KAPITOLA 28

Podpora návrhového režimu

Na vlastní ovládací prvky jsou kladeny dva požadavky: musejí komunikovat s webovou stránkou (a s vaším kódem), když běží, a současně musí umět komunikovat s Visual Studiem v návrhovém režimu. Tyto dvě úlohy jsou navzájem propojeny. Mohou se ovšem vylepšovat a přizpůsobovat odděleně. Některé z nejpokročilejších ovládacích prvků ASP.NET (jako například `Wizard` a `GridView`) obsahují řadu inteligentních pomůcek určených pro návrhový režim, včetně výbavy, jež umožňuje konfigurovat složité vlastnosti a aplikovat motivy jediným kliknutím.

Patrně jste si už povšimli, že Visual Studio automaticky poskytuje vlastním ovládacím prvkům základní úroveň podpory pro návrhový režim. Například když zkompilujete projekt vlastního ovládacího prvku, přidají se ovládací prvky, které jsou v něm obsaženy, automaticky do Toolboxu. Pak je můžete přetáhnout a upustit na webovém formuláři, konfigurovat jejich vlastnosti, přiřazovat obsluhy událostí atd. Podle toho, jak jste implementovali realizační logiku, můžete v návrhovém režimu dokonce vidět HTML kód ovládacího prvku. V této kapitole se dozvíte, jak se rozšiřuje tato úroveň podpory návrhového režimu.

Mnohé z technik, které zde uvidíte, jsou jen parádičky a drobnosti usnadňující práci s vlastními ovládacími prvky. Podporu návrhového režimu můžete například využít pro přidání popisů, které se objevují v okně Properties, nebo pro realizaci reprezentativnějšího vzhledu ovládacího prvku na návrhové ploše. Jindy je ovšem podpora návrhového režimu důležitější. Vytvoříte-li například ovládací prvek, který vystavuje složité objekty jako vlastnosti, přičemž nepodniknete nezbytné kroky navíc pro přidání podpory návrhového režimu, ovládací prvek se bude v návrhovém režimu chovat nevyzpytatelně. Možná zjistíte, že vlastnosti, které nastavíte v okně Properties, se tu a tam nastaví zpět na výchozí hodnoty, nebo zmizí značky vnořených dceřiných ovládacích prvků. Tyto výstřednosti jsou důsledkem toho, jak ASP.NET serializuje vlastnosti ovládacího prvku. V této kapitole se dozvíte, jak se s takovými záležitostmi vypořádat.

Klíčoví hráči

V .NET není žádná jednotlivá třída, která by poskytovala podporu návrhovému režimu. Místo toho je zde celá řada ingrediencí, které se zapojují do podpory návrhového režimu:

- **Atributy**. Atributy se aplikují na části ovládacího prvku hned z několika důvodů. Zaprvé, atributy dodávají informace, jimiž se dolaďuje zobrazení v okně Properties, a ikony ovládacího prvku v Toolboxu. Zadruhé, atributy specifikují, jak se mají vlastnosti serializovat. Zatřetí, atributy umožňují

přiřazovat k ovládacímu prvku nějaké jiné komponenty návrhového režimu, jako jsou typové konvertory, typové editory a designéři ovládacího prvku.

- **Webové zdroje**. Zde se jimi rozumějí podpůrné soubory, které používá vlastní ovládací prvek. Pomocí webových zdrojů můžete vkládat tyto soubory do assembly vlastního ovládacího prvku a použít speciálně naformátovanou URL v případě, kdy je potřebujete získat.
- **Typové konvertory**. Umožňují převádět složité nebo neobvyklé datové typy na reprezentace v běžnějších datových typech a naopak. Pokud například vytvoříte typový editor, který umožní převod z vlastního datového typu na řetězec, nebo naopak řetězec do vlastního datového typu, budete moci v okně Properties prohlížet a editovat vlastnost ovládacího prvku, jež používá tento datový typ. Typové konvertory mohou hrát také jistou roli při serializaci tím, že vygenerují inicializační kód, který se požaduje, když se vytváří instance nějakého složitého typu.
- **Typové editory**. Poskytují grafické rozhraní pro nastavování hodnot složitých typů. Pokud například v okně Properties vybíráte z rozevíracího seznamu název písma pro webový ovládací prvek, v tomto okamžiku používáte typový editor.
- **Designéři ovládacího prvku**. Supertěžké váhy pro vývoj vlastního ovládacího prvku. Každý ovládací prvek má svého designéra, který spravuje jeho vzhled a chování v návrhovém režimu. Designéra vlastního ovládacího prvku můžete využít nejenom k různým parádičkám (jako jsou například inteligentní značky), ale také pro poskytování výchozího obsahu HTML, který má Visual Studio použít k reprezentaci ovládacího prvku na návrhové ploše. Pomocí designéra můžete v návrhovém režimu skrýt vlastnosti třídy ovládacího prvku, nebo přidat vlastnosti určené pouze pro návrhový režim.

V této kapitole prozkoumáme několik vlastních ovládacích prvků, s nimiž jste se seznámili v kapitole 27, a budeme zvažovat, jak bychom mohli se všemi těmito ingrediencemi zdokonalit jejich podporu pro návrhový režim. Začneme tím, že tyto ovládací prvky dovybavíme vlastními ikonami pro Toolbox a řádnou podporou pro okno Properties. Pak se dozvíte, jak se dá dokončit základní serializace ovládacího prvku prostřednictvím atributů a typových konvertorů. Následně uvidíte, jak se dají vylepšit zážitky s ovládacím prvkem v návrhovém režimu pomocí typových editorů, protože se pak snadněji nastavují složité vlastnosti.

A nakonec uvidíte, jak se používají designéři ovládacího prvku u třech běžně se vyskytujících úloh, kterými jsou větší přizpůsobení serializace ovládacího prvku, změna výchozí reprezentace HTML vašeho ovládacího prvku a přidání inteligentní značky pro běžné konfigurační úlohy ovládacího prvku.

Atributy návrhového režimu

První úroveň podpory návrhového režimu představují atributy ovládacího prvku – deklarativní indikátory, které se zkompilují do metadat assembly vlastního ovládacího prvku. Atributy nabízejí způsob, jak přidávat informace vztahující se k jisté části kódu, aniž byste museli měnit kód nebo vytvářet samostatný soubor v nějakém úplně jiném formátu.

Atributy jsou vždy v hranatých závorkách a umisťují se před kód, který modifikují. Následující příklad ilustruje, jak můžete přidat atribut, který poskytuje popis pro vlastnost `Text` ovládacího prvku:

```
[Description("The text to be shown in the control")]
public string Text
{ ... }
```

V tomto případě atribut `Description` dekoruje vlastnost `Text`.

POZNÁMKA *Všechny atributy jsou ve skutečnosti třídy. Na základě dohody bylo stanoveno, že název třídy bude končit na slovo* `Attribute`. *Například atribut* `Description` *je vlastně reprezentován třídou* `DescriptionAttribute`. *Ačkoliv můžete používat plný název třídy, kompilátor C# povoluje zkrácenou verzi, takže můžete koncové slovo* `Attribute` *vynechat.*

V .NET se atributy používají pro nejrůznější potřeby. U atributů musíte pochopit jednu základní věc – mohou být čteny a interpretovány různými agenty. Můžete například přidat atribut, který předává informace CLR, kompilátoru nebo nějakému vlastnímu nástroji. Tato kapitola je především zaměřena na atributy, které poskytují informace Visual Studiu a sdělují mu, jak má pracovat s ovládacím prvkem v návrhovém režimu. Později v oddílu "Serializace ovládacích prvků" se seznámíte s některými atributy, jež ovlivňují proces, jakým parser ASP.NET interpretuje značky ovládacího prvku v souboru `.aspx`.

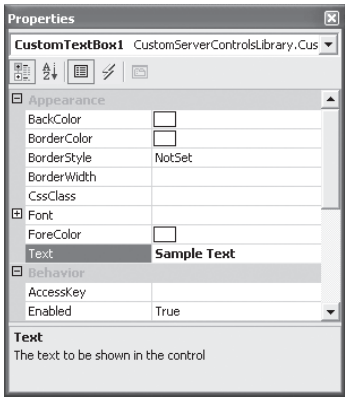
TIP *Podobně jako mnohé jiné třídy pro podporu návrhového režimu, i většina z atributů, s nimiž se seznámíte v této kapitole, se nachází ve jmenném prostoru* `System.ComponentModel`. *Než tyto atributy použijete, měli byste jmenný prostor importovat do souborů kódu vašich vlastních ovládacích prvků.*

Okno Properties

Nejjednodušší atributy ovlivňují, jak se vlastnosti ovládacího prvku projevují v okně Properties. Například jste si pravděpodobně všimli, že základní sada webových ovládacích prvků ASP.NET seskupuje své vlastnosti do několika kategorií. Když vyberete některou z vlastností, zobrazí okno Properties její stručný popis. Chcete-li přidat tuto informaci k nějakému vlastnímu ovládacímu prvku, musíte každou vlastnost dekorovat atributy `Category` a `Description`, jak vidíte zde:

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

Jak je zde názorně předvedeno, oba atributy `Category` a `Description` přebírají jako argument jediný řetězec. Na obrázku 28-1 vidíte výsledné zobrazení, když se v okně Properties vybere vlastnost `Text`.



Obrázek 28-1. *Vlastnost s popisem.*

Klíčové atributy, které ovlivňují způsob, jakým je příslušná vlastnost zobrazena v okně Properties, jsou uvedeny v tabulce 28-1.

Tabulka 28-1. Atributy pro vlastnosti ovládacího prvku.

