31

Windows Forms

За последние несколько лет Web-ориентированные приложения стали чрезвычайно популярными. Возможность размещать всю логику приложений на централизованном сервере выглядит очень привлекательной с точки зрения администратора. Развертывание программного обеспечения, базирующегося на клиенте, очень трудно, особенно, если оно основано на COM-объектах. Недостаток Web-ориентированных приложений состоит в том, что они не могут предоставить достаточно богатых возможностей пользователю.

Платформа .NET Framework позволяет разрабатывать интеллектуальные клиентские приложения с богатыми возможностями, при этом избегая проблем с развертыванием и "DLL-адом", как это было раньше. Независимо от того, что будет выбрано — Windows Forms или Windows Presentation Foundation (см. главу 34) — разработка или развертывание клиентских приложений теперь не представляет особой сложности.

Windows Forms уже оказал влияние на разработки для Windows. Теперь, когда приложение находится на начальной стадии проектирования, принять решение о том, нужно ли строить Web-ориентированное приложение либо же полностью клиентское, стало немного труднее. Клиентские приложения Windows могут быть разработаны быстро и эффективно, при этом они предлагают пользователям гораздо более широкие возможности.

Windows Forms покажется вам знакомым, если у вас есть опыт разработки на Visual Basic. Вы создаете новые формы (также известные как окна или диалоги) в той же манере — перетаскивая и размещая элементы управления из панели инструментов на поверхность визуального дизайнера форм (Form Designer). Однако если ваш опыт в основном касается классического стиля языка C для Windows-программирования, где приходилось создавать конвейеры сообщений и отслеживать эти сообщения, или же если вы — программист, применяющий MFC, то в этом случае вы обнаружите, что и здесь при необходимости у вас есть возможность работать с низкоуровневыми деталями. Вы можете переопределить оконную процедуру (WndProc) и перехватывать сообщения, но в действительности вас удивит, что делать это придется нечасто.

В этой главе мы рассмотрим следующие аспекты Windows Forms:

- класс `Form`;
- иерархия классов Windows Forms;
- элементы управления и компоненты, являющиеся частью пространства имен `System.Windows.Forms`;
- меню и панели инструментов;
- создание элементов управления;
- создание пользовательских элементов управления.

Создание приложения Windows Forms

Первое, что необходимо сделать — создать приложение Windows Forms. Для примера создадим пустую форму и отобразим ее на экране. При разработке этого примера мы не будем использовать Visual Studio .NET. Наберем его в текстовом редакторе и соберем с помощью компилятора командной строки. Ниже показан код примера.

```
using System;
using System.Windows.Forms;
namespace NotepadForms
{
  public class MyForm : System.Windows.Forms.Form
  {
     public MyForm()
     {
     }
     [STAThread]
     static void Main()
     {
       Application.Run(new MyForm());
     }
  }
}
```

Когда мы скомпилируем и запустим этот пример, то получим маленькую пустую форму без заголовка. Никаких реальных функций, но это — Windows Forms.

В приведенном коде заслуживают внимания две вещи. Первая — тот факт, что при создании класса `MyForm` используется наследование. Следующая строка объявляет `MyForm` как наследника `System.Windows.Forms.Form`:

```
public class MyForm : System.Windows.Forms.Form
```

Класс `Form` — один из главных классов в пространстве имен `System.Windows.Forms`. Следующий фрагмент кода стоит рассмотреть более подробно:

```
[STAThread]
static void Main()
{
  Application.Run(new MyForm());
}
```

`Main` — точка входа по умолчанию в любое клиентское приложение на C#. Как правило, в более крупных приложениях метод `Main()` не будет находиться в классе формы, а скорее в классе, отвечающем за процесс запуска. В данном случае вы должны установить имя такого запускающего класса в диалоговом окне свойств проекта. Обратите внимание на атрибут `[STAThread]`. Он устанавливает модель многопоточности COM в STA (однопоточный апартамент). Модель многопоточности STA требуется для взаимодействия с COM и устанавливается по умолчанию в каждом проекте Windows Forms.

Метод `Application.Run()` отвечает за запуск стандартного цикла сообщений приложения. `Application.Run()` имеет три перегрузки.

Первая из них не принимает параметров; вторая принимает в качестве параметра объект `ApplicationContext`. В нашем примере объект `MyForm` становится главной формой приложения. Это означает, что когда форма закрывается, то приложение завершается. Используя класс `ApplicationContext`, можно в большей степени контролировать завершение главного цикла сообщений и выход из приложения.

Класс Application содержит в себе очень полезную функциональность. Он предоставляет группу статических методов и свойств для управления процессом запуска и останова приложения, а также обеспечивает доступ к сообщениям Windows, обрабатываемым приложением. В табл. 31.1 перечислены некоторые из этих наиболее полезных методов и свойств.

Таблица 31.1. Некоторые полезные методы и свойства класса Application

Метод/свойство	Описание
CommonAppDataPath	Путь к данным, общий для всех пользователей приложения. Обычно это БазовыйПуть\Название компании\Название продукта\Версия, где БазовыйПуть — C:\Documents and Settings\имя пользователя\ApplicationData. Если путь не существует, он будет создан.
ExecutablePath	Путь и имя исполняемого файла, запускающего приложение.
LocalUserAppDataPath	Подобно CommonAppDataPath, но с тем отличием, что поддерживается роуминг (перемещаемость).
MessageLoop	True или false — в зависимости от того, существует ли цикл сообщений в текущем потоке.
StartupPath	Подобно ExecutablePath, с тем отличием, что имя файла не возвращается.
AddMessageFilter	Используется для предварительной обработки сообщений. Объект, реализующий IMessageFilter, позволяет фильтровать сообщения в цикле или организовать специальную обработку, выполняемую перед тем, как сообщение попадет в цикл.
DoEvents	Аналогично оператору DoEvents языка Visual Basic. Позволяет обработать сообщения в очереди.
EnableVisualStyles	Обеспечивает визуальный стиль Windows XP для различных визуальных элементов приложения. Существуют две перегрузки, принимающие информацию манифеста. Одна работает с потоком манифеста, вторая — принимает полное имя и путь файла манифеста.
Exit и ExitThread	Exit завершает текущий работающий цикл сообщений и вызывает выход из приложения. ExitThread завершает цикл сообщений и закрывает все окна текущего потока.

А теперь как будет выглядеть это приложение, если его сгенерировать в Visual Studio 2008? Первое, что следует отметить — будет создано два файла. Причина в том, что Visual Studio 2008 использует возможность частичных (partial) классов и выделяет весь код, сгенерированный визуальным дизайнером, в отдельный файл. Если используется имя по умолчанию — Form1, то эти два файла будет называться Form1.cs и Form1.Designer.cs. Если только у вас не включена опция Show All Files (Показать все файлы) в меню Project (Проект), то вы не увидите в проводнике Solution Explorer файла Form1.Designer.cs. Ниже показан код этих двух файлов, сгенерированных Visual Studio. Сначала — Form1.cs:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
```

```
using System.Windows.Forms;

namespace VisualStudioForm
{
     public partial class Form1 : Form
     {
       public Form1()
       {
          InitializeComponent();
       }
     }
}
```

Здесь мы видим только операторы `using` и простой конструктор. А вот код `Form1.Designer.cs`:

```
namespace VisualStudioForm
{
  partial class Form1
  {
     /// <summary>
     /// Required designer variable.
     /// </summary>
     private System.ComponentModel.IContainer components = null;
     /// <summary>
     /// Clean up any resources being used.
     /// </summary>
     /// < param name="disposing" > true if managed resources should be disposed;
     otherwise, false. < /param >
     protected override void Dispose(bool disposing)
     {
       if (disposing && (components != null))
       {
            components.Dispose();
       }
       base.Dispose(disposing);
     }
     #region Windows Form Designer generated code
     /// <summary>
     /// Required method for Designer support - do not modify
     /// the contents of this method with the code editor.
     /// </summary>
     private void InitializeComponent()
     {
       this.components = new System.ComponentModel.Container();
       this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
       this.Text = "Form1";
     }
     #endregion
  }
}
```

Файл, сгенерированный дизайнером форм, редко подвергается ручному редактированию. Единственным исключением может быть случай, когда необходима специальная обработка в методе `Dispose()`. Метод `InitializeComponent` мы обсудим позднее в этой главе.

Если взглянуть на этот код примера приложения в целом, то мы увидим, что он намного длиннее, чем простой пример командной строки. Здесь перед началом класса присутствует несколько операторов `using`, и большинство из них в данном примере не нужны. Однако их присутствие ничем не мешает. Класс `Form1` наследуется от

System.Windows.Forms.Form, как и в предыдущем, введенном в Notepad примере, но в этой точке начинаются расхождения. Во-первых, в файле Form1.Designer появляется строка:

```
private System.ComponentModel.IContainer components = null;
```

В данном примере эта строка кода ничего не делает. Но, добавляя компонент в форму, вы можете также добавить его в объект components, который представляет собой контейнер. Причина добавления этого контейнера — в необходимости правильной обработки уничтожения формы. Класс формы поддерживает интерфейс IDisposable, потому что он реализован в классе Component. Когда компонент добавляется в контейнер, то этот контейнер должен позаботиться о корректном уничтожении своего содержимого при закрытии формы. Это можно увидеть в методе Dispose нашего примера:

```
protected override void Dispose(bool disposing)
{
  if (disposing && (components != null))
  {
       components.Dispose();
  }
  base.Dispose(disposing);
}
```

Здесь мы видим, что когда вызывается метод Dispose формы, то также вызывается метод Dispose объекта components, поскольку он содержит в себе другие компоненты, которые также должны быть корректно удалены.

Конструктор класса Form1, находящийся в файле Form1.cs, выглядит так:

```
public Form1()
{
  InitializeComponent();
}
```

Обратите внимание на вызов InitializeComponent().

Метод InitializeComponent() находится в файле Form1.Designer.cs и делает то, что следует из его наименования — инициализирует все элементы управления, которые могут быть добавлены к форме. Он также инициализирует свойства формы.

В нашем примере метод InitializeComponent() выглядит следующим образом:

```
private void InitializeComponent()
{
  this.components = new System.ComponentModel.Container();
  this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
  this.Text = "Form1";
}
```

Как видите, здесь присутствует лишь базовый код инициализации. Этот метод связан с визуальным дизайнером Visual Studio. Когда в форму вносятся изменения в дизайнере, они отражаются на InitializeComponent(). Если вы вносите любые изменения в InitializeComponent(), то следующий раз после того, как что-то будет изменено в дизайнере, эти ручные изменения будут утеряны. InitializeComponent() повторно генерируется после каждого изменения дизайна формы. Если возникает необходимость добавить некоторый дополнительный код для формы или элементов управления и компонентов формы, это должно быть сделано после вызова InitializeComponent(). Этот метод также отвечает за создание экземпляров элементов управления, поэтому любой вызов, ссылающийся на них, выполненный до InitializeComponent(), завершится возбуждением исключения нулевой ссылки.

Чтобы добавить элемент управления или компонент в форму, нажмите комбинацию клавиш <Ctrl+Alt+X> или выберите пункт меню View⇨Toolbox (Вид⇨Панель инструментов) в среде Visual Studio .NET. Щелкните правой кнопкой мыши на `Form1.cs` в проводнике Solution Explorer и в появившемся контекстном меню выберите пункт View Designer (Показать дизайнер). Выберите элемент управления `Button` и перетащите на поверхность формы в визуальном дизайнере. Можно также дважды щелкнуть на выбранном элементе управления, и он будет добавлен в форму. То же самое следует проделать с элементом `TextBox`.

Теперь, после добавления этих двух элементов управления на форму, метод `InitializeComponent()` расширяется и содержит такой код:

```
private void InitializeComponent()
{
  this.button1 = new System.Windows.Forms.Button();
  this.textBox1 = new System.Windows.Forms.TextBox();
  this.SuspendLayout();
  //
  // button1
  //
  this.button1.Location = new System.Drawing.Point(77, 137);
  this.button1.Name = "button1";
  this.button1.Size = new System.Drawing.Size(75, 23);
  this.button1.TabIndex = 0;
  this.button1.Text = "button1";
  this.button1.UseVisualStyleBackColor = true;
  //
  // textBox1
  //
  this.textBox1.Location = new System.Drawing.Point(67, 75);
  this.textBox1.Name = "textBox1";
  this.textBox1.Size = new System.Drawing.Size(100, 20);
  this.textBox1.TabIndex = 1;
  //
  // Form1
  //
  this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
  this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
  this.ClientSize = new System.Drawing.Size(284, 264);
  this.Controls.Add(this.textBox1);
  this.Controls.Add(this.button1);
  this.Name = "Form1";
  this.Text = "Form1";
  this.ResumeLayout(false);
  this.PerformLayout();
}
```

Если посмотреть на первые три строки кода этого метода, мы увидим в них создание экземпляров элементов управления `Button` и `TextBox`. Обратите внимание на имена, присвоенные им — `textBox1` и `button1`. По умолчанию дизайнер в качестве имен использует имя класса элемента управления, дополненное целым числом. Когда вы добавляете следующую кнопку, дизайнер называет ее `button2` и т.д. Следующая строка — часть пары `SuspendLayout/ResumeLayout`. Метод `SuspendLayout()` временно приостанавливает события размещения, которые имеют место при первоначальной инициализации элемента управления. В конце метода вызывается `ResumeLayout()`, чтобы вернуть все в норму. В сложной форме с множеством элементов управления метод `InitializeComponent()` может стать достаточно большим.

Чтобы изменить значения свойств элемента управления, нужно либо нажать <F4>, либо выбрать пункт меню View⇨Properties Window (Вид⇨Окно свойств). Это окно позволяет модифицировать большинство свойств элемента управления или компонента. При внесении изменений в окне свойств метод `InitializeComponent()` автоматически переписывается с тем, чтобы отразить новые значения свойств. Например, изменив свойство `Text` на `My  Button` в окне свойств, получим следующий код в `InitializeComponent()`:

```
//
// button1
//
this.button1.Location = new System.Drawing.Point(77, 137);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(75, 23);
this.button1.TabIndex = 0;
this.button1.Text = "My Button";
this.button1.UseVisualStyleBackColor = true;
```

Даже если вы предпочитаете использовать какой-то редактор, отличный от Visual Studio .NET, то наверняка захотите включать функции вроде `InitializeComponent()` в свои проекты. Сохранение всего кода инициализации в одном месте обеспечит возможность его вызова из каждого конструктора.

Иерархия классов

Важность понимания иерархии становится очевидной в процессе проектирования и конструирования пользовательских элементов управления. Если такой элемент управления унаследован от конкретного библиотечного элемента управления, например, когда создается текстовое поле с некоторыми дополнительными методами и свойствами, то имеет смысл унаследовать его от обычного текстового поля и затем переопределить и добавить необходимые методы. Однако если приходится создавать элемент управления, который не соответствует ни одному из существующих в .NET Framework, то его придется унаследовать от одного из базовых классов: `Control` или `ScrollableControl`, если нужны возможности прокрутки, либо `ContainerControl`, если он должен служить контейнером для других элементов управления.

Остальная часть этой главы посвящена изучению большинства из этих классов — как они работают вместе, и как их можно использовать для построения профессионально выглядящих клиентских приложений.

Класс `Control`

Пространство имен `System.Windows.Forms` включает один класс, который является базовым почти для всех элементов управления и форм — `System.Windows.Forms.Control`. Он реализует основную функциональность для создания экранов, которые видит пользователь. Класс `Control` унаследован от `System.ComponentModel.Component`. Класс `Component` обеспечивает классу `Control` инфраструктуру, необходимую для того, чтобы его можно было перетаскивать и помещать на поле дизайнера, а также, чтобы он мог включать в себя другие элементы управления. Класс `Control` предлагает огромный объем функциональности классам, наследуемым от него. Этот список слишком большой, чтобы приводить его здесь, поэтому в данном разделе мы рассмотрим только наиболее важные возможности, предоставляемые классом `Control`. Позднее в этой главе, когда мы будем рассматривать специфические элементы управления, основанные на `Control`, мы увидим эти методы и свойства в коде примеров. Следующие

подразделы группируют методы и свойства по их функциональности, поэтому взаимосвязанные элементы могут быть рассмотрены вместе.

Размер и местоположение

Размер и местоположение элементов управления определяются свойствами `Height`, `Width`, `Top`, `Bottom`, `Left` и `Right`, вместе с дополняющими их `Size` и `Location`. Отличие состоит в том, что `Height`, `Width`, `Top`, `Bottom`, `Left` и `Right` принимают одно целое значение. `Size` принимает значение структуры `Size`, а `Location` — значение структуры `Point`. Структуры `Size` и `Point` включают в себя координаты X, Y. `Point` обычно описывает местоположение, а `Size` — высоту и ширину объекта. `Size` и `Point` определены в пространстве имен `System.Drawing`. Обе структуры очень похожи в том, что представляют пары координат X, Y, но, кроме того — переопределенные операции, упрощающие сравнения и преобразования. Вы можете, например, складывать вместе две структуры `Size`. В случае структуры `Point` операция сложения переопределена таким образом, что можно прибавить к `Point` структуру `Size` и получить в результате `Point`. Это дает эффект прибавления расстояния к местоположению, чтобы получить новое местоположение, что очень удобно для динамического создания форм и элементов управления.

Свойство `Bounds` возвращает объект `Rectangle`, представляющий экранную область, занятую элементом управления. Эта область включает полосы прокрутки и заголовка. `Rectangle` также относится к пространству имен `System.Drawing`. Свойство `ClientSize` — структура `Size`, представляющая клиентскую область элемента управления за вычетом полос прокрутки и заголовка.

Методы `PointToClient` и `PointToScreen` — удобные методы преобразования, которые принимают `Point` и возвращают `Point`. Метод `PointToClient` принимает структуру `Point`, представляющую экранные координаты, и транслирует их в координаты текущего клиентского объекта. Это удобно для операций перетаскивания. Метод `PointToScreen` выполняет обратную операцию — принимает координаты в клиентском объекте и транслирует их в экранные координаты.

Методы `RectangleToScreen` и `ScreenToRectangle` выполняют те же операции, но со структурами `Rectangle` вместо `Point`.

Свойство `Dock` определяет, к какой грани родительского элемента управления должен пристыковываться данный элемент. Перечисление `DockStyle` задает возможные значения этого свойства. Они могут быть такими: `Top`, `Bottom`, `Right`, `Left`, `Fill` и `None`. Значение `Fill` устанавливает размер данного элемента управления равным размеру родительского.

Свойство `Anchor` (якорь) прикрепляет грань данного элемента управления к грани родительского элемента управления. Это отличается от стыковки (docking) тем, что не устанавливает грань дочернего элемента управления точно на грань родительского, а просто выдерживает постоянное расстояние между ними. Например, если якорь правой грани элемента управления установлен на правую грань родительского элемента, и если родитель изменяет размер, то правая грань данного элемента сохраняет постоянную дистанцию от правой грани родителя — т.е. он изменяет размер вместе с родителем. Свойство `Anchor` принимает значения из перечисления `AnchorStyle`, а именно: `Top`, `Bottom`, `Left`, `Right` и `None`. Устанавливая эти значения, можно заставить элемент управления изменять свой размер динамически вместе с родителем. Таким образом, кнопки и текстовые поля не будут усечены или скрыты при изменении размеров формы пользователем.

Свойства `Dock` и `Anchor` применяются в сочетании с компоновками элементов управления `Flow` и `Table` (о которых мы поговорим позднее в этой главе) и позволя-

ют создавать очень сложные пользовательские окна. Изменение размеров окна может оказаться достаточно непростой задачей для сложных форм с множеством элементов управления. Эти инструменты существенно облегчают задачу.

Внешний вид

Свойства, имеющие отношение к внешнему виду элемента управления — это `BackColor` и `ForeColor`, которые принимают объект `System.Drawing.Color` в качестве значения. Свойство `BackGroundImage` принимает объект графического образа как значение. Класс `System.Drawing.Image` — абстрактный класс, служащий в качестве базового для классов `Bitmap` и `Metafile`. Свойство `BackgroundImageLayout` использует перечисление `ImageLayout` для определения способа отображения графического образа в элементе управления. Допустимые значения таковы: `Center`, `Tile`, `Stretch`, `Zoom` или `None`.

Свойства `Font` и `Text` работают с надписями. Чтобы изменить `Font`, необходимо создать объект `Font`. При создании этого объекта указывается имя, стиль и размер шрифта.

Взаимодействие с пользователем

Взаимодействие с пользователем лучше всего описывается серией событий, которые генерирует элемент управления и на которые он реагирует. Некоторые из наиболее часто используемых событий: `Click`, `DoubleClick`, `KeyDown`, `KeyPress`, `Validating` и `Paint`.

События, связанные с мышью — `Click`, `DoubleClick`, `MouseDown`, `MouseUp`, `MouseEnter`, `MouseLeave` и `MouseHover` — описывают взаимодействие мыши и экранного элемента управления. Если вы обрабатываете оба события — `Click` и `DoubleClick` — то всякий раз, когда перехватывается событие `DoubleClick`, также возбуждается и событие `Click`. Это может привести к нежелательным последствиям при неправильной обработке. К тому же и `Click`, и `DoubleClick` принимают в качестве аргумента `EventArgs`, в то время как события `MouseDown` и `MouseUp` принимают `MouseEventArgs`. Структура `MouseEventArgs` содержит несколько частей полезной информации — например, о кнопке, на которой был выполнен щелчок, количестве щелчков на кнопке, количестве щелчков колесика мыши (при условии его наличия), текущих координатах X и Y указателя мыши. Если нужен доступ к любой подобной информации, то вместо событий `Click` или `DoubleClick` потребуется обрабатывать события `MouseDown` и `MouseUp`.

События клавиатуры работают подобным образом. Объем необходимой информации определяет выбор обрабатываемых событий. Для простейших случаев событие `KeyPress` принимает `KeyPressEventArgs`. Эта структура включает `KeyChar`, представляющий символ нажатой клавиши. Свойство `Handled` используется для определения того, было ли событие обработано. Установив значение `Handled` в `true`, можно добиться того, что событие не будет передано операционной системе для совершения стандартной обработки. Если необходима дополнительная информация о нажатой клавише, то больше подойдут события `KeyDown` или `KeyUp`. Оба принимают структуру `KeyEventArgs`. Свойства `KeyEventArgs` включают признак одновременного состояния клавиш <Ctrl>, <Alt> или <Shift>. Свойство `KeyCode` возвращает значение типа перечисления `Keys`, идентифицирующее нажатую клавишу. В отличие от свойства `KeyPressEventArgs.KeyChar`, свойство `KeyCode` сообщает о каждой клавише клавиатуры, а не только о буквенно-цифровых клавишах. Свойство `KeyData` возвращает значение типа `Keys`, а также устанавливает модификатор. Значение модификатора со-

провождает значение клавиши, объединяясь с ним двоичной логической операцией "ИЛИ". Таким образом, можно получить информацию о том, была ли одновременно нажата клавиша <Shift> или <Ctrl>. Свойство `KeyValue` — целое значение из перечисления `Keys`. Свойство `Modifiers` содержит значение типа `Keys`, представляющее нажатые модифицирующие клавиши. Если было нажато более одной такой клавиши, их значения объединяются операцией "ИЛИ". События клавиш поступают в следующем порядке:

1. `KeyDown`
2. `KeyPress`
3. `KeyUp`

События `Validating`, `Validated`, `Enter`, `Leave`, `GotFocus` и `LostFocus` имеют отношение к получению фокуса элементами управления (т.е. когда становятся активными) и утере его. Это случается, когда пользователь нажатием клавиши <Tab> переходит к данному элементу управления либо выбирает его мышью. Может показаться, что события `Enter`, `Leave`, `GotFocus` и `LostFocus` очень похожи в том, что они делают. События `GotFocus` и `LostFocus` относятся к низкоуровневым, и связаны с событиями Windows `WM_SETFOCUS` и `WM_KILLFOCUS`. Обычно когда возможно, лучше использовать события `Enter` и `Leave`. События `Validating` и `Validated` возбуждаются при проверке данных в элементе управления. Эти события принимают аргумент `CancelEventArgs`. С его помощью можно отменить последующие события, установив свойство `Cancel` в `true`. Если вы разрабатываете собственный проверочный код, и проверка завершается неудачно, то в этом случае можно установить `Cancel` в `true` — тогда элемент управления не утратит фокус. `Validating` происходит во время проверки, а `Validated` — после нее. Порядок возникновения событий следующий:

1. `Enter`
2. `GotFocus`
3. `Leave`
4. `Validating`
5. `Validated`
6. `LostFocus`

Понимание последовательности этих событий важно, чтобы избежать рекурсивных ситуаций. Например, попытка установить фокус элемента управления внутри обработчика события `LostFocus` создает ситуацию взаимоблокировки в цикле событий, и приложение перестает реагировать на внешние воздействия.

Функциональность Windows

Пространство имен `System.Windows.Forms` — одно из немногих, полагающихся на функциональность операционной системы Windows. Класс `Control` — хороший тому пример. Если выполнить дизассемблирование `System.Windows.Forms.dll`, то можно увидеть список ссылок на класс `UnsafeNativeMethods`. Среда .NET Framework использует этот класс как оболочку для всех стандартных вызовов Win32 API. Благодаря возможности взаимодействия с Win32 API, внешний вид и поведение стандартного приложения Windows можно обеспечить средствами пространства имен `System.Windows.Forms`.

Функциональность, которая поддерживает взаимодействие с Windows, включает свойства `Handle` и `IsHandleCreated`. Свойство `Handle` возвращает `IntPtr`, содержащий `HWND` (дескриптор окна) элемента управления. Дескриптор окна — это `HWND`, уни-

кально идентифицирующий окно. Элемент управления может рассматриваться как окно, поэтому у него есть соответствующий HWND. Свойство Handle можно использовать для обращения к любым вызовам Win32 API.

Для получения доступа к внутренним сообщениям Windows можно переопределить метод WndProc. Метод WndProc принимает в качестве параметра объект Message. Этот объект представляет собой просто оболочку для сообщения окна. Он содержит свойства HWnd, LParam, WParam, Msg и Result. Если нужно, чтобы сообщение было обработано системой, его потребуется передать на обработку базовому методу base.WndProc(msg). Если же нужно обработать его в вашем приложении специальным образом, то передавать его этому методу не следует.

Прочая функциональность

Некоторые вещи, которые несколько сложнее классифицировать — это возможности привязки данных. Свойство BindingContext возвращает объект BindingManagerBase.

Коллекция DataBindings поддерживает ControlBindingsCollection, которая представляет коллекцию привязанных объектов элемента управления. Привязка данных обсуждается в главе 32.

CompanyName, ProductName и ProductVersions представляют данные о происхождении элемента управления и его текущей версии.

Метод Invalidate позволяет объявить видимую область элемента управления недействительной, чтобы инициировать ее перерисовку. Это можно сделать с целым элементом управления либо с его частью. После этого сообщение перерисовки отправляется методу WndProc этого элемента управления. Можно в то же время объявить недействительным любой дочерний элемент управления.

Класс Control состоит из десятков других свойств, методов и событий. Приведенный список представляет лишь некоторые из наиболее часто используемых, и предназначен для того, чтобы дать вам представление о доступной функциональности.

Стандартные элементы управления и компоненты

В предыдущем разделе описаны некоторые общие методы и свойства элементов управления. Здесь же мы рассмотрим различные стандартные элементы управления, поставляемые в составе .NET Framework, и объясним, какую дополнительную функциональность они предлагают. Среди примеров, которые доступны на прилагаемом компакт-диске, есть пример приложения по имени FormsSample. Это MDI-приложение (обсуждается позже), включающее форму frmControls, которая содержит многие элементы управления с базовой функциональностью. На рис. 31.1 показан ее внешний вид.

Button

Класс Button представляет простую командную кнопку и наследуется от ButtonBase. Чаще всего требует написания кода обработки события Click. Следующий фрагмент кода реализует обработчик события Click. Когда выполняется щелчок на кнопке, появляется окно сообщения, отображающее имя кнопки.

