что вы не увидите на их месте ничего кроме пустого прямоугольника.

Чтобы справиться с этой проблемой, элементы управления часто используют так называемые пользовательские визуальные конструкторы, которые генерируют базовый HTML, предназначенный для отображения только во время проектирования. Это отображение может быть сложным блоком HTML, предназначенным для отражения реального внешнего вида элемента, базовым "снимком", демонстрирующим типичный пример элемента (как вы увидите на примере `GridView` или `FormView`, который не имеет никаких сконфигурированных столбцов) или просто серым прямоугольником с сообщением (как отображаются `ListView` и `DetailsView`, когда они не имеют никаких шаблонов).

Если вы хотите настроить HTML времени проектирования для своего элемента управления, то можете наследовать базовый класс `ControlDesigner` и переопределить один из его трех методов.

- **`GetDesignTimeHtml()`**. Возвращает HTML, который используется для представления текущего состояния элемента управления во время проектирования. Реализация этого метода по умолчанию просто возвращает результат вызова метода `RenderControl()`.
- **`GetEmptyDesignTimeHtml()`**. Возвращает HTML, который используется для представления пустого элемента управления. Реализация этого метода по умолчанию просто возвращает строку, содержащую имя класса и ID элемента управления.
- **`GetErrorDesignTimeHtml()`**. Возвращает HTML, который используется, если в элементе возникают ошибки во время проектирования. HTML может представлять информацию об исключении (которая передается в аргументе метода).

Конечно, эти методы отображают только малую часть функциональности, доступной через `ControlDesigner`. Вы можете переопределить намного больше методов, чтобы сконфигурировать различные аспекты поведения времени проектирования. В следующем разделе будет показано, как создать визуальный дизайнер элемента управления, который добавляет расширенную поддержку `SuperSimpleRepeater`.

В следующем примере представлен визуальный дизайнер элемента управления, который генерирует подходящее представление `SuperSimpleRepeater`, разработанного в предыдущей главе. Без пользовательского визуального дизайнера элемента содержимое `SuperSimpleRepeater` времени проектирования представлено пустой строкой.

Первый шаг при создании визуального дизайнера — построить класс, унаследованный от `ControlDesigner` из пространства имен `System.Web.UI.Design`, как показано ниже:

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

Вы можете применить этот визуальный дизайнер к своему элементу управления, указав его в атрибуте `Designer`:

```
[Designer(typeof(SuperSimpleRepeaterDesigner))]
public class SuperSimpleRepeater : WebControl, INamingContainer
{ ... }
```

При создании визуального дизайнера элемента управления первый шаг состоит в разработке метода `GetEmptyDesignTimeHtml()`. Этот метод должен вернуть просто статический кусок текста. `ControlDesigner` включает вспомогательный метод по имени `CreatePlaceHolderDesignTimeHtml()`, который генерирует HTML серого прямоугольника с сообщением, которое вы укажете (как это делает элемент `ListView` без каких-либо шаблонов). Вы можете использовать этот метод для упрощения кода генерации, как показано ниже:

```
protected override string GetEmptyDesignTimeHtml()
{
 string text = "Переключитесь в режим дизайна для добавления шаблона к этому элементу.";
 return CreatePlaceHolderDesignTimeHtml(text);
}
```

На рис. 28.8 виден пустой вид элемента управления `SuperSimpleRepeater` во время проектирования.

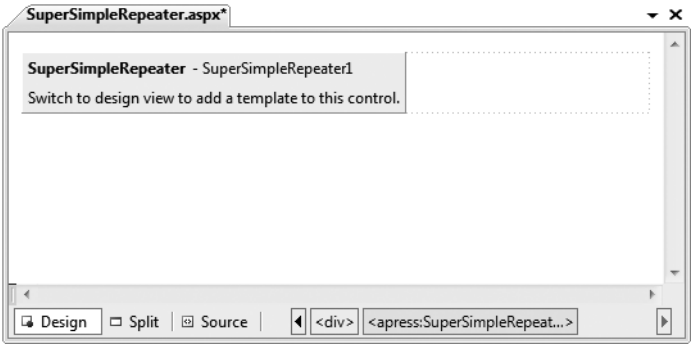

Рис. 28.8. Пустой вид элемента управления `SuperSimpleRepeater` во время проектирования

На заметку! Имейте в виду, что ASP.NET не может решить, когда ваш элемент пуст. Вместо этого вы должны вызывать при необходимости метод `GetEmptyDesignTimeHtml()`. Как вы увидите в этом примере, метод `GetDesignTimeHtml()` вызывает `GetEmptyDesignTimeHtml()`, если отсутствует шаблон.