Atribut	*Popis*
`Browsable(true|false)`	*Je-li nastaven na hodnotu* `false`*, vlastnost se neobjeví v okně Properties (ačkoliv ji programátor může změnit buď v kódu, nebo ručním přidáním atributu značky ovládacího prvku, pokud zařadíte proceduru vlastnosti* `Set`*). Jedním z důvodů, proč byste měli tento atribut používat, je snaha ukrýt vypočítávané vlastnosti nebo runtime vlastnosti, které se nemohou měnit v návrhovém režimu.*
`Category(řetězec)`	*Řetězec indikující kategorii, pod kterou se vlastnost objeví v okně Properties.*
`Description(řetězec)`	*Řetězec indikující popis, který bude mít vlastnost, když se vybere v okně Properties.*
`DefaultValue()`	*Nastaví výchozí hodnotu, která bude u dané vlastnosti zobrazena v okně Properties. Obvyklou výchozí hodnotou je počáteční hodnota, v takovém případě atribut* `DefaultValue` *nepotřebujete. Použití tohoto atributu ovšem někdy umožní generátoru kódu optimalizovat značky, které generuje (tím, že vynechá informaci, pokud se shoduje s výchozí hodnotou).*
`Themeable(true|false)`	*Všechny vlastní ovládací prvky automaticky podporují motivy (popisují se v kapitole 16). Pokud ovšem nechcete, aby bylo možné určitou hodnotu nakonfigurovat v rámci skinu, použijte atribut* `Themeable` *s hodnotou* `false`*. Na rozdíl od ostatních atributů uvedených v této tabulce je atribut* `Themeable` *definován ve jmenném prostoru* `System.Web.UI`*.*

Atribut	Popis
`Localizable(true|false)`	*Lokalizace je zapnuta u všech ovládacích prvků a objektů. Pokud je vlastnost lokalizovatelná, Visual Studio umožní, aby její hodnoty zůstávaly v satelitní assembly. Pokud tuto funkci nechcete povolit, nebo pokud máte vlastnost, která by se neměla měnit v závislosti na lokalizaci, nastavte tento atribut na* `false`.
`ReadOnly(true|false)`	*Pokud je nastaven na* `true`, *je vlastnost v návrhovém režimu nastavena v okně Properties jen pro čtení.*
`DesignOnly(true|false)`	*Pokud je tento atribut nastaven na* `true`, *vlastnost bude dostupná pouze v návrhovém režimu. Obvykle se využívá u speciálních vlastností, jež konfigurují chování ovládacího prvku v návrhovém režimu, které neodpovídají žádné "opravdové" informaci o ovládacím prvku.*
`ImmutableObject(true|false)`	*Pokud je nastaven na* `true` *u vlastnosti objektu, zajistí, že podvlastnosti tohoto objektu budou zobrazeny pouze pro čtení. Aplikujete-li například tento atribut u vlastnosti, která používá objekt* `Point`, *podvlastnosti X a Y budou pouze pro čtení.*
`MergableProperty(true|false)`	*Konfiguruje chování okna Properties, pokud je současně vybráno několik instancí daného ovládacího prvku. Pokud je tento atribut nastaven na* `false`, *vlastnost se nezobrazí. Je-li nastaven na* `true` *(výchozí nastavení), dá se vlastnost nastavit u všech vybraných ovládacích prvků najednou.*
`ParenthesizePropertyName (true|false)`	*Je-li tento atribut nastaven na hodnotu* `true`, *Visual Studio zobrazí kulaté závorky kolem dané vlastnosti v okně Properties (jako u vlastnosti ID).*
`Bindable(true|false)`	*Je-li nastaven na* `true`, *Visual Studio zobrazí tuto vlastnost v dialogovém okně* `DataBindings` *a umožní ji svázat s polem ve zdroji dat.*
`RefreshProperties(hodnota výčtu RefreshProperties)`	*Tento atribut specifikuje hodnotu z výčtu* `RefreshProperties`. *Jde o to, zdali se bude zbytek okna Properties aktualizovat, když se změní vlastnost (například tehdy, pokud by procedura jedné vlastnosti změnila jinou vlastnost).*

Další dva atributy, `DefaultEvent` a `DefaultProperty`, se aplikují spíše při deklaraci třídy vlastního ovládacího prvku, než u specifické vlastnosti. Atribut `TagPrefix` se dále používá na úrovni assembly a není přiřazen k nějaké konstrukci kódu. Atributy pro třídy a assembly jsou popsány v tabulce 28-2.

Tabulka 28-2. Atributy pro třídy a assembly ovládacího prvku.

Atribut	Popis
`DefaultEvent(řetězec)`	*Označuje název výchozí události. Kliknete-li dvojitě na ovládacím prvku na návrhové ploše, Visual Studio automaticky přidá obsluhu k výchozí události. (Pokud není definována výchozí událost, dvojitým kliknutím jednoduše prvek vyberete.)*

Atribut	*Popis*
`DefaultProperty(řetězec)`	*Označuje název výchozí vlastnosti. Vlastnost* `DefaultProperty` *je v okně Properties standardně zvýrazněna, když se ovládací prvek vybere poprvé. (Není-li definovaná výchozí vlastnost, nebude vybrána žádná vlastnost, když se poprvé vybere ovládací prvek.)*
`ControlValuePropertyAttribute (řetězec)`	*Vyjadřuje název vlastnosti, která se má standardně použít, když se váže* `ControlParameter` *na tento ovládací prvek pro použití ve zdroji dat (popisuje se to v kapitole 9).*
`NonVisualControl()`	*Vyjadřuje, že ovládací prvek nebude mít při běhu viditelný vzhled. V návrhovém režimu můžete zvolit View -> Visual Aids -> ASP.NET Non-Visual Controls, pokud chcete skrýt všechny ovládací prvky, které mají atribut* `NonVisualControl`*. Příkladem "neviditelných" ovládacích prvků jsou ovládací prvky pro zdroje dat.*
`TagPrefix(řetězec, řetězec)`	*Sdruží jmenný prostor s prefixem, který se použije, když se přidávají značky ovládacího prvku na* `.aspx` *stránku.*
`ToolboxBitmap(type, string)`	*Specifikuje bitmapu, která bude tento ovládací prvek reprezentovat v Toolboxu. Standardně používá Visual Studio obecnou ikonu ozubeného kola.*
`ToolboxData(string)`	*Specifikuje značku, která se pro tento ovládací prvek vytvoří v souboru* `.aspx`*, když přetáhnete ikonu tohoto ovládacího prvku z Toolboxu. Standardně vytvoří Visual Studio prázdnou značku obsahující pouze atributy* `ID` *a* `Runat`*.*

Jak jste se dozvěděli v kapitole 27, každý vlastní ovládací prvek má prefix, který se registruje direktivou `Register` na `.aspx` stránce. Visual Studio tuto direktivu přidá automaticky, když ovládací prvek vložíte. Chcete-li si tento prefix přizpůsobit, můžete použít atribut `TagPrefix`, který přebírá dva parametry řetězce. První řetězec je název jmenného prostoru, ve kterém se nacházejí vaše ovládací prvky. Druhým řetězcem je prefix značky, který chcete použít.

Následující příklad specifikuje, že ovládací prvky v `CustomServerControlsLibrary` mají používat předponu `apress`:

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

Nevkládejte atribut assembly dovnitř bloku jmenného prostoru. Měl by být definován vně všech jmenných prostorů, aby měl globální obor.

Pokud nyní přidáte ovládací prvek s názvem třídy `CustomTextBox` ze jmenného prostoru `CustomServerControlsLibrary`, použije Visual Studio tuto značku:

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

Pokud máte ovládací prvky ve více jmenných prostorech, musíte `TagPrefix` použít vícekrát – u každého jmenného prostoru jednou. Můžete použít stejný prefix, nebo různé prefixy. Atribut `TagPrefix` bývá často umístěn v souboru `AssemblyInfo.cs`.

Přetáhnete-li vlastní ovládací prvek z Toolboxu, Visual Studio standardně vytvoří značku ovládacího prvku s atributy `ID` a `runat`, jako to vidíte zde:

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

Atributem `ToolboxData` můžete dodat i jiné značkování, které může zahrnovat i výchozí hodnoty. Ukázka:

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

Povšimněte si zástupného symbolu `{0}`, který reprezentuje prefix značky. Když specifikujete hodnoty atributů s atributem `ToolboxData`, často se vynechávají oddělovače uvozovek (jako v tomto příkladě u atributů `ID` a `Runat`). Oddělovače jsou ovšem nezbytné, pracujete-li s řetězci, které obsahují mezery nebo speciální znaky (jako je to u atributu `Text`). Chcete-li do řetězce `ToolboxData` zařadit uvozovky, musíte je řádně předznačit prostřednictvím tzv. únikového znaku `\"`.

Když nyní přetáhnete `CustomTextBox` na formulář, dostanete tuto značku:

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

Pokud vedete všechny tyto detaily v patrnosti, jste dostatečně vybaveni k tomu, abyste obyčejnému vlastnímu ovládacímu prvku mohli dodat respekt vzbuzující úroveň podpory návrhového režimu. Zde máte obyčejný ovládací prvek `CustomTextBox` z předchozí kapitoly, ale s kompletní výbavou atributů. Kód jsme vynechali.

```
[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)
    { ... }
  }
}
```

Atributy a dědičnost

Pokud odvozujete ovládací prvek ze základní třídy, která má atributy pro návrhový režim, ovládací prvek po svém rodiči zdědí funkcionalitu pro návrhový režim (stejně jako dědí metody a vlastnosti). Pokud je implementace rodičovské třídy atributů pro návrhový režim nějakého vašeho ovládacího prvku postačující, nemusíte ji opětovně aplikovat.

V některých případech ovšem chcete v návrhovém režimu změnit chování existující vlastnosti. Pak musíte vlastnost překrýt a opětovně použít již změněné atributy, nebo přidat nové atributy.

Většina vlastností v základních třídách `WebControl` a `Control` je označena jako virtuální vlastnosti, což povoluje změnit jejich chování. Pokud byste například chtěli skrýt vlastnost `Height` vlastního ovládacího prvku, který je odvozen z `WebControl` (možná proto, že se vypočítává z obsahu; vývojář ji nenastavuje), můžete vlastnost `Height` překrýt a aplikovat atribut `Browsable`, jak to vidíte zde:

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

Ikona v Toolboxu

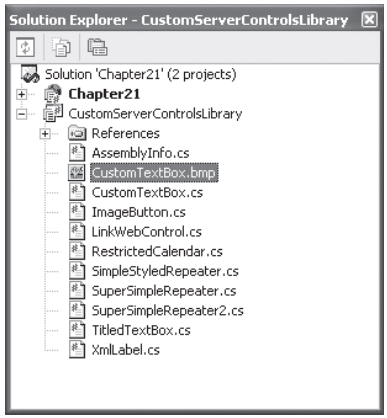
Ikona vlastního ovládacího prvku se do Toolboxu přidá velmi jednoduše. Do vašeho projektu přidáte bitmapový obrázek, který ovšem musí vyhovovat několika následujícím pravidlům:

- Soubor bitmapy musí mít stejný název souboru jako má třída vlastního ovládacího prvku (ovšem s příponou `.bmp`). Například pro ovládací prvek `CustomTextBox` byste použili bitmapu s názvem `CustomTextBox.bmp`.
- Bitmapa musí mít rozměry 16x16 pixelů, jinak bude deformovaná, jakmile se Visual Studio pokusí tuto ikonu zpracovat.
- Musí používat pouze 16 barev.
- Jakmile přidáte soubor bitmapy, musíte v okně Properties nastavit Build Action na Embedded Resource.