```
private void btnTest_Click(object sender, System.EventArgs e)
{
  MessageBox.Show("Выполнен щелчок на " + ((Button)sender).Name + ".");
}
```

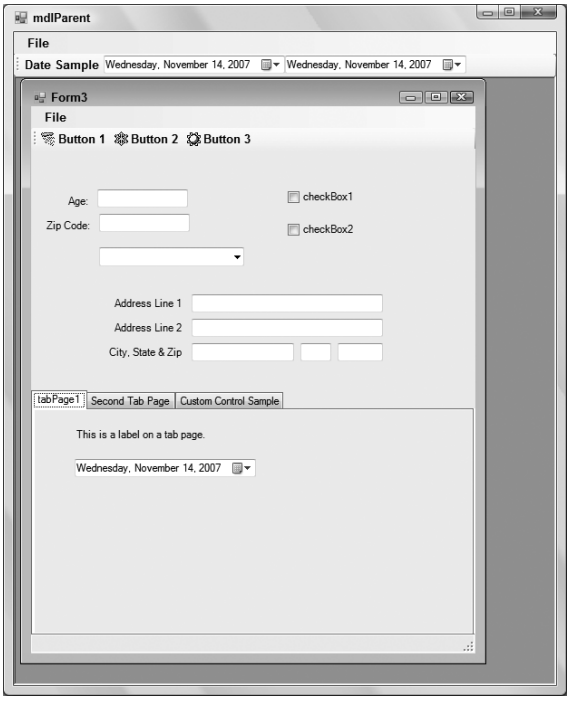

Рис. 31.1. Внешний вид формы `frmControls` приложения `FormsSample`

С помощью метода `PerformClick` можно эмулировать событие `Click` кнопки без необходимости действительного выполнения щелчка пользователем. Метод `NotifyDefault` принимает в параметре значение булевского типа и сообщает кнопке, чтобы она отобразила себя как кнопку по умолчанию. Обычно кнопка по умолчанию на форме имеет слегка утолщенную рамку.

Чтобы идентифицировать кнопку как кнопку по умолчанию, потребуется установить свойство `AcceptButton` формы равным ссылке на эту кнопку. После этого, если пользователь нажмет клавишу <Enter>, сгенерируется событие `Click` этой кнопки по умолчанию. На рис. 31.2 кнопка с меткой Default (По умолчанию) является кнопкой по умолчанию (обратите внимание на темную рамку).

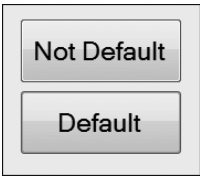

Рис. 31.2. Кнопка с меткой Default является кнопкой по умолчанию

Кнопки могут содержать на своей поверхности как текст, так и графическое изображение. Изображения доступны для кнопок через объект `ImageList` или свойство `Image`. Объект `ImageList` представляет собой именно то, что можно предположить по его названию: список изображений, который управляется компонентом, помещенным на форму. Позднее в этой главе мы рассмотрим его более детально.

Как `Text`, так и `Image` имеют свойство `Align`, предназначенное для выравнивания текста или изображения на поверхности кнопки. Свойство `Align` принимает значения типа перечисления `ContentAlignment`. Текст или изображение могут быть выровнены в комбинации по левой или правой границе кнопки либо по верхней или нижней границе.

CheckBox

Элемент управления CheckBox (флажок) также унаследован от ButtonBase и применяется для принятия команды пользователя с двумя или тремя состояниями. Если свойство ThreeState установлено в true, то свойство CheckState элемента CheckBox может принимать одно из следующих трех перечислимых значений:

Checked	Элемент CheckBox отмечен.
Unchecked	Элемент CheckBox не отмечен.
Indeterminate	В этом состоянии элемент CheckBox не доступен.

Состояние Indeterminate может быть установлено только программно, а не пользователем. Это удобно, если вы хотите сообщить пользователю, что опция не была установлена. Для получения текущего состояния в виде булевского значения можно обратиться к свойству Checked.

События CheckedChanged и CheckStateChanged возникают, когда изменяется свойство CheckState или Checked. Перехват этих событий может пригодиться для установки других значений на основе нового состояния CheckBox. В классе формы frmControls событие CheckedChanged для нескольких элементов CheckBox обрабатывается следующим методом:

```
private void checkBoxChanged(object sender, EventArgs e)
{
  CheckBox checkBox = (CheckBox)sender;
  MessageBox.Show("Новое значение " + checkBox.Name + " равно " +
     checkBox.Checked.ToString());
}
```

При изменении состояния каждого из этих элементов отображается окно сообщения с именем элемента CheckBox и его новым состоянием.

RadioButton

Последний элемент управления, унаследованный от ButtonBase — это RadioButton (переключатель). Переключатели обычно используются в составе групп. Иногда называемые кнопками выбора (option buttons), переключатели дают возможность пользователю выбирать одну из нескольких опций. Когда вы используете множество элементов управления RadioButton в одном контейнере, выбранным может быть только один из них. Поэтому если у вас есть три опции, например, Red, Green и Blue, и если выбрана опция Red, а пользователь щелкает на Blue, то Red автоматически отключается.

Свойство Appearance принимает значение из перечисления Appearance. Оно может быть либо Button, либо Normal. Когда выбирается Normal, то переключатель выглядит как маленький кружок с меткой рядом с ним. Выбор его заполняет кружок, выбор другого переключателя из той же группы отменяет выбор текущего выбранного переключателя и делает его кружок пустым. При установке значения Appearance равным Button переключатель выглядит подобно стандартной кнопке, но работает подобно переключателю — выбранная кнопка нажата, не выбранная — отпущена.

Свойство CheckedAlign определяет, где находится кружок по отношению к тексту метки. Он может быть над текстом, под ним, справа или слева.

Событие CheckedChanged возникает всякий раз, когда значение свойства Checked изменяется. Подобным образом можно выполнить другие действия на основе нового значения элемента управления.

ComboBox, ListBox и CheckedListBox

ComboBox, ListBox и CheckedListBox — все унаследованы от класса ListControl. Этот класс определяет некоторую базовую функциональность управления списками. Самое главное в использовании списочных элементов управления — это добавление и выбор элементов списка. То, какой список нужно применять, в основном определяется тем, как его предполагается использовать, и типом данных, которые в нем должны содержаться. Если необходимо иметь возможность множественного выбора, или пользователю нужно видеть в любой момент несколько позиций списка, то лучше всего подойдут ListBox или CheckListBox. Если же в списке может быть выбран только один элемент за раз, то больше подойдет ComboBox.

Прежде чем списком можно будет пользоваться, к нему нужно добавить данные. Это делается добавлением объектов в ListBox.ObjectCollection. Эта коллекция представлена свойством списка Items. Поскольку коллекция сохраняет объекты, любой корректный тип .NET может быть добавлен в список. Для того чтобы идентифицировать элементы, необходимо установить два важных свойства. Первое из них — DisplayMember. Эта установка сообщает ListControl, какое свойство вашего объекта должно быть отображено в списке. Второе — ValueMember, оно указывает свойство вашего объекта, которое нужно возвратить в качестве его значения. Если в список добавляются строки, то по умолчанию они и используются для обоих этих свойств. Форма frmLists в примере приложения показывает, как и объекты, и строки (которые, разумеется, тоже представляют собой объекты) могут быть загружены в окно списка. В примере в качестве данных списка применяются объекты Vendor. Объект Vendor включает в себя только два свойства: Name и PhoneNo. Свойство DisplayMember установлено в свойство Name. Это заставляет списочный элемент управления отображать значения свойств Name содержащихся в нем объектов.

Существуют два способа доступа к данным в списочном элементе управления, как показано в следующем примере кода. Список загружается объектами Vendor. Устанавливаются свойства DisplayMember и ValueMember. Этот код можно найти в классе формы frmList приложения-примера.

В начале идет метод LoadList. Этот метод загружает список объектами Vendor или простыми строками, содержащими имя поставщика. Кнопка-переключатель используется для выбора того, что именно должно загружаться в список.