Кодирование метода `GetErrorDesignTimeHtml()` почти так же просто. Опять-таки, вы можете использовать метод `CreatePlaceHolderDesignTimeHtml()`, но на этот раз передать подробности возникшего исключения:

```
protected override string GetErrorDesignTimeHtml(Exception e)
{
 string text = string.Format("{0}{1}{2}{3}",
  "Произошла ошибка и элемент не может быть отображен.",
  "<br />", "Exception: ", e.Message);
 return CreatePlaceHolderDesignTimeHtml(text);
}
```

И заключительный шаг — разработка метода `GetDesignTimeHtml()`. Этот код извлекает текущий экземпляр элемента управления `SuperSimpleRepeater` из свойства `ControlDesigner.Component`. Затем он проверяет шаблон. Если шаблона нет, отображается пустой HTML. Если шаблон присутствует, элемент управления привязывается к данным и *затем* отображается HTML времени проектирования, как показано ниже:

```
public override string GetDesignTimeHtml()
{
 try
 {
  SuperSimpleRepeater repeater = (SuperSimpleRepeater1)base.Component;
  if (repeater.ItemTemplate == null)
  {
    return GetEmptyDesignTimeHtml();
  }
  else
  {
    String designTimeHtml = String.Empty;
    repeater.DataBind();
    return base.GetDesignTimeHtml();
  }
  return base.GetDesignTimeHtml();
 }
```

```
 catch (Exception e)
 {
  return GetErrorDesignTimeHtml(e);
 }
}
```

Этот метод порождает солидно усовершенствованное представление времени проектирования, как показано на рис. 28.9. Это представление очень близко к тому внешнему виду, который имеет `SuperSimpleRepeater` во время выполнения.

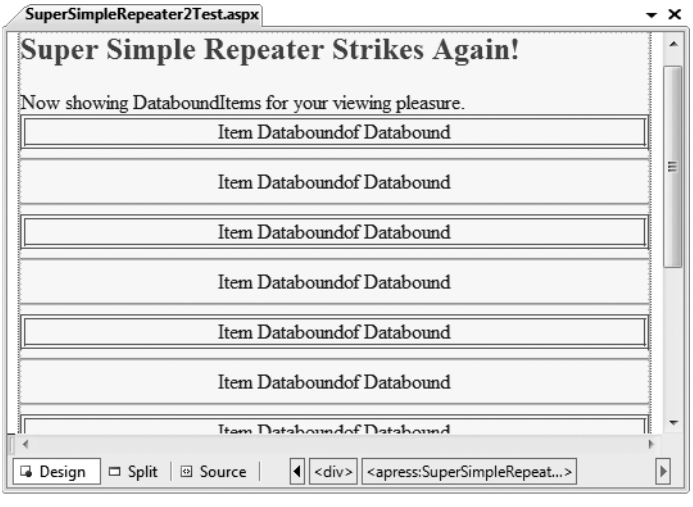

Рис. 28.9. Усовершенствованное представление элемента `SuperSimpleRepeater` во время проектирования

Интеллектуальные дескрипторы

В Visual Studio включено еще одно средство для создания богатого впечатления во время проектирования — *интеллектуальные дескрипторы* (smart tags — "смарт-тэги"). Они представляют собой всплывающие окна, которые появляются рядом с элементом управления, когда вы щелкаете на маленькой стрелке в углу.

Интеллектуальные дескрипторы похожи на меню в том, что они содержат списки элементов. Однако эти элементы могут быть командами (которые отображаются в виде гиперссылок) или другими элементами управления — такими как флажки, раскрывающиеся списки и тому подобное. Они также могут содержать статический описательный текст. Таким образом, интеллектуальный дескриптор может вести себя как маленькое окно Properties.