Požadovaný obrázek pro ovládací prvek `CustomTextBox` zobrazuje obrázek 28-2.

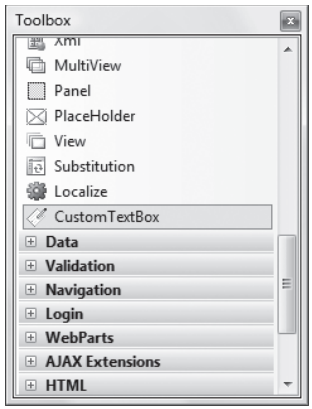
Jakmile přidáte správný obrázek a nastavíte vlastnost Build Action na Embedded Resource, znovu zkompilujte projekt ovládacího prvku. Své vlastní ikony ovšem neuvidíte na té záložce Toolboxu, kterou Visual Studio automaticky přidá pro váš projekt ovládacího prvku. Vlastní ikony uvidíte pouze tehdy, pokud ručně nakonfigurujete Toolbox a přidáte vlastní ovládací prvek na některou z trvalých záložek Toolboxu. Dělá se to

tak, že na dané záložce kliknete pravým tlačítkem myši, vyberete Choose Items, a pak procházením naleznete assembly, která má vaši knihovnu ovládacího prvku (jak jsme to popisovali v kapitole 27).



Obrázek 28-2. *Přidání bitmapové ikony do Toolboxu.*

Obrázek 28-3 demonstruje příklad se dvěma vlastními ovládacími prvky. Jeden má obecnou ikonu ozubeného kola, zatímco `CustomTextBox` vlastní bitmapový obrázek.



Obrázek 28-3. *Vlastní bitmapový obrázek v Toolboxu.*

Mimochodem, v Toolboxu je možné použít i ikonu, jejíž název souboru není shodný s názvem třídy vlastního ovládacího prvku. V takovém případě ale potřebujete výpomoc od atributu `ToolboxBitmap`. Například – následující kód nakonfiguruje ovládací prvek `CustomTextBox` tak, aby použil bitmapu s názvem `CustomTextBox1.bmp`:

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

Někdy může být žádoucí uložit bitmapy do nějaké samostatné podsložky projektu. Například, takhle je možné se odkázat na bitmapu z adresáře s názvem `Images`:

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

A konečně, je také možné "krást" bitmapy ze základních ovládacích prvků ASP.NET:

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

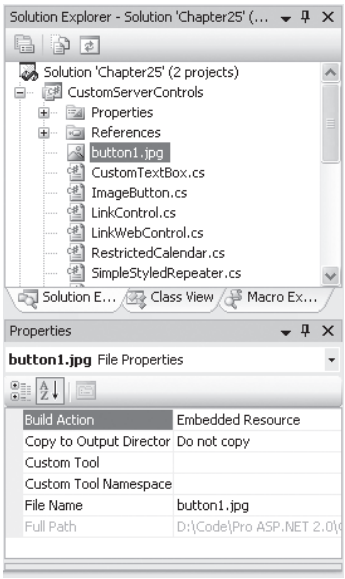
Pokud vytváříte jednoduchý ovládací prvek, často dostačuje, když přidáte pouze sadu popisných vlastností a ikonu pro Toolbox. Složitější ovládací prvky ovšem mnohdy vyžadují více. Musíte promyslet všechny kroky od serializace kódu (jak se vytvoří značka ovládacího prvku pomocí okna Properties) až k designérům ovládacího prvku (což jsou pokročilé nástroje pro přizpůsobování HTML kódu vašeho ovládacího prvku v době návrhu). Právě těmto tématům se budeme věnovat ve zbývajících oddílech kapitoly.

Webové zdroje

S vlastními ovládacími prvky jsou často sdruženy jiné zdroje (někdy se říká prostředky, resources), které nemají charakter souborů s kódem. Mohou to být například soubory skriptů, stylové předpisy nebo obrázky, které potřebujete používat s ovládacím prvkem. Pochopitelně se objevuje problém s rozmisťováním, protože tyto zdrojové soubory musíte zkopírovat do adresáře každé webové aplikace, která používá váš ovládací prvek. Naštěstí tohle ASP.NET řeší za vás, pomocí modelu webových zdrojů (web resources model).

Vytvoření zdroje

Chcete-li vytvořit webový zdroj, začněte tím, že soubory zdroje přidáte do projektu vlastního ovládacího prvku. Pak musíte v okně Properties změnit vlastnost Build Action na Embedded Resource (vložený zdroj), nesmí to být Content (viz obrázek 28-4). Takto zajistíte, že soubor se vloží do zkompilované assembly.



Obrázek 28-4. *Vložený zdroj.*

Dalším krokem je zpřístupnit URL zdroje. To se dělá atributem `WebResource` na úrovni assembly.

Představte si například, že chcete přistoupit k obrázku tlačítka, který se používá pro ovládací prvek `CustomImageButton`. Jakmile jste změnili nastavení vlastnosti Build Action, musíte do kódu přidat tento atribut:

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

Atribut `WebResource` přebírá dva parametry. První parametr je celý název vloženého zdroje, který musí jako prefix obsahovat standardní jmenný prostor vašeho projektu. (Tento detail, který uvidíte, když v okně průzkumníka projektu kliknete pravým tlačítkem myši na projektu a vyberete Properties, se automaticky přidá na začátek názvu zdroje). Druhým parametrem je typ obsahu.

Po těchto krocích musíte provést poslední věc – získat URL pro vložený zdroj. ASP.NET tuto operaci podporuje přes ovladač `WebResource.axd`. Tento ovladač přijímá požadavky na URL, extrahuje zdroj z patřičné assembly a poté vrací obsah. Řečeno jinými slovy, nemusíte se potýkat se změtí obrázků a skriptů, protože soubor `WebResource.axd` je obsluhuje podle potřeby (přímo z vaší assembly vlastního ovládacího prvku).

Získání zdroje

Pokud chcete získat svůj zdroj, musíte ovladači `WebResource.axd` sdělit, jaký zdroj vlastně potřebujete a která assembly jej obsahuje. Metodou `Page.ClientScript.GetWebResourceUrl()` pak vygenerujete správnou URL. Tady máte ukázku, ve které se touto technikou vytáhne výchozí obrázek pro ovládací prvek `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;}
}
```

POZNÁMKA *Mimochodem – webové zdroje automaticky počítají s lokalizací. Pokud máte satelitní assembly zdroje s lokalizovanou verzí tohoto obrázkového souboru, bude použita právě tato verze. Více informací naleznete v první bonusové kapitole, kterou naleznete v sekci Download webu* `http://www.zonerpress.cz`.

Skutečná URL vypadá přibližně takto:

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

Parametr `a` dotazovacího řetězce specifikuje název assembly, parametr `r` pak název zdroje. Další parametr `t` je náhradní řešení pro cachování. `WebResource.axd` v podstatě ukládá do cache každý požadovaný zdroj. Můžete tedy třeba tisíckrát požádat o jeden stejný obrázek, aniž byste neustále vnášeli dodatečnou zátěž tím, že obrázek by se znovu a znovu získával z assembly. (Toto je důležité pro zajištění optimálního výkonu zejména tehdy, pokud je ve vaší assembly zabaleno velké množství zdrojů). Ukládání do cache ovšem samo o sobě vnáší svůj specifický problém. Jestliže assembly obsahuje novější verzi zdroje, určitě nebudete chtít opětovně použít zdroj z cache. Tohle právě zajišťuje parametr `t`. Je to časová známka assembly. Když se assembly vlast-

ního ovládacího prvku znovu sestaví, budou mít vygenerované URL adresy odlišné hodnoty parametru `t`. V důsledku toho prohlížeč učiní nový požadavek, takže ovladač `WebResource.axd` získá nejnovější obsah.

Pokud si to chcete vyzkoušet, kompletní příklad s ovládacím prvkem `CustomImageButton` můžete najít ve zdrojových souborech k této knize.

Nahrazení textu

Ovladač `WebResource.axd` má v záloze ještě jeden trik. Je totiž možné nastavit logickou vlastnost `PerformSubstitution` atributu `WebResource`, čímž mu sdělujete, aby vykonával automatické nahrazování (substitution). To vám umožňuje vytvořit textový zdroj, který ukazuje na jiné vložené zdroje. Dejme tomu, že máte webový ovládací prvek, který dodává nápovědu pomocí několika souborů HTML. Vy chcete propojit tyto soubory HTML hypertextovými odkazy, nicméně všechny je chcete vložit jako webové zdroje.

Kdyby nebylo k dispozici automatické nahrazování, nebyli byste schopni vytvořit zdroj, který by ukazoval na jiný zdroj, protože předem neznáte názvy zdrojů, jež vygeneruje ASP.NET. S automatickým nahrazováním ovšem můžete použít původní název souboru. Kompilátor totiž nahradí vaši referenci na název souboru správným, automaticky vygenerovaným, názvem zdroje.

Představte si například, že jste vytvořili soubor s názvem `CustomServerControls.Help.htm`, který obsahuje hypertextové odkazy na jiné zdroje. Podívejte se na atribut, který potřebujete pro soubor `CustomServerControls.Help.htm`:

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

Nyní `WebResource.axd` projde celý text souboru zdroje (v tomto případě se jedná o HTML soubor s názvem `CustomServerControls.Help.htm`), přičemž vyhledává výrazy v tomto formátu:

```
<%= WebResource(...) %>
```

Například následujícím značkováním byste zobrazili obrázek s názvem `HelpTitle.gif`, který je rovněž webovým zdrojem:

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

Pokaždé, když kompilátor najde nějakou takovou referenci, nahradí ji URL adresou odpovídajícího webového zdroje (v tomto případě to bude URL ukazující na soubor `HelpTitle.gif`). Ukázka:

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

Jakmile se dokončí transformace, modifikovaná verze `CustomServerControls.Help.htm` se vloží do assembly v podobě zdroje.

POZNÁMKA *Automatické nahrazování funguje pouze u textových zdrojů.*

Serializace kódu

Konfigurujete-li vlastnosti ovládacího prvku v okně Properties, Visual Studio musí mít možnost vytvořit a modifikovat odpovídající značku ovládacího prvku v `.aspx` souboru. Tomuto procesu se říká serializace kódu a často se děje automaticky. Pokud ovšem používáte vlastnosti, které jsou samy o sobě složitými typy, nebo pokud vytváříte šablonový ovládací prvek či nějaký ovládací prvek, jenž podporuje dceřiné ovládací prvky, můžete se dostat do potíží.

V následujících oddílech se dozvíte, co všechno ovlivňuje serializaci ovládacích prvků, a co je potřeba změnit, abyste vyřešili běžně se vyskytující problémy.

Typové konvertory

Okno Properties hladce zpracovává běžné datové typy. Nedělají mu žádné potíže řetězce, které dokonce umí převádět na číselné typy. Pokud se například podíváte na vlastnost `Width` ovládacího prvku, uvidíte něco jako 50 px. Do tohoto pole můžete vložit jakékoliv znaky. Ovšem pokud vložíte znaky, které se nedají interpretovat jako nějaká (délková) jednotka, a pokusíte se změnu potvrdit (tím, že stisknete klávesu Enter, nebo se přesunete do dalšího pole), vaše snaha o změnu bude zamítnuta.

Toto chování způsobují typové konvertory, což jsou specializované třídy, jejichž jediným úkolem je převádět specializované datové typy na řetězcovou reprezentaci a zpět. Většina hlavních datových typů .NET používá standardní typové konvertory, které pracují velice dobře. (Tyto konvertory naleznete ve jmenném prostoru `System.ComponentModel`). Pokud si ovšem vytvoříte vlastní struktury nebo třídy, které použijete jako vlastnosti, s velkou pravděpodobností také budete chtít vytvořit vlastní typové konvertory, jež jim umožní fungovat v okně Properties.

Ovládací prvek s objektovými vlastnostmi

V následujícím příkladu je uveden ovládací prvek `RichLabel`, který představuje mírně revidovanou verzi ovládacího prvku `XmlLabel`, o němž jsme hovořili v kapitole 27. Rozdíl mezi nimi je v tom, že zatímco prvek `XmlLabel` má za úkol zobrazovat pouze XML dokumenty, ovládací prvek `RichLabel` bude podporovat různé typy obsahu.

Prvek `RichLabel` bude v podstatě schopen podporovat jakýkoliv typ obsahu, který je definován v následujícím výčtu `RichLabelTextType`. V tomto jednoduchém příkladu obsahuje výčet `RichLabelTextType` pouze dvě volby: `Xml` (používá stejný kód jako `XmlLabel`) a `Html` (pracuje s textem v nezměněné podobě, a dále ho již nezpracovává). Snadno si ovšem můžete přidat další realizační kód pro odlišné typy textů.

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

Prvek `RichLabel` také umožní volit, která značka se má použít pro formátování důležitých detailů (jako jsou například XML značky v realizačním režimu XML). Funguje to prostřednictvím jiné třídy, s názvem `RichLabelFormattingOptions`. Třída `RichLabelFormattingOptions` definuje dvě vlastnosti: `Type` (reprezentuje hodnotu z výčtu `RichLabelTextType`) a `HighlightTag` (ukládá název značky jako řetězec, například `b` pro značku `<b>`, která zvýrazňuje text tučně).

```
[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;
  }
}
```

Třída `RichLabel` obsahuje vlastnost `Format`, která vystavuje instanci vlastní třídy `RichLabelFormatting-Options`. Realizační logika v ovládacím prvku `RichLabel` používá tuto informaci pro úpravu HTML kódu, který generuje.

A zde máte kód ovládacího prvku `RichLabel`:

```
[DefaultProperty("RichText")]
public class RichLabel : WebControl
{
  public RichLabel()
  {
    Text = "";
    // Výchozí je XML text se značkami naformátovanými tučně.
    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:
        // Najde a zvýrazní XML značky.
        convertedText = RichLabel.ConvertXmlTextToHtmlText(
        Text, Format.HighlightTag);
        break;
      case RichLabelTextType.Html:
        // Ponechá text takový, jaký je.
        convertedText = Text;
        break;
    }
    output.Write(convertedText);
  }
  public static string ConvertXmlTextToHtmlText(string inputText,
  string highlightTag)
  {
    // (Kód jsme vynechali.)
  }
}
```

Pochopitelně existují i alternativní návrhy. Tyto dvě informace (`Type` a `HighlightTag`) můžete například přidat do třídy `RichLabel` jako samostatné vlastnosti. V tom případě nemusíte podnikat žádné další kroky pro zajištění správné serializace. Z mnoha různých důvodů se ovšem můžete rozhodnout seskupit příbuzné vlastnosti pomocí vlastní třídy (jako je `RichLabelFormattingOptions`). Možná chcete třídu `RichLabelFormattingOptions` využívat opakovaně, abyste s ní mohli specifikovat volby pro formátování textu i v jiných ovládacích prvcích. Nebo potřebujete vytvořit složitější ovládací prvek, který bude akceptovat několik odlišných textů, přičemž je dokáže všechny zkonvertovat pomocí nezávislých nastavení `RichLabelFormattingOptions`. V obou těchto situacích bude velmi užitečné seskupit do třídy `RichLabelFormattingOptions` všechny příbuzné vlastnosti.

Ovládací prvek `RichLabel` bohužel dobře nespolupracuje s Visual Studiem. Pokud se jej pokusíte modifikovat v návrhovém režimu, okamžitě narazíte na potíže. Okno Properties vám totiž nedovolí editovat vlastnost `RichLabel.Format`. Zobrazí jen prázdné editační pole, kam nejde nic napsat. Abyste tento problém vyřešili, musíte si vytvořit vlastní typový konvertor. Nevíte jak? To vám vysvětlíme v příštím oddílu.

Vytvoření vlastního typového konvertoru

Vlastní typový konvertor je třída, která dokáže převádět z vašeho datového typu (v tomto případě se jedná o třídu `RichLabelFormattingOptions`) na řetězec a zpět. V následujícím příkladu uvidíte takovou třídu, která se jmenuje `RichLabelFormattingOptionsConverter`.

Prvním krokem je vytvořit vlastní třídu, která je odvozena ze základní třídy `TypeConverter`, jak vidíte zde:

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

Dodržuje se konvence, že název třídy typového konvertoru se skládá z názvu typu třídy (kterou konvertor převádí) a slova `Converter`.

Jakmile máte vytvořený typový konvertor, musíte překrýt několik metod:

- **CanConvertFrom()**. Prozkoumá datový typ a vrací hodnotu `true`, pokud typový konvertor může provést konverzi z tohoto datového typu na váš vlastní datový typ.
- **ConvertFrom()**. Provádí konverzi z dodaného datového typu na váš vlastní datový typ.
- **CanConvertTo()**. Prozkoumá datový typ a vrací hodnotu `true`, pokud typový konvertor může provést konverzi z vašeho vlastního objektu na tento datový typ.
- **ConvertTo()**. Provádí konverzi z vašeho vlastního datového typu na požadovaný datový typ.

Zapamatujte si, že základním úkolem typového konvertoru je provádět konverzi mezi vaším vlastním typem dat a řetězcovou reprezentací. V tomto příkladu je použita taková řetězcová reprezentace, která obsahuje obě hodnoty z objektu `RichLabelFormattingOptions`. Hodnoty jsou odděleny mezerou. Od názvu značky ve špičatých závorkách jsou odděleny pomocí čárky a mezery. Formát řetězce tedy vypadá následovně:

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

Zde je uveden příklad s formátováním XML a značkou `<b>`:

```
Xml, <b>
```

Když tohle vedete v patrnosti, můžete ve třídě konvertoru vytvořit dvě pomocné metody, které tuto konverzi zařídí. První z nich je metoda `ToString()`, která vytvoří požadovanou řetězcovou reprezentaci:

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

Druhou metodou je `FromString()`, která řetězcovou reprezentaci dekóduje. Pokud není řetězec ve formátu, který potřebujete, kód `FromString()` vyvolá výjimku. Jinak vrátí instanci nového objektu.

```
private RichLabelFormattingOptions FromString(object value)
{
  string[] values = ((string)value).Split(,,‘);
  if (values.Length != 2)
  throw new ArgumentException("Could not convert the value");
  try
  {
    // Převede název uvedené hodnoty na odpovídající
    // hodnotu (což je v tomto případě celočíselná konstanta).
    RichLabelTextType type = (RichLabelTextType)Enum.Parse(
    typeof(RichLabelTextType), values[0], true);
    // Odstraní mezery a úhlové závorky kolem názvu značky.
```

```
    string tag = values[1].Trim(new char[]{, ,,'<','>'});
    return new RichLabelFormattingOptions(type, tag);
  }
  catch
  {
    throw new ArgumentException("Could not convert the value");
  }
}
```

Předtím, než se okno Properties pustí do konverze řetězce na objekt `RichLabelFormattingOptions`, metodou `CanConvertFrom()` vykoná dotaz. Pokud obdrží hodnotu `true`, zavolá aktuální metodu `ConvertFrom()`. Metoda `CanConvertFrom()` níže uvedeným způsobem zkontroluje, zdali je dodaný typ řetězcem:

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

Metoda `ConvertFrom()` následně vyvolá konverzi tím, že zavolá metodu `FromString()` uvedenou výše:

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

POZNÁMKA *Při objektově orientovaném programování je běžnou programátorskou praxí vždy specifikovat základní třídy, z nichž dědíte, čímž vzniká šance zpracovat zprávu, kterou nehodláte podporovat. V takovém případě se všechny požadavky na konverzi z neznámého typu předají do základní třídy.*

Stejný proces, ovšem v opačném směru, nastává při konverzi objektu `RichLabelFormattingOptions` na řetězec. Okno Properties nejdříve zavolá `CanConvertTo()`. Pokud se vrátí hodnota `true`, v dalším kroku se zavolá metoda `ConvertTo()`. Zde máte potřebný kód:

```
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);
  }
}
```

Nyní máte plně funkční typový konvertor, který v dalším kroku připojíte k odpovídající vlastnosti.

Připojení typového konvertoru

Konvertor typu můžete připojit dvěma způsoby. K související třídě (v tomto případě `RichLabelFormattingOptions`) můžete přidat atribut `TypeConverter`:

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

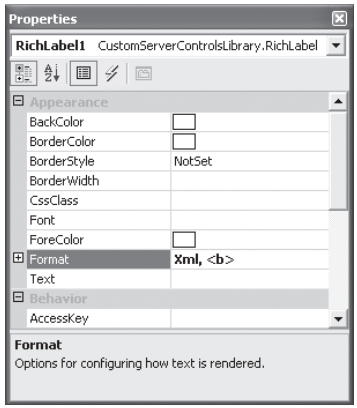
Vždy, když se pro ovládací prvek použije instance této třídy, Visual Studio bude vědět, že má použít váš typový konvertor. Alternativou je připojit typový konvertor ve vlastním ovládacím prvku přímo k vlastnosti, která jej používá, jak vidíte zde:

```
public class RichLabel : WebControl
{
  [TypeConverter(typeof(RichLabelFormattingOptionsConverter))]
```

```
  public RichLabelFormattingOptions Format
  { ... }
  ...
}
```

Tento přístup má největší smysl tehdy, když používáte nějaký obecný datový typ (například řetězec), a kdy chcete přizpůsobit jeho chování pouze v tomto jediném případě.

Nyní můžete kód znovu zkompilovat a pokusit se použít ovládací prvek `RichLabel` ve vzorové webové stránce. Jakmile vyberete `RichLabel`, uvidíte v okně Properties aktuální hodnotu vlastnosti `RichLabel.Format` (viz obrázek 28-5), kterou můžete ručně upravovat.



Obrázek 28-5. *Řetězcová reprezentace objektu* `RichLabelFormattingOptions`.

POZNÁMKA *Pokud měníte jisté detaily, jako je typový konvertor, designér ovládacího prvku nebo tvůrce ovládacího prvku, změny se v návrhovém režimu prostředí neprojeví okamžitě po rekompilaci. Pravděpodobně budete muset zavřít řešení s testovacími webovými stránkami a znovu je otevřít.*

Samozřejmě, pokud nezadáte správnou řetězcovou reprezentaci, dostanete chybovou zprávu a vaše změna bude zamítnuta. Řečeno jinak – zde uvedený vlastní typový konvertor vám sice dává možnost specifikovat objekt `RichLabelFormattingOptions` jako řetězec, nicméně tento proces není vůbec uživatelsky přívětivý.

V další sekci se dozvíte, jak vylepšit tuto úroveň podpory.

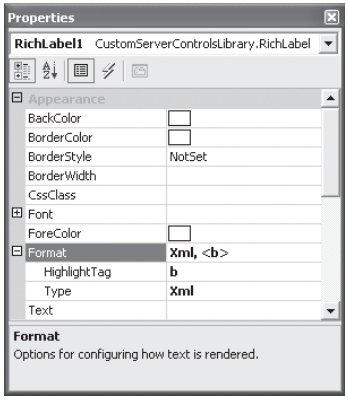
ExpandableObjectConverter

Webové ovládací prvky ASP.NET podporují nejrůznější objektové vlastnosti. Nejlepším příkladem je vlastnost `Font`, která se odkazuje na objekt `FontInfo` s vlastnostmi jako je `Bold`, `Italic`, `Name` atd. Vyberete-li vlastnost `Font` v okně Properties, nemusíte psát všechny tyto informace do jediného, správně naformátovaného řetězce. Kliknutím na ikonku + si vlastnost `Font` rozbalíte a editujete jednotlivé vlastnosti `FontInfo`.

Stejný typ editace můžete zapnout i pro vlastní objektové typy – potřebujete k tomu pouze vytvořit vlastní typový konvertor, který bude odvozen z třídy `ExpandableObjectConverter`, nikoliv ze základní třídy `TypeConverter`. Například můžete vzít třídu `RichLabelFormattingOptionsConverter`, kterou jsme vytvořili v předchozím oddílu, a změnit ji takto:

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

Nyní můžete specifikovat vlastnost `Format` tak, že buď vložíte řetězec, nebo rozbalíte vlastnost a pozměníte jednu ze dvou podvlastností. Toto mnohem přívětivější rozhraní okna Properties vidíte na obrázku 28-6.



Obrázek 28-6. *Editace vlastností objektu RichLabelFormattingOptions.*

Na první pohled je rozhraní velmi uspokojivé, má ovšem několik háčků. Prvním je to, že pokud změníte jednu z podvlastností (`Type` nebo `HighlightTag`), řetězcová reprezentace, která je zobrazena v poli Format, nebude ihned aktualizována. Chcete-li to vyřešit, musíte na vlastnosti třídy `RichLabelFormattingOptions` aplikovat atributy `NotifyParentProperty` a `RefreshProperties`. Současně přidejte atribut `Description`, abyste mohli nastavit popisný text, který se v okně Properties objeví pro podvlastnost.

Zde máte revidovaný kód třídy `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;
  }
}
```

Tímto jsme se zbavili potíží se synchronizací a editací, ovšem pořád leccos zbývá dořešit. Problém je v tom, že ačkoliv můžete editovat vlastnost `RichLabel.Format`, informace, kterou nastavíte, nebude ve značce ovládacího prvku umístěna natrvalo. To znamená, že změny, které provedete v návrhovém režimu, se de facto ignorují. Pokud chcete vyřešit tento problém, musíte více do hloubky pochopit, jak .NET serializuje vlastnosti ovládacího prvku, což se popisuje v příštím oddílu.

Serializační atributy

Pomocí atributů můžete řídit, jak se budou vlastnosti ovládacího prvku serializovat do `.aspx` souboru. Klíčové jsou dva atributy – `DesignerSerializationVisibility` a `PersistenceMode`.

Atribut `DesignerSerializationVisibility` určuje, zda se bude vlastnost serializovat. Máte tři volby (určují je hodnoty výčtu `DesignerSerializationVisibility`):

- **Visible**. Výchozí hodnota. Označuje, že vlastnost se má serializovat. Používá se nejenom u jednoduchých typů dat (jako jsou řetězec, datum, nebo výčet), ale také u číselných typů dat.
- **Content**. Bude se serializovat kompletní obsah objektu. Dá se použít pro serializaci složitých typů s mnoha vlastnostmi, jako je například kolekce.
- **Hidden**. Specifikuje, že vlastnost se vůbec nemá serializovat. Můžete ji například využít k tomu, aby se neserializovaly vypočítávané hodnoty.

Atribut `PersistenceMode` umožňuje specifikovat, jak se bude vlastnost serializovat. K dispozici máte tyto volby (určují je hodnoty výčtu `PersistenceMode`):

- **Attribute**. Výchozí volba. Vlastnost se bude serializovat jako HTML atribut ovládacího prvku.
- **InnerProperty**. Vlastnost bude setrvávat uvnitř ovládacího prvku jako vnořená značka. Je to preferované nastavení, chcete-li generovat složité vnořované hierarchie objektů. Příkladem jsou ovládací prvky `Calendar` a `GridView`.
- **InnerDefaultProperty**. Tato vlastnost bude setrvávat uvnitř značky ovládacího prvku. Bude jediným obsahem značky ovládacího prvku. Příkladem je vlastnost `Text` ovládacího prvku `Label`. Používáte-li výchozí vlastnost, její název se neobjeví ve vnořeném obsahu.
- **EncodedInnerDefaultProperty**. Tato vlastnost je stejná jako `InnerDefaultProperty`, liší se pouze tím, že obsah se zakóduje jako HTML předtím, než se stane trvalým.

Abyste snadněji pochopili, jak jednotlivé volby fungují, bude vhodné uvést několik příkladů. Volba `PersistenceMode.Attribute` je výchozí volbou, kterou uvidíte v základní sadě značek ovládacích prvků ASP.NET. Pokud zkombinujete tento atribut s `DesignerSerializationVisibility.Content` u vlastnosti, která

obsahuje nějaké podvlastnosti, ASP.NET použije syntaxi ve tvaru `VlastnostPodvlastnost="Hodnota"`. Tuto syntaxi můžete vidět aplikovanou na vlastnost `Font` v následující ukázce:

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

Zamyslete se ovšem nad tím, co se stane, pokud vytvoříte vlastní ovládací prvek, který překryje chování vlastnosti `Font` tak, aby bylo použito nastavení `PersistenceMode.InnerProperty`, jak vidíte zde:

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

Trvalý kód vlastnosti `Font` poté bude vypadat takto:

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

Aby mohl ovládací prvek `RichLabel` správně serializovat vlastnost `Format`, musíte aplikovat oba atributy, `PersistenceMode` i `DesignerSerializationVisibility`. Atribut `DesignerSerializationVisibility` bude specifikovat `Content`, protože vlastnost `Format` je složený objekt. Atribut `PersistenceMode` bude specifikovat `InnerProperty`, takže informace o vlastnosti `Format` se uloží v podobě samostatné vnořené značky. Zde je uveden způsob, jak tyto dva atributy aplikovat na vlastnost `RichLabelFormattingOptions.Format`:

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

Nyní, když budete vlastnost `Format` konfigurovat v okně Properties, ASP.NET vytvoří tuto značku:

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

Konečným výsledkem bude to, že když vložíte ovládací prvek `RichLabel` do webové stránky, bude fungovat perfektně jak při běhu aplikace, tak i v návrhovém režimu.