```
private void LoadList(Control ctrlToLoad)
{
  ListBox tmpCtrl = null;
  if (ctrlToLoad is ListBox)
     tmpCtrl = (ListBox)ctrlToLoad;
  tmpCtrl.Items.Clear();
  tmpCtrl.DataSource = null;
  if (radioButton1.Checked)
  {
     // загрузить объекты
     tmpCtrl.Items.Add(new Vendor("XYZ Company", "555-555-1234"));
     tmpCtrl.Items.Add(new Vendor("ABC Company", "555-555-2345"));
     tmpCtrl.Items.Add(new Vendor("Other Company", "555-555-3456"));
     tmpCtrl.Items.Add(new Vendor("Another Company", "555-555-4567"));
     tmpCtrl.Items.Add(new Vendor("More Company", "555-555-6789"));
     tmpCtrl.Items.Add(new Vendor("Last Company", "555-555-7890"));
     tmpCtrl.DisplayMember = "Name";
  }
  else
  {
     tmpCtrl.Items.Clear();
```

```
        tmpCtrl.Items.Add("XYZ Company");
        tmpCtrl.Items.Add("ABC Company");
        tmpCtrl.Items.Add("Other Company");
        tmpCtrl.Items.Add("Another Company");
        tmpCtrl.Items.Add("More Company");
        tmpCtrl.Items.Add("Last Company");
    }
}
```

После того, как данные загружены в список, для их получения можно использовать свойства `SelectedItem` и `SelectedIndex`. Свойство `SelectedItem` возвращает текущий выбранный объект. Если список разрешает множественный выбор, нет гарантии того, какой именно элемент будет возвращен. В этом случае должна использоваться коллекция `SelectObject`. Она содержит список всех текущих выбранных элементов списка.

Если нужно получить элемент по определенному индексу, то свойство `Items` может быть использовано для доступа к `ListBox.ObjectCollection`. Поскольку это стандартный класс коллекции .NET, элементы коллекции могут быть получены так же, как элементы любого класса коллекции.

Если для наполнения списка используется `DataBinding` (привязка данных), то свойство `SelectedValue` вернет то свойство выбранного объекта, которое было указано через свойство `ValueMember`. Если `ValueMember` установлено в `Phone`, то `SelectedValue` вернет значение `Phone` выбранного в данный момент элемента списка. Для того чтобы можно было использовать `ValueMember` и `SelectedValue`, список должен быть загружен способом `DataSource`. `ArrayList` или любая другая основанная на `IList` коллекция должна быть сначала заполнена объектами, потом готовый список может быть присвоен свойству `DataSource`. Следующий короткий пример демонстрирует сказанное.

```
listBox1.DataSource = null;
System.Collections.ArrayList lst = new System.Collections.ArrayList();
lst.Add(new Vendor("XYZ Company", "555-555-1234"));
lst.Add(new Vendor("ABC Company", "555-555-2345"));
lst.Add(new Vendor("Other Company", "555-555-3456"));
lst.Add(new Vendor("Another Company", "555-555-4567"));
lst.Add(new Vendor("More Company", "555-555-6789"));
lst.Add(new Vendor("Last Company", "555-555-7890"));
listBox1.Items.Clear();
listBox1.DataSource = lst;
listBox1.DisplayMember = "Name";
listBox1.ValueMember = "Phone";
```

Использование `SelectedValue` без `DataBinding` приведет к исключению `NullException`.

В следующем фрагменте кода показан синтаксис доступа к данным списка.

```
// obj установлен в ссылку на выбранный объект Vendor
obj = listBox1.SelectedItem;

// obj установлен в ссылку на объект Vendor по индексу 3 (4-й по счету объект)
obj = listBox.Items[3];

// obj установлен в значение свойства Phone выбранного объекта поставщика
// в этом примере предполагается, что для наполнения списка
// использовалась привязка данных
listBox1.ValuesMember = "Phone";
obj = listBox1.SelectValue;
```

Что необходимо помнить — все эти методы возвращают тип `object`. Чтобы использовать реальное значение возвращенного объекта, нужно выполнить приведение к соответствующему типу.

Свойство `Items` класса `ComboBox` возвращает `ComboBox.ObjectCollection`. `ComboBox` представляет собой комбинацию редактируемого текстового поля и окна списка. Стиль `ComboBox` устанавливается передачей значения типа перечисления `DropDownStyle` свойству `DropDownStyle`. Возможные значения `DropDownStyle` перечислены в табл. 31.2.

Таблица 31.2. Возможные значения `DropDownStyle`

Значение	Описание
`DropDown`	Текстовую часть можно редактировать — пользователь может вводить значение. Он также может щелкнуть на кнопке со стрелкой, чтобы развернуть список.
`DropDownList`	Текстовая часть не редактируема. Пользователь должен делать выбор из списка.
`Simple`	Аналогично `DropDownList`, но окно списка видно постоянно.

Если значения в списке слишком широкие, можно изменить ширину выпадающей части элемента управления свойством `DropDownWidth`.

Свойство `MaxDropDownItems` устанавливает количество отображаемых элементов при отображении выпадающего списка.

`FindString` и `FindStringExact` — это два полезных метода списочных элементов управления. `FindString` находит первую строку в списке, которая начинается с переданного в аргументе фрагмента. `FindStringExact` ищет первую строку, которая буквально соответствует строке, переданной в аргументе. Оба метода возвращают индекс найденного значения либо `-1`, если значение не найдено. Они также могут принимать дополнительный целочисленный аргумент — стартовую позицию поиска.

DateTimePicker

`DateTimePicker` дает возможность пользователю выбирать значение даты или времени (либо и того, и другого) во множестве разнообразных форматов. Можно отображать значения `DateTime` в любом из стандартных форматов даты и времени. Свойство `Format` принимает значения типа перечисления `DateTimePickerFormat`, которые устанавливают формат в `Long`, `Short`, `Time` или `Custom`.

Если свойство `Format` установлено в `DateTimePickerFormat.Custom`, то можно установить свойство `CustomFormat` в строку, представляющую формат.

Предусмотрены также еще два свойства — `Text` и `Value`. Свойство `Text` возвращает текстовое представление значений `DateTime`, в то время как `Value` возвращает сам объект типа `DateTime`. Можно также установить максимальное и минимальное допустимые значения с помощью свойств `MinDate` и `MaxDate`.

Когда пользователь щелкает на стрелке, направленной вниз, отображается календарь, позволяющий выбрать нужную дату. Доступны свойства, предоставляющие возможность изменять внешний вид календаря, устанавливая заголовок и цвета текста и фона для названий месяцев.

Свойство `ShowUpDown` определяет, должны ли в элементе управления отображаться стрелка `UpDown`. Текущее высвеченное значение может изменяться щелчком на стрелке вверх или вниз.

ErrorProvider

ErrorProvider — на самом деле не элемент управления, а компонент. Когда вы перетаскиваете компонент в дизайнер форм, он отображается в лотке компонентов под дизайнером. Назначение ErrorProvider заключается в том, чтобы высвечивать пиктограмму рядом с элементом управления, когда возникает ошибочная ситуация или не проходит проверка. Предположим, что у вас есть поле TextBox, предназначенное для ввода возраста. Ваше бизнес-правило гласит, что значение возраста не должно превышать 65. Если пользователь попытается ввести большее значение, его нужно будет информировать, что введен возраст, превышающий допустимый, и это следует исправить. Проверка правильности введенного значения выполняется в обработчике события Validated этого текстового поля. Если проверка не прошла, можно вызвать метод SetError, передав ссылку на тот элемент управления, который вызвал ошибку, и когда пользователь наведет курсор мыши на пиктограмму, будет отображен текст сообщения об ошибке. На рис. 31.3 показана пиктограмма, которая появляется в случае ввода в текстовое поле недопустимого значения.

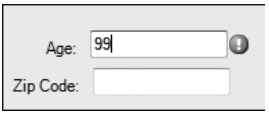

Рис. 31.3. Пиктограмма, которая появляется в случае ввода в текстовом поле недопустимого значения

Вы можете создать ErrorProvider для каждого элемента управления на форме, который может быть причиной ошибки, но если у вас очень много элементов управления, это может оказаться слишком громоздко. Другой вариант — использовать один поставщик ошибок, и в событии проверки вызывать метод IconLocation с тем элементом управления, который вызвал проверку, и одним из значений перечисления ErrorIconAlignment. Это значение устанавливает выравнивание пиктограммы по элементу управления. Затем следует вызвать метод SetError. Если нет никаких ошибочных условий, можно очистить ErrorProvider, вызвав SetError с пустой строкой ошибки. В следующем примере показано, как это работает.