На рис. 28.10 показан пример пользовательского интеллектуального дескриптора, который создан в следующем примере. Он позволяет разработчику устанавливать комбинацию свойств `TitledTextBox` и включает в себя два текстовых поля, которые позволяют устанавливать текст, ссылку See Website Information (Информация о веб-сайте), которая вызывает браузер со специфическим URL, а также некоторую статическую информацию, указывающую имя элемента управления.

Чтобы создать этот интеллектуальный дескриптор, понадобятся следующие ингредиенты.

- *Коллекция объектов* `DesignerActionItem`. Каждый `DesignerActionItem` представляет одну позицию (элемент) интеллектуального дескриптора.

- *Класс-список действий.* Этот класс играет две роли — он конфигурирует коллекцию экземпляров `DesignerActionItem` интеллектуального дескриптора, и когда выполняется команда или проводится изменение, осуществляет соответствующую операцию над связанным элементом управления.
- *Визуальный конструктор элемента управления.* Прикрепляет ваш список действий к элементу управления, так что интеллектуальный дескриптор появляется во время проектирования.

В следующих разделах шаг за шагом мы с вами построим это решение.

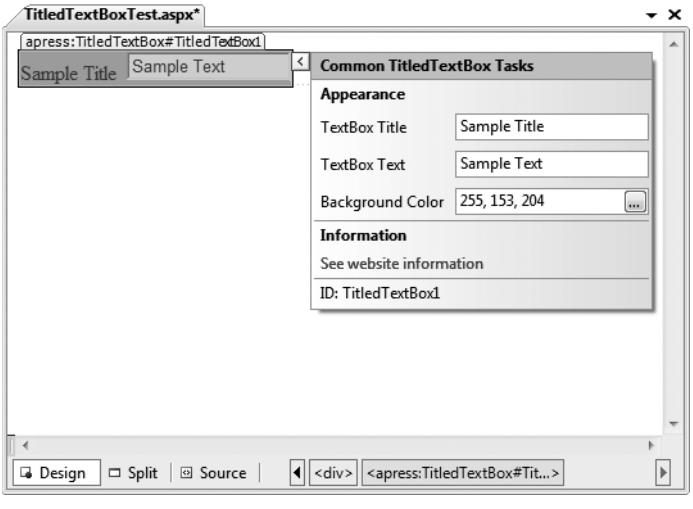

Рис. 28.10. Пользовательский интеллектуальный дескриптор

Список действий

Интеллектуальные дескрипторы предоставляют много вариантов выбора. Чтобы держать их хорошо организованными, неплохо бы отделить код, создав пользовательский класс, инкапсулирующий ваш список действий. Этот пользовательский класс должен наследоваться от `DesignerActionList` (из пространства имен `System.ComponentModel.Design`).

Ниже показан пример, создающий список действий, предназначенных для использования с `TitledTextBox`:

```
public class TitledTextBoxActionList : DesignerActionList
{ ... }
```

Вы должны добавить единственный конструктор, который принимает соответствующий тип элемента управления. Затем ссылку на него можно сохранить в переменной-члене. Это не обязательно, потому что базовый класс `ActionList` содержит свойство `Component`, обеспечивающее доступ к вашему элементу управления.

```
private TitledTextBox linkedControl;
public TitledTextBoxActionList(TitledTextBox ctrl) : base(ctrl)
{
   linkedControl = ctrl;
}
```

Прежде чем вы сможете построить интеллектуальный дескриптор, необходимо оснастить класс списка действий необходимыми членами. Для каждой ссылки, которую

вы хотите присоединить к дескриптору (через DesignerActionMethodItem), вы должны предусмотреть метод. Для каждого свойства, которое вы хотите добавить (через DesignerActionPropertyItem), необходимо создать процедуру свойства.

Интеллектуальный дескриптор на рис. 28.10 включает семь элементов: два заголовка категорий, три свойства, одну ссылку и один фрагмент статического текста (в нижней части дескриптора).