Na úrovni třídy můžete použít dvě další vlastnosti související se serializací – `PersistChildren` a `ParseChildren`. Oba atributy řídí, jak ASP.NET zachází s vnořenými značkami, a zdali podporuje dceřiné ovládací prvky. Má-li atribut `PersistChildren` hodnotu `true`, přetrvávají dceřiné ovládací prvky jako vnořené značky. Má-li `PersistChildren` hodnotu `false`, všechny vnořené značky vyznačují vlastnosti. `ParseChildren` hraje stejnou roli při čtení značek ovládacích prvků. Má-li atribut `ParseChildren` hodnotu `true`, parser ASP.NET interpretuje všechny vnořené značky jako vlastnosti, nikoliv jako ovládací prvky.

Při odvozování z třídy `WebControl` má `PersistChildren` výchozí hodnotu `false` a `ParseChildren` hodnotu `true`. V tomto případě se tedy s libovolnými vnořenými značkami zachází jako s hodnotami vlastností. Pokud chcete, aby se s dceřiným obsahem v hierarchii ovládacích prvků zacházelo jako s dceřinými ovládacími prvky, musíte explicitně nastavit `PersistChildren` na hodnotu `true` a `ParseChildren` na `false`. Jelikož ovládací prvek `RichLabel` není navržen tak, aby mohl obsahovat další ovládací prvky, tento krok není nutný – výchozí hodnoty jsou již nastaveny tak, jak potřebujete.

`RichLabel` není jediným ovládacím prvkem, který potřebuje serializační atributy. Abyste mohli úspěšně používat šablonové ovládací prvky, které byly popsány v kapitole 27 (jako je prvek `SuperSimpleRepeater`), musí všechny vlastnosti šablony používat serializaci `PersistenceMode.InnerProperty`.

A zde máte ukázku správně nakonfigurované šablonové vlastnosti:

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

Jestliže nastavíte v ovládacím prvku nějaké jiné vlastnosti, obsah šablony bude vymazán.

Typové editory

Zatím jste pouze viděli, jak mohou typové konvertory konvertovat různé datové typy na řetězce pro potřeby reprezentace v okně `Properties`. Některé datové typy se ovšem vůbec nespoléhají na editaci řetězců. Pokud například potřebujete nastavit výčtovou hodnotu (jako je `BorderStyle`), vybíráte ji z rozevíracího seznamu, který obsahuje všechny hodnoty výčtu. Ještě působivěji se nastavuje barva. Volíte ji z rozevírací palety barev (tzv. color picker). Některé vlastnosti mají dokonce takovou výbavu, že zcela obcházejí okno Properties. Příkladem je vlastnost `Columns` z `GridView`. Kliknete-li na tři tečky vedle názvu vlastnosti, objeví se dialog, ve kterém můžete nakonfigurovat kolekci sloupce pomocí bohatě vybaveného uživatelského rozhraní.

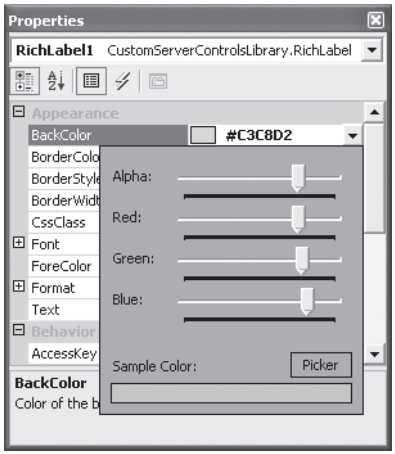
Všechny tyto vlastnosti se spoléhají na typové editory uživatelského rozhraní. Typové editory mají pouze jediný úkol – generovat uživatelská rozhraní, která vám umožní pohodlněji nastavovat vlastnosti ovládacího prvku. Jisté datové typy (jako například kolekce, výčty a barvy) jsou automaticky sdruženy s pokročilými typovými editory. Jindy budete muset vytvářet své vlastní třídy typových editorů úplně od začátku. Všechny typové editory uživatelského rozhraní jsou umístěny ve jmenném prostoru `System.Drawing.Design`.

Právě tak jako u typových konvertorů (a téměř všeho ostatního v rozšiřitelné architektuře .NET ohledně podpory návrhového režimu), i zde je vytvoření nového typového editoru založeno na dědění ze základní třídy (v tomto případě se jedná o třídu `UITypeEditor`) a překrytí požadovaných členů. Metody, které můžete překrývat, jsou následující:

- **GetEditStyle()**. Specifikuje, zdali je typový editor `DropDown` (tj. poskytuje rozvírací seznam voleb), `Modal` (poskytuje pro výběr vlastnosti dialog), nebo `None` (nemá žádnou podporu editace).
- **EditValue()**. Volá se, když se vlastnost edituje (například po kliknutí na tři tečky vedle názvu vlastnosti v okně Properties). Obvykle se tím vytváří speciální dialog pro editaci vlastností.
- **GetPaintValueSupported()**. Vraťte hodnotu `true`, poskytujete-li implementaci `PaintValue()`.

- **PaintValue()**. Volá se, aby se namalovala grafická miniatura reprezentující hodnotu v mřížce vlastností. Vhodná je tehdy, pokud např. chcete pro vlastnosti vyjadřující barvy vytvořit čtverečky barev.

Ačkoliv kód typových editorů uživatelského rozhraní nebývá příliš komplikovaný, někdy chvíli trvá, než si na něj weboví vývojáři přivyknou. Je to proto, že kód vyžaduje použití jiného uživatelského rozhraní .NET – a sice Windows Forms. Ačkoliv téma formulářů Windows nepochybně přesahuje rámec této knihy, můžete se o něm hodně naučit prostřednictvím následujícího triviálního příkladu. Na obrázku 28-7 vidíte vlastní ovládací prvek pro editaci barev, který vám umožňuje nastavovat jednotlivé komponenty barvy nezávisle na ostatních pomocí posuvníků. Když posunujete jednotlivými jezdci posuvníků, dole se zobrazuje výsledná barva (pod textem Sample Color).



Obrázek 28-7. *Vlastní typový editor.*

Kód tohoto ovládacího prvku (`ColorTypeEditorControl`) v této knize sice uveden není, nicméně jej můžete najít ve zdrojových příkladech k této knize, které jsou ke stažení na adrese `www.zonerpress.cz`. Kompletní kód typového editoru, který používá tento ovládací prvek, zde ovšem uveden je:

```
public class ColorTypeEditor : UITypeEditor
{
  public override UITypeEditorEditStyle GetEditStyle(
  ITypeDescriptorContext context)
  {
    // Tento editor se objeví, kliknete-li na rozbalovací šipku.
    return UITypeEditorEditStyle.DropDown;
  }
  public override object EditValue(ITypeDescriptorContext context,
  IServiceProvider provider, object value)
  {
    IWindowsFormsEditorService srv = null;
    // Získá od poskytovatele editační službu,
    // potřebnou pro vytvoření rozvíracího okna.
    if (provider != null)
    srv = (IWindowsFormsEditorService)
```

```
    provider.GetService(typeof(IWindowsFormsEditorService));
    if (srv != null)
    {
      // Vytvoří instanci vlastního ovládacího prvku
      // color-picker z Windows Forms
      // Předá aktuální hodnotu barvy.
      ColorTypeEditorControl editor =
      new ColorTypeEditorControl((System.Drawing.Color)value,
      context.Instance as WebControl);
      // Zobrazí ovládací prvek.
      srv.DropDownControl(editor);
      // Vrátí informaci o vybrané barvě.
      return editor.SelectedColor;
    }
    else
    {
      // Vrátí aktuální hodnotu.
      return value;
    }
  }
  public override bool GetPaintValueSupported(ITypeDescriptorContext context)
  {
    // Tento typový editor generuje miniaturu pole barvy.
    return true;
  }
  public override void PaintValue(PaintValueEventArgs e)
  {
    // Vyplní obdélník barvou.
    WebControl control = e.Context.Instance as WebControl;
    e.Graphics.FillRegion(new SolidBrush(control.BackColor),
    new Region(e.Bounds));
  }
}
```

Pokud chcete vytvořit tento příklad, musíte přidat referenci na assembly `System.Windows.Forms`. Pomůže také, když importujete jmenný prostor `System.Windows.Forms.Design`, který obsahuje rozhraní `IWindowsFormsEditorService`.

Pokud chcete tento typový editor používat, musíte jej připojit k vlastnosti, která používá datový typ `Color`. Většina webových ovládacích prvků již obsahuje vlastnosti pro barvy, takže můžete jednu z nich překrýt a aplikovat nový atribut `Editor`.

Zde uvádíme příklad, který přesně ilustruje tento postup. Připojuje typový editor k vlastnosti `BackColor` ovládacího prvku `RichLabel`:

```
[Editor(typeof(ColorTypeEditor), typeof(UITypeEditor))]
public override Color BackColor
{
```

```
  get {return base.BackColor;}
  set {base.BackColor = value;}
}
```

Designéři ovládacího prvku

Designér ovládacího prvku ovlivňuje chování a vzhled ovládacího prvku v návrhovém režimu. Každý ovládací prvek používá předem vyhotoveného designéra z .NET Frameworku, v závislosti na třídě, ze které dědí. Tento detail ovšem můžete svobodně překrýt a nakonfigurovat svůj ovládací prvek tak, aby používal vlastního designéra ovládacího prvku. Vlastního designéra ovládacího prvku vytvoříte tak, že odvodíte třídu ze `System.Web.UI.Design.ControlDesigner`. Pak můžete do designéra ovládacího prvku přidat svou vlastní funkcionalitu tím, že překryjete jeho základní metody.

Ze všech témat týkajících se návrhového režimu, která se probírají v této kapitole, jsou designéři ovládacího prvku nejnáročnější. V příštích oddílech uvidíte, jak se s vlastními designéry ovládacího prvku řeší dvě poměrně běžné úlohy. V té první uvidíte, jak se designérem ovládacího prvku přizpůsobuje HTML návrhového režimu ovládacího prvku. Následně prostřednictvím designéra ovládacího prvku vytvoříte inteligentní značku. Ačkoliv tyto dva příklady vyžadují poměrně hodně kódu, pořád se jedná o poměrně malou sadu schopností designéra ovládacího prvku, které může ostřílený vývojář ovládacího prvku využívat. Například – pomocí designéra ovládacího prvku můžete ještě více sofistikovaněji přizpůsobovat způsob, jakým se bude serializovat vnořené značkování uvnitř ovládacího prvku, odfiltrovat vlastnosti, které nechcete zobrazovat v návrhovém režimu, vytvořit vlastnosti určené pouze pro návrhový režim, nebo poskytnout šablonovou editační funkcionalitu. Další informace k této problematice naleznete v dokumentaci MSDN, nebo pokračujte se svým bádáním s nějakou knihou, která se kompletně věnuje serverovým ovládacím prvkům.