Первый шаг состоит в добавлении свойств. Процедура свойства get должна извлекать значения свойства из связанного элемента управления. Процедура свойства set должна применять новое значение к связанному элементу управления. Однако здесь таится ловушка — вы не можете устанавливать новое значение напрямую. Если этого не сделать, то другие части инфраструктуры визуального конструктора не будут извещены об изменении. Поэтому необходимо работать через метод PropertyDescriptor.SetValue(). Чтобы облегчить задачу, в классе-списке действий можно определить приватный вспомогательный метод, который извлекает PropertyDescriptor для заданного свойства по имени:

```
private PropertyDescriptor GetPropertyByName(string propName)
{
  PropertyDescriptor prop;
  prop = TypeDescriptor.GetProperties(linkedControl)[propName];
  if (null == prop)
  {
    throw new ArgumentException("Указанное свойство не найдено.", propName);
  }
  else
  {
    return prop;
  }
}
```

Теперь вы можете создать три свойства, который обертывают свойства в элементе управления TitledTextBox:

```
public string Text
{
  get { return linkedControl.Text; }
  set { GetPropertyByName("Text").SetValue(linkedControl, value); }
}
public string Title
{
  get { return linkedControl.Title; }
  set { GetPropertyByName("Title").SetValue(linkedControl, value); }
}
public Color BackColor
{
  get { return linkedControl.BackColor; }
  set { GetPropertyByName("BackColor").SetValue(linkedControl, value);
}
```

На заметку! Не все свойства могут быть отредактированы в интеллектуальном дескрипторе — это зависит от типа данных. Если тип данных имеет ассоциированный UITypeEditor (для графического редактирования свойства) или TypeConvertor (для преобразования типа данных в строковое представление и обратно), то редактирование будет работать. Наиболее распространенные типы данных обладают этими ингредиентами, но ваши пользовательские объекты — нет (в результате чего все, что вы увидите — это доступную только для чтения строку, сгенерированную вызовом ToString() объекта). За дополнительной информацией обращайтесь в следующую главу, где преобразование типов рассматривается в подробностях.

Следующий шаг — построение функциональности для ссылки See Website Information (Информация о веб-сайте). Чтобы сделать это, создадим метод в классе списка действий. Ниже приведен код, использующий класс `Process` для запуска браузера по умолчанию:

```
public void LaunchSite()
{
 try
 {
  System.Diagnostics.Process.Start("http://www.prosetech.com");
 }
 catch { }
}
```

Коллекция DesignerActionItem

Индивидуальные элементы интеллектуального дескриптора представлены классом `DesignerActionItem`. .NET Framework предлагает четыре базовых класса, унаследованных от `DesignerActionItem`, как показано в табл. 28.3.

Таблица 28.3. Классы-наследники DesignerActionItem

Метод	Описание
`DesignerActionMethodItem`	Этот элемент визуализируется как ссылка. Когда вы щелкаете на ней, инициируется действие путем вызова метода вашего класса `DesignerActionList`.
`DesignerActionPropertyItem`	Этот элемент визуализируется как редактирующий элемент управления и использует логику, подобную окну Properties. Строки редактируются через текстовые полях, перечисления — через раскрывающиеся списки, а булевские значения — через флажки. При изменении значения модифицируется лежащее в основе свойство.
`DesignerActionTextItem`	Этот элемент визуализируется как статический фрагмент текста. Обычно предоставляет дополнительную информацию об элементе управления. На щелчки не реагирует.
`DesignerActionHeaderItem`	Этот элемент унаследован от `DesignerActionTextItem` и представляет собой статический фрагмент текста, оформленный в виде заголовка. Используя один или более заголовочных элементов, вы можете разделить интеллектуальный дескриптор на отдельные категории и группировать свойства соответствующим образом. На щелчки не реагирует.

Чтобы создать собственный пользовательский интеллектуальный дескриптор, нужно построить коллекцию `DesignerActionItemCollection`, комбинирующую вашу группу объектов `DesignerActionItem`. Последовательность компонентов этой коллекции важна, потому что Visual Studio добавляет объекты `DesignerActionItem` к интеллектуальному дескриптору сверху вниз, в порядке их появления.

Чтобы построить список действий, вы переопределяете метод `DesignerActionList.GetSortedActionItems()`, создаете `DesignerActionItemCollection`, добавляете к ней каждый `DesignerActionItem` и возвращаете коллекцию. В зависимости от сложности вашего интеллектуального дескриптора, это может потребовать нескольких шагов.