HTML v návrhovém režimu

Pravděpodobně jste si už povšimli, že na návrhové ploše Visual Studia není se všemi vlastními ovládacími prvky nakládáno úplně stejně. ASP.NET se snaží v návrhovém režimu zobrazit realistickou podobu ovládacího prvku tím, že spouští realizační logiku, nicméně existují určité výjimky. Například složené ovládací prvky a šablonové ovládací prvky se v návrhovém režimu vůbec nerealizují, což znamená, že na vaší návrhové ploše uvidíte pouze prázdný obdélník.

Tyto záležitosti se často řeší tak, že ovládací prvky používají vlastního designéra ovládacího prvku, který vyprodukuje základní HTML kód, jenž je určen pouze pro potřeby zobrazení v návrhovém režimu. Toto zobrazení může být buď promyšlený blok HTML kódu, který má reflektovat skutečný vzhled ovládacího prvku, nebo nějaká momentka zobrazující typický vzhled daného ovládacího prvku (jak to vidíte u prvků `GridView` a `FormView`, jež nemají nakonfigurované sloupce), nebo se jedná o obyčejný šedý rámeček se zprávou (jak to vidíte u prvků `ListView` nebo `DetailsView`, které nemají žádné šablony).

Pokud si chcete přizpůsobit HTML kód, který je určen pro zobrazování vlastního ovládacího prvku v návrhovém režimu, můžete si odvodit vlastního designéra ze základní třídy `ControlDesigner` a překrýt jednu z následujících tří metod:

- **GetDesignTimeHtml()**. Vrací HTML kód, který reprezentuje aktuální stav ovládacího prvku v návrhovém režimu. Výchozí implementace této metody jednoduše vrací výsledek volání metody `RenderControl()`.

- **GetEmptyDesignTimeHtml()**. Vrací HTML, který reprezentuje prázdný ovládací prvek. Výchozí implementace této metody jednoduše vrací řetězec, který obsahuje název třídy ovládacího prvku a `ID`.
- **GetErrorDesignTimeHtml()**. Vrací HTML kód, který je použit, pokud se v návrhovém režimu vyskytne v ovládacím prvku nějaká chyba. Tento HTML kód vám může poskytnout užitečné informace o dané výjimce (výjimka se do metody předává jako argument).

Tyto metody samozřejmě reprezentují pouze malou část funkcionality, která je k dispozici prostřednictvím třídy `ControlDesigner`. Můžete překrýt mnohem více metod, čímž můžete konfigurovat různé aspekty chování v návrhovém režimu. V příštím oddílu se dozvíte, jak se vytvoří designér ovládacího prvku, který posílí podporu pro ovládací prvek `SuperSimpleRepeater`.

V následujícím příkladu vyvineme designéra ovládacího prvku, který vygeneruje rozumnou reprezentaci ovládacího prvku `SuperSimpleRepeater` vytvořeného v předchozí kapitole. Bez vlastního designéra ovládacího prvku je obsah prvku `SuperSimpleRepeater` v návrhovém režimu reprezentován prázdným řetězcem.

Prvním krokem při vytváření designéra je vybudovat třídu odvozenou ze třídy `ControlDesigner` ze jmenného prostoru `System.Web.UI.Design`, jak vidíte zde:

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

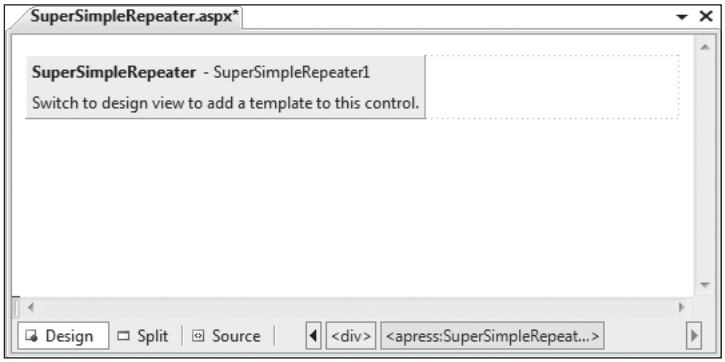
Pak je potřeba atributem `Designer` aplikovat designéra na ovládací prvek, jak to vidíte zde:

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

Při vytváření designéra ovládacího prvku je prvním krokem vytvořit metodu `GetEmptyDesignTimeHtml()`. Tato metoda jednoduše vrací statický úsek textu. `ControlDesigner` obsahuje pomocnou metodu `CreatePlaceHolderDesignTimeHtml()`, která vygeneruje šedý HTML rámeček se zprávou, již si sami specifikujete (stejně jako u ovládacího prvku `ListView`, který nemá žádné šablony). Touto metodou můžete zjednodušit realizační kód, jako zde:

```
protected override string GetEmptyDesignTimeHtml()
{
  string text = "Switch to design view to add a template to this control.";
  return CreatePlaceHolderDesignTimeHtml(text);
}
```

Prázdnou podobu ovládacího prvku `SuperSimpleRepeater` v návrhovém režimu vidíte na obrázku 28-8.



Obrázek 28-8. *Prázdná podoba ovládacího prvku* `SuperSimpleRepeater`.

POZNÁMKA *Pamatujte si, že ASP.NET nedokáže určit, kdy je váš ovládací prvek prázdný. V případě potřeby tedy musíte zavolat metodu* `GetEmptyDesignTimeHtml()`. *Jak vidíte na tomto příkladu – pokud není přítomna šablona, metoda* `GetDesignTimeHtml()` *zavolá metodu* `GetEmptyDesignTimeHtml()`.

Napsat metodu `GetErrorDesignTimeHtml()` je velmi snadné. Opět můžete použít metodu `CreatePlaceHolderDesignTimeHtml()`, tentokrát ovšem dodáte údaje o výjimce, ke které došlo.

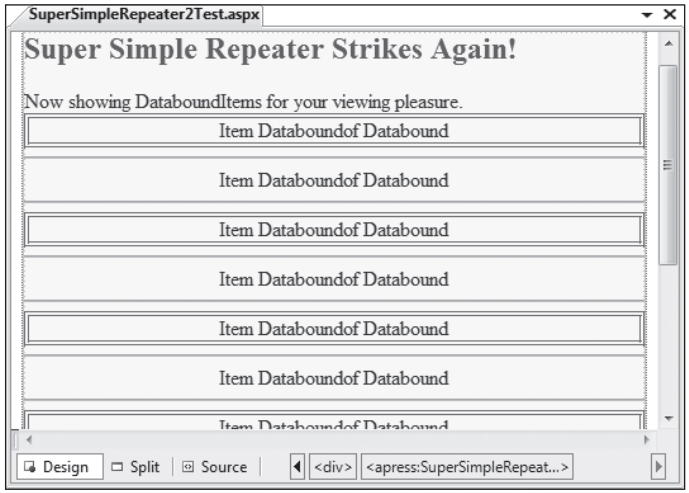
```
protected override string GetErrorDesignTimeHtml(Exception e)
{
  string text = string.Format("{0}{1}{2}{3}",
                "There was an error and the control can't be displayed.",
                "<br />", "Exception: ", e.Message);
  return CreatePlaceHolderDesignTimeHtml(text);
}
```

Posledním krokem je napsat metodu `GetDesignTimeHtml()`. Tento kód získává z vlastnosti `ControlDesigner.Component` aktuální instanci ovládacího prvku `SuperSimpleRepeater`. Pak zjistí, je-li přítomna šablona prvku. Pokud není, zobrazí se prázdný HTML kód. Pokud je šablona přítomna, ovládací prvek se sváže s daty, a teprve pak se zobrazí HTML kód pro návrhový režim.

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

```
      repeater.DataBind();
      return base.GetDesignTimeHtml();
    }
  }
  catch (Exception e)
  {
    return GetErrorDesignTimeHtml(e);
  }
}
```

Tím vznikne obrovsky zdokonalená reprezentace ovládacího prvku pro návrhový režim, kterou vidíte na obrázku 28-9. Velmi se podobá skutečnému vzhledu ovládacího prvku `SuperSimpleRepeater` při běhu.

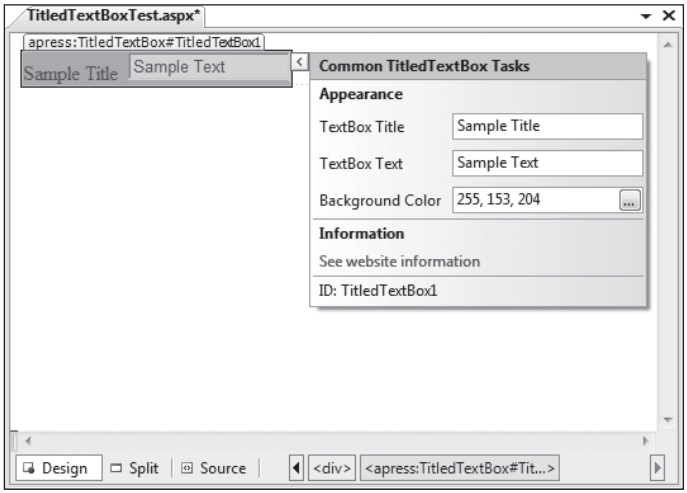


Obrázek 28-9. *Zdokonalená reprezentace pro návrhový režim.*

Inteligentní značky

Visual Studio obsahuje ještě jednu funkci pro vytváření bohatších prožitků v návrhovém režimu – tzv. inteligentní značky (smart tags). Jsou to pop-up okna, která se objevují vedle ovládacího prvku, pokud kliknete na malou šipku v rohu ovládacího prvku. Inteligentní značky jsou podobné nabídkám, protože poskytují seznam položek. Těmito položkami ovšem mohou být i příkazy (realizují se jako hypertextové odkazy) nebo nějaké jiné ovládací prvky, například zaškrtávací políčka, rozevírací seznamy a další. Mohou také obsahovat statický popisný text. Díky tomu se mohou inteligentní značky chovat jako zmenšené okno Properties.

Na obrázku 28-10 vidíte vlastní inteligentní značku, kterou společně vytvoříme v následujícím příkladu. Tato značka umožňuje vývojáři nastavit kombinaci vlastností `TitledTextBox`. Obsahuje dvě textová pole, do kterých můžete napsat text, hypertextový odkaz "See Website Information", jenž spouští prohlížeč a otevírá zadanou URL adresu, a statickou informaci s názvem ovládacího prvku.



Obrázek 28-10. *Vlastní inteligentní značka.*

Abyste mohli vytvořit takovou inteligentní značku, potřebujete tyto ingredience:

- **Kolekci objektů DesignerActionItem**. Každý `DesignerActionItem` reprezentuje jednu položku v inteligentní značce.
- **Třídu se seznamem akcí**. Tato třída hraje dvě role – konfiguruje kolekci instancí `DesignerActionItem` inteligentní značky, a pokud se provede nějaký příkaz nebo změna, vykoná odpovídající operaci na propojeném ovládacím prvku.
- **Designéra ovládacího prvku**. Připojí seznam akcí k ovládacímu prvku, takže v návrhovém režimu se objeví inteligentní značka.

V následujících oddílech vybudujete toto řešení krok za krokem.

Seznam akcí

V inteligentní značce můžete mít spoustu voleb. Aby mohly být přehledně uspořádané, je vhodné oddělit kód tím, že vytvoříte vlastní třídu, která zapouzdří seznam akcí. Tato vlastní třída by měla být odvozena z `DesignerActionList` (ze jmenného prostoru `System.ComponentModel.Design`).

Zde uvádíme příklad, který vytvoří seznam akcí určený pro `TitledTextBox`:

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

K seznamu akcí byste měli přidat jediný konstruktor, který vyžaduje shodný typ ovládacího prvku. Pak můžete v ovládacím prvku uložit referenci do členské proměnné. Není to však povinné, protože základní třída `ActionList` má vlastnost `Component`, která poskytuje přístup k vašemu ovládacímu prvku. Při tomto přístupu ovšem budete moci těžit ze silně typového přístupu k ovládacímu prvku.

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

```
  linkedControl = ctrl;
}
```

Než budete moci vybudovat inteligentní značku, musíte třídu se seznamem akcí vybavit požadovanými členy. Pro každý odkaz, který chcete přidat do značky (přes `DesignerActionMethodItem`), musíte vytvořit metodu. Pro každou vlastnost, kterou chcete přidat (přes `DesignerActionPropertyItem`), je potřeba vytvořit proceduru vlastnosti.

Inteligentní značka na obrázku 28-10 obsahuje sedm vlastních položek – dvě záhlaví kategorie, tři vlastnosti, jeden hypertextový odkaz akce a jeden úsek statického textu (u dolní strany značky).

V prvním kroku přidáte vlastnosti. Procedura `Get` vlastnosti potřebuje získat hodnotu vlastnosti z propojeného ovládacího prvku. Procedura `Set` vlastnosti potřebuje aplikovat novou hodnotu na propojený ovládací prvek. Je v tom ale jeden háček – novou hodnotu nemůžete nastavit přímo. Pokud byste tak učinili, dalším částem infrastruktury designéra by tato změna nebyla oznámena. Musíte to udělat jinak, metodou `PropertyDescriptor.SetValue()`. Abyste si to usnadnili, nadefinujte si ve třídě se seznamem akcí privátní pomocnou metodu, která získá `PropertyDescriptor` podle názvu dané vlastnosti:

```
private PropertyDescriptor GetPropertyByName(string propName)
{
  PropertyDescriptor prop;
  prop = TypeDescriptor.GetProperties(linkedControl)[propName];
  if (null == prop)
  {
    throw new ArgumentException("Matching property not found.", propName);
  }
  else
  {
    return prop;
  }
}
```

Nyní můžete vytvořit tři vlastnosti, které obalují vlastnosti v ovládacím prvku `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);
}
```

POZNÁMKA *Ne každá vlastnost se dá editovat přímo v inteligentní značce – vše závisí na datovém typu. Je-li daný datový typ asociován s* `UITypeEditor` *(pro grafickou editaci vlastnosti) nebo s* `TypeConverter` *(pro konverzi datového typu na řetězcovou reprezentaci a zpět), editace bude fungovat. Tyto ingredience sice má většina běžných typů dat, nicméně je nemají vaše vlastní objekty. Důsledkem je, že uvidíte řetězec určený pouze ke čtení, který se vygeneruje voláním* `ToString()` *na objektu. Více se dozvíte v příští kapitole, která se věnuje typovým konverzím.*

Dalším krokem je vybudovat funkcionalitu pro hypertextový odkaz "See Website Information". Pro tento účel tedy vytvořte metodu ve třídě se seznamem akcí. Zde uvádíme kód, který prostřednictvím třídy `Process` spustí výchozí prohlížeč uživatele:

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

Kolekce DesignerActionItem

Jednotlivé položky inteligentní značky jsou reprezentovány třídou `DesignerActionItem`. .NET Framework poskytuje čtyři základní třídy, které jsou odvozeny z `DesignerActionItem`. Tyto třídy jsou včetně svého popisu uvedeny v tabulce 28-3.

Tabulka 28-3. Třídy odvozené z DesignerActionItem.

Metoda	*Popis*
`DesignerActionMethodItem`	*Tato položka se realizuje jako hypertextový odkaz. Jestliže na ni kliknete, spustí akci tím, že zavolá metodu z vaší třídy* `DesignerActionList`.
`DesignerActionPropertyItem`	*Tato položka se realizuje jako editační ovládací prvek a používá logiku, která je podobná oknu Properties. Řetězce jsou textová pole, hodnoty výčtů tvoří rozvírací seznam, logické hodnoty jsou zaškrtávací políčka. Změníte-li hodnotu, změní se odpovídající podkladová vlastnost.*
`DesignerActionTextItem`	*Tato položka se realizuje jako statický text. Obvykle poskytuje doplňkové informace o ovládacím prvku. Nelze na ni kliknout.*
`DesignerActionHeaderItem`	*Tato položka je odvozena z* `DesignerActionTextItem`. *Je to statický text, který je stylovaný jako záhlaví. Použijete-li jednu nebo více položek záhlaví, můžete rozdělit inteligentní značku do několika oddělených kategorií a seskupit do nich vlastnosti. Nelze na ni kliknout.*

Abyste vytvořili naši inteligentní značku, musíte vybudovat `DesignerActionItemCollection`, která zkombinuje vaši skupinu objektů `DesignerActionItem`. V této kolekci je důležité pořadí, protože Visual Studio přidává objekty `DesignerActionItem` k inteligentní značce odshora dolů v pořadí, v jakém se objevují.

Seznam akcí vybudujete tak, že překryjete metodu `DesignerActionList.GetSortedActionItems()`, vytvoříte `DesignerActionItemCollection`, přidáte do ní jednotlivé `DesignerActionItem` a poté kolekci vrátíte. Podle toho, jak je inteligentní značka složitá, může se tento postup skládat z několika kroků.

Prvním krokem je vytvoření záhlaví, které rozdělí inteligentní značku na oddělené oblasti. K těmto kategoriím pak můžete přidat jiné položky. V našem příkladu použijeme dvě záhlaví:

```
public override DesignerActionItemCollection GetSortedActionItems()
{
  // Vytvoří se 8 položek.
  DesignerActionItemCollection items = new DesignerActionItemCollection();
  // Začneme vytvořením záhlaví.
  items.Add(new DesignerActionHeaderItem("Appearance"));
  items.Add(new DesignerActionHeaderItem("Information"));
  ...
```

Teď můžeme přidat vlastnosti. Uveďte název vlastnosti, pak název, který se má objevit v inteligentní značce. Poslední dvě položky představují kategorii, do které má být položka umístěna (odpovídá jedné z položek `DesignerActionHeaderItems`, již jste právě vytvořili), a popis (ten se objeví jako vysvětlivka, když na položce podržíte kurzor myši).

```
  ...
  // Přidáme položky, které obalují vlastnosti.
  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 pomocí reflexe připojí položku akce k vlastnosti ve třídě se seznamem akcí. Vy pouze dodáte název vlastnosti. Pokud přidáte do jedné kategorie více než jednu vlastnost, budou vlastnosti seřazeny v pořadí, v jakém je přidáváte. Pokud přidáte více než jedno záhlaví kategorie, kategorie budou rovněž seřazeny podle pořadí, v jakém se přidávaly.

Dalším krokem je vytvořit `DesignerActionMethodItem()`, která sváže položku inteligentní značky s metodou. V tomto případě specifikujete objekt, ve kterém je implementována metoda zpětného volání, název metody, název, jenž se má zobrazit v inteligentní značce, kategorie, kde se má položka objevit, a text vysvětlivky. Poslední parametr je logická hodnota. Má-li hodnotu `true`, přidá se položka nejenom do inteligentní značky, ale také do kontextové nabídky ovládacího prvku.

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

A nakonec můžete vytvořit nové objekty `DesignerActionTextItem` se statickým textem, který chcete zobrazit a vrátit kompletní kolekci položek, například takto:

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

Designér ovládacího prvku

Jakmile máte seznam akcí inteligentní značky v perfektním stavu, musíte ho ještě připojit k ovládacímu prvku. Dělá se to tak, že vytvoříte vlastního designéra ovládacího prvku a překryjete vlastnost `ActionLists`, aby vracela instanci vlastní třídy seznamu akcí. Ukážeme si to na následujícím designéru ovládacího prvku. Povšimněte si, že seznam akcí se nevytváří pokaždé, když se zavolá `ActionList`, protože v zájmu optimalizace výkonu se ukládá do cache v privátní členské proměnné.

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

Shrnutí

V této kapitole jsme probrali několik jednoduchých i složitých aspektů architektury .NET pro návrhový režim. Dozvěděli jste se, jak se konfiguruje způsob, jímž se zobrazují vlastnosti ovládacího prvku v okně Properties, a jak můžete převzít kontrolu nad serializací a analýzou (parsing) ovládacího prvku. Více informací o pokročilejších tématech, jako jsou například vlastní designéři ovládacích prvků, naleznete v dokumentaci MSDN.