Первый шаг состоит в создании двух заголовков, разделяющих интеллектуальный дескриптор на отдельные регионы. Затем вы можете добавлять другие элементы к этим категориям. В примере используются два заголовка:

```
public override DesignerActionItemCollection GetSortedActionItems()
{
   // Создать 8 элементов.
   DesignerActionItemCollection items = new DesignerActionItemCollection();
   // Начать с создания заголовков.
   items.Add(new DesignerActionHeaderItem("Appearance"));
   items.Add(new DesignerActionHeaderItem("Information"));
   ...
```

Далее можно добавлять свойства. Вы специфицируете имя свойства класса, за которым следует имя, которое должно появляться в интеллектуальном дескрипторе. Последние два элемента включают категорию, куда следует поместить элемент (соответственно одному из `DesignerActionHeaderItems`, которые были только что созданы) и описание (которое появляется как всплывающая подсказка при наведении курсора мыши).

```
...
// Добавить элемент, являющийся оболочкой свойства.
items.Add(new DesignerActionPropertyItem("Title",
   "TextBox Title", "Appearance",
   "The heading for this control."));
items.Add(new DesignerActionPropertyItem("Text",
  "TextBox Text", "Appearance",
  "The content in the TextBox."));
items.Add(new DesignerActionPropertyItem("BackColor",
  "Background Color", "Appearance",
  "The color shown behind the control as a background."));
...
```

Visual Studio подключает элемент действия к свойству в классе элемента действия, используя рефлексию с именем переданного свойства. Если вы добавляете более одного свойства в одну и ту же категорию, они выстраиваются в том порядке, в котором были добавлены. Если вы добавляете более одного заголовка категории, категории также размещаются согласно порядку добавления.

Следующий шаг — создание `DesignerActionMethodItem()`, который привязывает элемент интеллектуального дескриптора к методу. В данном случае вы специфицируете объект, в котором реализован метод обратного вызова, имя метода, затем имя, которое должно появляться при отображении интеллектуального дескриптора, затем категорию, где он должен появляться, и описание всплывающей подсказки. Последний параметр — булевское значение. Если оно равно `true`, элемент будет добавлен в контекстное меню элемента управления вместе с интеллектуальным дескриптором.

```
...
items.Add(new DesignerActionMethodItem(this,
  "LaunchSite", "See website information",
  "Information",
  "Opens a web browser with the company site.",
  true));
...
```

И, наконец, вы можете создать объекты `DesignerActionTextItem` со статическим текстом, который вы хотите отобразить, и вернуть полную коллекцию элементов, примерно так:

```
 ...
 items.Add(new DesignerActionTextItem(
  "ID: " + linkedControl.ID,
  "ID"));
 return items;
}
```

Визуальный дизайнер элемента управления

После того как вы довели до ума список действий интеллектуального дескриптора, его еще необходимо подключить к вашему элементу управления. Это делается путем создания пользовательского визуального конструктора и переопределения свойства `ActionLists`, чтобы оно возвращало экземпляр вашего пользовательского списка действий. Это демонстрируется в следующем визуальном конструкторе элемента управления. Обратите внимание, что список действий не создается при каждом вызове `ActionLists` — вместо этого он кэшируется в приватной переменной-члене для оптимизации производительности.

```
public class TitledTextBoxDesigner : ControlDesigner
{
  private DesignerActionListCollection actionLists;
  public override DesignerActionListCollection ActionLists
  {
    get
    {
      if (actionLists == null)
      {
        actionLists = new DesignerActionListCollection();
        actionLists.Add(
         new TitledTextBoxActionList((TitledTextBox)Control));
      }
      return actionLists;
    }
  }
}
```

Резюме

В настоящей главе мы прошлись по некоторым простым и сложным аспектам архитектуры времени проектирования .NET. Вы узнали, как сконфигурировать способ отображения свойств элемента управления в окне Properties (Свойства) и как обеспечить его сериализацию и синтаксический разбор. Многие наиболее сложные темы, такие как пользовательские визуальные конструкторы элементов управления, подробнее освещены в документации MSDN.