```
private void txtAge_Validating(object sender,
        System.ComponentModel.CancelEventArgs e)
{
  if(txtAge.TextLength > 0 && Convert.ToInt32(txtAge.Text) > 65)
  {
     errMain.SetIconAlignment((Control)sender, ErrorIconAlignment.MiddleRight);
     errMain.SetError((Control)sender, "Значение должно быть меньше 65.");
     e.Cancel = true;
  }
  else
  {
     errMain.SetError((Control)sender, "");
  }
}
private void txtZipCode_Validating(object sender, CancelEventArgs e)
{
  if(txtZipCode.TextLength != 5)
  {
       errMain.SetIconAlignment((Control)sender, ErrorIconAlignment.MiddleRight);
       errMain.SetError((Control)sender, "Должно быть 5 символов.");
       e.Cancel = true;
  }
  else
  {
     errMain.SetError((Control)sender, "");
  }
}
```

Если проверка не проходит (например, в txtAge введено число больше 65), вызывается метод SetIcon поставщика ошибок errMain. Он устанавливает пиктограмму рядом с элементом управления, не прошедшим проверку. Тут же устанавливается текст ошибки, так что когда пользователь наведет курсор мыши на эту пиктограмму, то увидит сообщение, информирующее его о том, что является причиной неудачной проверки.

HelpProvider

HelpProvider (поставщик справки), подобно ErrorProvider, является компонентом, а не элементом управления. HelpProvider позволяет связать элементы управления с темами подсказки. Чтобы ассоциировать элемент управления с поставщиком справки, необходимо вызвать метод SetShowHelp, передав элемент управления и булевское значение, указывающее на необходимость отображения текста подсказки. Свойство HelpNamespace позволяет установить справочный файл. Когда установлено свойство HelpNamespace, содержимое справочного файла отображается в любой момент по нажатию клавиши <F1>, когда элемент управления, зарегистрированный с HelpProvider, находится в фокусе. Можно также установить ключевое слово методом SetHelpKeyword. SetHelpNavigator принимает значение из перечисления HelpNavigator для определения того, какой элемент из справочного файла должен быть отображен. Его можно установить на определенную тему, индекс, таблицу содержания или страницу поиска. SetHelpString ассоциирует строковое значение текста справки с элементом управления. Если свойство HelpNamespace не установлено, то нажатие <F1> покажет текст во всплывающем окне. Двинемся дальше и добавим HelpProvider к предыдущему примеру:

```
helpProvider1.SetHelpString(txtAge,"Введите возраст, не старше 65 лет");
helpProvider1.SetHelpString(txtZipCode, "Введите 5-значный код почтового индекса");
```

ImageList

Компонент ImageList — это именно то, что следует из его названия — список графических изображений. Обычно этот компонент применяется для хранения коллекции изображений, используемых в качестве пиктограмм в панели инструментов или пиктограмм в элементах управления TreeView. Многие элементы управления включают в себя свойство ImageList. Свойство ImageList обычно идет в комплекте с ImageIndex. Свойству ImageList присваивается компонент ImageList, а свойству ImageIndex — индекс в ImageList, представляющий изображение, которое подлежит отображению в элементе управления. Изображения добавляются в компонент ImageList с помощью метода Add свойства ImageList.Images. Свойство Images возвращает ImageCollection.

Два наиболее часто используемых свойства — это ImageSize и ColorDepth. ImageSize получает в качестве значения структуру Size. По умолчанию выбран размер 16×16, но допускается любое значение от 1 до 256. ColorDepth использует значения перечисления ColorDepth. Глубина цвета может быть в пределах от 4 до 32 бит. Для .NET Framework 1.1 значением по умолчанию является ColorDepth.Depth8Bit.

Label

Метки Label применяются для представления пользователю описательного текста. Текст может иметь отношение к другому элементу управления либо к текущему состоянию системы. Обычно метки помещаются рядом с текстовыми полями. Метка предлагает пользователю описание типа данных для ввода в текстовое поле. Элемент

управления Label всегда доступен только для чтения — пользователь не может изменить значение строки в его свойстве Text. Однако вы можете изменять значение свойства Text программно. Свойство UseMnemonic позволяет включить функциональность клавиши доступа. Когда букве в свойстве Text предшествует символ амперсанда (&), эта буква высвечивается с подчеркиванием. Нажатие клавиши <Alt> в сочетании с клавишей этой буквы устанавливает фокус на следующий (в порядке обхода) после метки элемент управления. Если свойство Text уже содержит в тексте амперсанд, то добавление второго не вызовет подчеркивания буквы. Например, если текстом метки должно быть Nuts & Bolts, то свойство должно иметь значение Nuts && Bolts. Поскольку элемент управления Label доступен только для чтения, он не может получать фокус — вот почему фокус передается следующему доступному элементу управления. По этой причине важно помнить, что если вы используете мнемонику (т.е. клавишу быстрого доступа), нужно правильно устанавливать в форме порядок обхода с помощью клавиши табуляции.

Свойство AutoSize содержит булевское значение, указывающее на то, что Label может автоматически изменять свой размер в соответствии со значением текста метки. Это может быть удобно для многоязычных приложений, где длина свойства Text изменяется в зависимости от текущего языка.

ListView

Элемент управления ListView позволяет отображать список элементов одним из нескольких способов. Можно отобразить текст с необязательной крупной пиктограммой, текст с необязательной маленькой пиктограммой или текст и маленькую пиктограмму в вертикальном списке либо в детальном представлении — когда отображается текст элемента с некоторыми дополнительными элементами в последующих столбцах. Это должно показаться вам знакомым, потому что именно так в правой части проводника файлов отображается содержимое текущей папки. ListView содержит коллекцию элементов типа ListViewItem. Класс ListViewItem позволяет устанавливать свойство Text, используемое для отображения. Кроме того, ListViewItem содержит свойство с именем SubItems, которое включает текст, появляющийся в детальном представлении.

Следующий пример демонстрирует использование ListView. Этот пример имеет дело с коротким списком стран. Каждый объект CountryList включает свойства наименования страны, аббревиатуры и валюты. Код класса CountryItem — элемента CountryList — выглядит следующим образом:

```
using System;
namespace FormsSample
{
  public class CountryItem : System.Windows.Forms.ListViewItem
  {
     string _cntryName = "";
     string _cntryAbbrev = "";
     public CountryItem(string countryName,
     string countryAbbreviation, string currency)
     {
        _cntryName = countryName;
        _cntryAbbrev = countryAbbreviation;
        base.Text = _cntryName;
        base.SubItems.Add(currency);
     }
     public string CountryName
     {
```

```
        get {return _cntryName;}
      }
      public string CountryAbbreviation
      {
        get {return _cntryAbbrev;}
      }
    }
}
```

Обратите внимание, что класс `CountryItem` унаследован от `ListViewItem`. Это объясняется тем, что в элемент управления `ListView` можно добавлять только объекты, базирующиеся на `ListViewItem`. В конструкторе передается название страны свойству `base.Text` и добавляется значение валюты в свойство `base.SubItems`. Это позволяет отобразить название страны в списке и валюту — в отдельном столбце детального представления.

Далее в коде формы необходимо добавить несколько объектов `CountryItem` к элементу управления `ListView`:

```
lvCountries.Items.Add(new CountryItem("United States", "US", "Dollar"));
lvCountries.Items[0].ImageIndex = 0;
lvCountries.Items.Add(new CountryItem("Great Britain", "GB", "Pound"));
lvCountries.Items[1].ImageIndex = 1;
lvCountries.Items.Add(new CountryItem("Canada", "CA", "Dollar"));
lvCountries.Items[2].ImageIndex = 2;
lvCountries.Items.Add(new CountryItem("Japan", "JP", "Yen"));
lvCountries.Items[3].ImageIndex = 3;
lvCountries.Items.Add(new CountryItem("Germany", "GM", "Euro"));
lvCountries.Items[4].ImageIndex = 4;
```

Здесь мы добавляем новые элементы типа `CountryItem` в коллекцию `Items` элемента управления `ListView` (`lvCountries`). Отметим, что свойство `ImageIndex` элемента устанавливается после его добавления в список. Предусмотрено два объекта `ImageIndex` — один для больших пиктограмм и один для маленьких (свойства `SmallImageList` и `LargeImageList`). Цель наличия двух `ImageList` с изображениями разного размера состоит в том, чтобы обеспечить добавление элементов в `ImageList` в одинаковом порядке. Таким образом, индекс в каждом `ImageList` будет указывать на одно и то же изображение, но разного размера. В этом примере `ImageList` содержит пиктограммы флагов каждой добавленной в список страны.

В верхней части формы находится элемент управления `ComboBox` (`cbView`), в котором перечислены четыре значения перечисления `View`. Его элементы вставлены следующим образом:

```
cbView.Items.Add(View.LargeIcon);
cbView.Items.Add(View.SmallIcon);
cbView.Items.Add(View.List);
cbView.Items.Add(View.Details);
cbView.SelectedIndex = 0;
```

В событие `SelectedIndexChanged` элемента `cbView` добавляем единственную строку кода:

```
lvCountries.View = (View)cbView.SelectedItem;
```

Это присваивает свойству `View` элемента `lvCountries` новое значение, выбранное в выпадающем списке — элементе управления `ComboBox`. Обратите внимание, что при этом должно быть выполнено приведение к типу `View`, поскольку его свойство `SelectedItem` возвращает `object`.

И последнее, что нужно сделать — по порядку, но не по важности — добавить столбцы в коллекцию `Columns`. Столбцы необходимы для отображения в детальном пред-

ставлении. В данном случае мы добавляем два столбца — Country (Страна) и Currency (Валюта).

Порядок следования столбцов следующий: сначала идет Text элемента типа ListViewItem, затем каждый из элементов коллекции ListViewItem.SubItem — в том порядке, как они появляются в коллекции. Столбцы можно добавлять путем создания объекта ColumnHeader и установки его свойства Text, а также необязательных свойств Width и Alignment. После создания объекта ColumnHeader его можно добавить к свойству Columns. Другой способ добавления столбцов предполагает использование переопределения метода Columns.Add. Он позволяет передать ему сразу значения Text, Width и Alignment нового столбца. Вот пример:

```
lvCountries.Columns.Add("Country", 100, HorizontalAlignment.Left);
lvCountries.Columns.Add("Currency", 100, HorizontalAlignment.Left);
```

Если свойство AllowColumnReorder установлено в true, пользователь имеет возможность перетаскивать заголовки столбцов с места на место, изменяя их последовательность.

Свойство CheckBoxes элемента ListView показывает флажки рядом с элементами списка в ListView. Это дает возможность пользователю легко выбирать множество элементов в элементе управления ListView. Проверить, какие элементы отмечены, можно в коллекции CheckedItems.

Свойство Alingment устанавливает выравнивание пиктограмм в представлениях списка с большими и малыми пиктограммами. Значение может быть любым из перечисления ListViewAlignment, а именно: Default, Left, Top и SnapToGrid. Значение Default разрешает пользователю размещать пиктограммы в любых позициях, где он пожелает. При выборе Left или Top элементы выравниваются по левому краю или по верху ListView. При выборе SnapToGrid элементы ListView расставляются в узлах невидимой сетки в элементе управления ListView. Свойство AutoArrange принимает булевское значение, указывающее на необходимость автоматического выравнивания в соответствии с выбранным значением Alignment.

PictureBox

Элемент управления PictureBox применяется для отображения графических изображений. Изображение может быть в формате BMP, JPEG, GIF, PNG, метафайла или пиктограммы.

Свойство SizeMode использует перечисление PictureBoxSizeMode для определения того, как изображение размещается в элементе управления. SizeMode может быть равно AutoSize, CenterImage, Normal или StretchImage.

Размер отображения PictureBox можно изменять, устанавливая свойство ClientSize. При создании PictureBox сначала создается объект, базирующийся на Image. Например, чтобы загрузить файл JPEG в PictureBox, нужно поступить следующим образом:

```
Bitmap myJpeg = new Bitmap("mypic.jpg");
pictureBox1.Image = (Image)myJpeg;
```

Отметим необходимость приведения к типу Image, поскольку свойство Image элемента управления PictureBox имеет этот тип.

ProgressBar

Элемент управления ProgressBar (индикатор хода работ) используется для визуального представления состояния длительного действия. Он уведомляет пользователя, что нечто происходит, поэтому следует подождать. Для элемента управления ProgressBar устанавливаются значения свойств Minimum и Maximum. Эти свойства со-

ответствуют положению индикатора хода работ в крайнем левом (Minimum) и крайнем правом (Maximum) положениях. Свойство Step устанавливает число, на которое увеличивается значение при каждом вызове метода PerformStep. Можно также использовать метод Increment и увеличивать значение на переданную ему величину. Свойство Value возвращает текущее значение ProgressBar.

С помощью свойства Text можно информировать пользователя о процентной доле выполнения работы или же о количестве оставшихся до ее завершения позиций. Имеется также свойство BackgroundImage, предназначенное для настройки внешнего вида индикатора выполнения.

TextBox, RichTextBox и MaskedTextBox

Элемент управления TextBox — один из наиболее часто используемых. TextBox, RichTextBox и MaskedTextBox унаследованы от TextBoxBase. Класс TextBoxBase представляет такие свойства, как MultiLine и Lines. Свойство MultiLine — булевское значение, позволяющее элементу управления TextBox отображать текст в более чем одной строке. При этом каждая строка в текстовом окне является частью массива строк. Этот массив доступен через свойство Lines. Свойство Text возвращает полное содержимое текстового окна в виде одной строки. TextLength — общая длина текста. Свойство MaxLength ограничивает длину текста определенной величиной.

SelectedText, SelectionLength и SelectionStart имеют дело с текущим выделенным текстом в текстовом окне. Выделенный текст подсвечивается, когда элемент управления получает фокус.

TextBox добавляет множество интересных свойств. AcceptsReturn — булевское значение, позволяющее TextBox воспринимать клавишу <Enter> как символ новой строки либо активизировать кнопку по умолчанию на форме. Когда это свойство имеет значение true, то нажатие <Enter> создает новую строку в TextBox. Свойство CharacterCasing определяет регистр текста в текстовом окне. Перечисление CharacterCasing содержит три значения: Lower, Normal и Upper. Значение Lower переводит в нижний регистр весь текст, независимо от того, как он был введен, Upper переводит весь текст в верхний регистр, а Normal отображает текст так, как он был введен. Свойство PasswordChar позволяет указать символ, который будет отображаться при вводе пользователем всех символов в текстовом окне. Это применяется при вводе паролей и PIN-кодов. Свойство text вернет действительный введенный текст; свойство PasswordChar касается только отображения символов.

RichTextBox — элемент управления, служащий для редактирования текста с расширенными возможностями форматирования. Как следует из его названия, RichTextBox использует Rich Text Format (RTF) для обработки специального форматирования.

Изменения формата обеспечивается свойствами SelectionFont, SelectionColor и SelectionBullet, а форматирование параграфов — свойствами SelectionIndent, SelectionRightIndent и SelectionHangingIndent. Все свойства их группы Selection работают одинаково. Если выделена часть текста, то изменение свойства касается этого выделенного фрагмента. Если же выделенного фрагмента нет, то изменения затрагивают любой текст, вставляемый справа от текущей позиции вставки.

Текст данного элемента управления может быть извлечен из свойства Text либо Rtf. Свойство Text возвращает простой текст элемента управления, в то время как Rtf — форматированный текст.

Метод LoadFile может загружать текст из файла двумя различными способами. Он может использовать либо строку, представляющую путевое имя файла, либо потоковый объект. Можно также специфицировать RichTextBoxStreamType. В табл. 31.3 перечислены значения RichTextBoxStreamType.

Таблица 31.3. Значения `RichTextBoxStreamType`

Значение	Описание
`PlainText`	Информация, связанная с форматированием, отсутствует. В тех местах, где находятся OLE-объекты, используются пробелы.
`RichNoOleObjs`	Форматирование Rich Text Format, но на месте OLE-объектов находятся пробелы.
`RichText`	Форматированный текст RTF и OLE-объектами на месте.
`TextTextOleObjs`	Простой текст с текстом, заменяющим OLE-объекты.
`UnicodePlainText`	То же, что и `PlainText`, но в кодировке Unicode.

Метод `SaveFile` работает с несколькими параметрами и сохраняет текст из элемента управления в указанном файле. Если файл с таким именем уже существует, он перезаписывается.

`MaskedTextBox` предоставляет возможность ограничить то, что пользователь может ввести, а также позволяет автоматически форматировать введенные данные. Несколько свойств используются для проверки допустимости формата пользовательского ввода. `Mask` — свойство, содержащее строку маски. Маскирующая строка аналогична строке форматирования. Допустимое количество символов, тип допустимых символов, формат данных — все это устанавливается строкой `Mask`. Класс, основанный на `MaskedTextProvider`, также может представлять необходимую информацию для форматирования и проверки. `MaskedTextProvider` можно устанавливать, только передавая его одному из конструкторов.

Три различных свойства возвращают текст `MaskedTextProvider`. Свойство `Text` возвращает текст, содержащийся в элементе управления в данный момент. Оно может отличаться в зависимости от того, имеет ли элемент фокус, и от значения свойства `HidePromptOnLeave`. Приглашение (prompt) — это строка, которую видит пользователь и которая подсказывает ему, что нужно ввести. Свойство `InputText` всегда возвращает только тот текст, который ввел пользователь.

Свойство `OutputText` возвращает текст, сформатированный на базе свойств `IncludeLiterals` и `IncludePrompt`. Если, например, маска предназначена для ввода номера телефона, то строка `Mask`, по всей видимости, должна включать скобки и несколько тире. Это могут быть литеральные символы, которые включаются в свойство `OutputText`, если свойству `IncludeLiteral` было присвоено значение `true`.

В элементе управления `MaskedTextBox` также присутствует пара дополнительных событий. `OutputTextChanged` и `InputTextChanged` возбуждаются, когда изменяются значения `InputText` или `OutputText`.

Panel

`Panel` — простой элемент управления, содержащий в себе другие элементы управления. За счет группирования вместе элементов управления и помещения их в панель существенно упрощается управление ими. Например, можно сделать недоступными все элементы управления в панели, просто сделав недоступной всю панель. Поскольку `Panel` наследуется от `ScrollableControl`, также можно воспользоваться преимуществами `AutoScroll`. Если в пределах доступной области нужно отобразить слишком много элементов управления, поместите их в панель и установите значение `true` свойству `AutoScroll` — после этого их можно будет прокручивать в пределах этой области.

Панели по умолчанию не отображают рамки, но, присвоив значение свойству BorderStyle, можно визуально группировать взаимосвязанные элементы управления посредством рамок. Это делает пользовательский интерфейс более дружественным.

Panel — базовый класс для FlowLayoutPanel, TableLayoutPanel, TabPage и SplitterPanel. Используя эти элементы управления, можно создавать сложные и профессионально выглядящие экранные формы или окна. FlowLayoutPanel и TableLayoutPanel особенно удобны для создания форм с изменяемым размером.

FlowLayoutPanel и TableLayoutPanel

FlowLayoutPanel и TableLayoutPanel — это новые дополнения к .NET Framework. Как можно предположить по их названиям, эти панели предоставляют возможность компоновки элементов управления с использованием той же парадигмы, что и Web Forms. FlowLayoutPanel — это контейнер, позволяющий содержащимся в нем элементам "плавать" либо в горизонтальном, либо в вертикальном направлении. Вместо "плавания" элементы панели можно закрепить. Направление размещения устанавливается свойством FlowDirection и перечислением FlowDirection.

Свойство WrapContents определяет, должны ли элементы управления переходить на следующую строку или столбец, когда размер формы изменяется или когда элементы закрепляются.

TableLayoutPanel использует сеточную структуру для управления компоновкой элементов управления. Любой элемент управления Windows Forms может быть вставлен в TableLayoutPanel, включая другую панель TableLayoutPanel. Это позволяет получать очень гибкий и динамичный дизайн окон. Когда элемент управления добавляется в TableLayoutPanel, четыре дополнительных свойства добавляются в категорию Layout окна свойств. Это Column, ColumnSpan, Row и RowSpan. Во многом подобно HTML-таблице на Web-странице, для каждого элемента управления могут быть установлены промежутки между столбцами и строками. По умолчанию элемент управления центрируется в ячейке таблицы, но это можно изменить свойствами Dock и Anchor.

Стиль по умолчанию для строк и столбцов может быть изменен с помощью коллекций RowStyles и ColumnsStyles. Эти коллекции содержат, соответственно, объекты RowStyle и ColumnsStyle. Объекты Style имеют общее свойство — SizeType. Это свойство содержит значение типа перечисления SizeType, определяющее то, как должны устанавливаться ширина столбца и высота строки. Значения включают AutoSize, Absolute и Percent. Значение AutoSize разделяет пространство с другими равноправными элементами управления, Absolute позволяет установить количество пикселей размера, а Percent задает размер элемента управления в процентах от размера родительского элемента (панели).

Строки, столбцы и дочерние элементы управления могут добавляться и удаляться динамически во время выполнения. Свойство GrowStyle принимает значение из перечисления TableLayoutPanelGrowStyle, которое заставляет таблицу добавлять столбец, строку или сохранять фиксированный размер, когда новый элемент управления добавляется в полную таблицу. Если установлено значение FixedSized, то при попытке добавить элемент возбуждается исключение ArgumentException. Если же в таблице есть пустая ячейка, то элемент управления помещается в нее. Это свойство имеет эффект, только если таблица полна, и в нее добавляется элемент управления.

Форма formPanel в примере приложения содержит панели FlowLayoutPanels и TableLayoutPanels, каждая из которых содержит разнообразные эелементы управления. Экспериментируя с элементами управления, особенно со свойствами Dock и Anchor элементов управления, расположенных в панелях компоновки, вам будет легче всего понять принцип их работы.

SplitContainer

Элемент управления SplitContainer — это на самом деле три элемента в одном. Он состоит из двух панелей с линейкой или разделителем между ними. Пользователь может перемещать эту линейку, изменяя размеры панелей. При изменении размеров панелей элементы управления, содержащиеся на них, также могут изменять свой размер. Лучшим примером SplitContainer может служить проводник файлов (File Explorer). Левая панель содержит TreeView с папками, а правая — ListView с содержимым этих папок. Когда пользователь перемещает курсор мыши над линейкой разделителя, он изменяет свой вид, говоря о том, что линейку можно перемещать. SplitContainer может содержать любые элементы управления, включая панели с компоновками и другие SplitContainer. Это позволяет создавать очень сложные и развитые формы.

Элемент управления Splitter генерирует два события, связанные с перемещением: SplitterMoving и SplitterMoved. Одно происходит во время перемещения линейки, а другое — после завершения движения. Оба принимают аргумент SplitterEventArgs. SplitterEventArgs содержит свойства для координат X и Y левого верхнего угла Splitter (SplitX и SplitY), а также координаты X и Y указателя мыши (X и Y).

TabControl и TabPages

TabControl позволяет группировать связанные элементы управления в серии страниц-вкладок. TabControl управляет коллекцией элементов типа TabPages. Несколько свойств управляют внешним видом TabControl. Свойство Appearance использует перечисление TabAppearance для определения внешнего вида вкладок. Допустимыми значениями являются FlatButtons, Buttons и Normal. Свойство Multiline булевского типа указывает на то, что может отображаться более одной строки вкладок. Если свойство Multiline установлено в false, а количество вкладок превышает такое, что не может уместиться на экране, появляется пара кнопок, позволяющая прокручивать вкладки и видеть те, что не уместились.

Свойство Text элемента TabPage — это то, что отображается на отдельной вкладке. Свойство Text устанавливается через параметр конструктора.

Создав элемент управления TabPage, вы получаете контейнер, куда можно помещать другие элементы управления. Средствами дизайнера Visual Studio .NET легко добавить элемент TabPage к элементу управления TabControl, используя редактор коллекций. При добавлении каждой такой страницы можно установить множество ее свойств, затем перетащить на нее другие дочерние элементы управления.

Получить текущую вкладку можно из свойства SelectedTab. Событие SelectedIndex возникает при каждом переключении вкладки. Прослушивая свойство SelectedIndex и затем подтверждая текущий выбор страницы через SelectedTab, вы можете организовать специальную обработку для каждой вкладки. Вы могли бы, к примеру, управлять данными, отображаемыми для каждой вкладки.

ToolStrip

Элемент управления ToolStrip — это контейнер, используемый для создания панелей инструментов, структур меню и строк состояния. ToolStrip используется непосредственно для панелей инструментов и в качестве базового класса для MenuStrip и StatusStrip.

Применяемый в качестве панели инструментов, элемент управления ToolStrip использует набор элементов управления, происходящих от класса ToolStripItem.

Класс ToolStripItem добавляет общую функциональность отображения и размещения, а также управляет большинством событий, связанных с элементами управления. ToolStripItem унаследован от класса System.ComponentModel.Component, а не от Control. Основанные на ToolStripItem классы должны содержаться в контейнере, происходящем от ToolStrip.

Image и Text — возможно, наиболее часто используемые свойства. Графические изображения могут быть установлены либо через свойство Image, либо с использованием элемента управления ImageList и установкой свойства ImageList элемента управления ToolStrip. Затем могут быть установлены свойства ImageIndex индивидуальных элементов управления.

Форматирование текста в ToolStripItem управляется свойствами Font, TextAlign и TextDirection. Свойство TextAlign устанавливает выравнивание текста относительно элемента управления. Это может быть любое значение из перечисления ControlAlignment. По умолчанию принимается MiddleRight.

Свойство TextDirection задает ориентацию текста. Значения могут быть любыми из перечисления ToolStripTextDirection, а именно — Horizontal, Inherit, Vertical270 и Vertical90. Vertical270 поворачивает текст на 270 градусов, а Vertical90 — на 90 градусов.

Свойство DisplayStyle управляет тем, отображается ли текст, изображение, то и другое или ни то ни другое на поверхности элемента управления. Когда AutoSize установлено в true, ToolStripItem будет изменять свой размер, потому потребуется минимальное пространство.

Элементы управления, унаследованные от ToolStripItem, перечислены в табл. 31.4.

Таблица 31.4. Элементы управления, унаследованные от ToolStripItem

Элемент	Описание
ToolStripButton	Представляет кнопку, доступную для пользовательского выбора.
ToolStripLabel	Отображает не выбираемый текст или изображения на ToolStrip. ToolStripLabel также может отображать одну или более гиперссылок.
ToolStripSeparator	Используется для отделения и группирования других ToolStripItems. Элементы могут быть сгруппированы в соответствии с их функциональностью.
ToolStripDropDownItem	Отображает выпадающие элементы. Базовый класс для ToolStripDropDownButton, ToolStripMenuItem и ToolStripSplitButton.
ToolStripControlHost	Место размещения в ToolStrip других элементов управления, не унаследованных от ToolStripItem. Базовый класс для ToolStripComboBox, ToolStripProgressBar и ToolStripTextBox.

Два элемента этого списка — ToolStripDropDownItem и ToolStripControlHost — требуют некоторых пояснений. ToolStripDropDownItem — это базовый класс для ToolStripMenuItem, используемого для построения структур меню. Элементы ToolStripMenuItem добавляются к элементам управления MenuStrip. Как упоминалось ранее, MenuStrip унаследован от ToolStrip. Это становится важным, когда приходит время манипулировать или расширять элементы меню. Поскольку панели инструментов и меню унаследованы от одних и тех же классов, создание каркасов для управления и выполнения команд становится намного проще.

ToolStripControlHost может использоваться для размещения в себе элементов управления, чей тип не наследуется от ToolStripItem. Вспомним, что напрямую в ToolStrip могут размещаться только элементы управления, унаследованные от ToolStripItem. В следующем примере показано, как поместить элемент управления DateTimePicker на ToolStrip.

```
public mdiParent()
{
  InitializeComponent();
  ToolStripControlHost _dateTimeCtl;
  dateTimeCtl = new ToolStripControlHost(new DateTimePicker());
     ((DateTimePicker)_dateTimeCtl.Control).ValueChanged +=
            delegate {
                     toolStripLabel1.Text =
  ((DateTimePicker)_dateTimeCtl.Control).Value.Subtract(DateTime.Now).ToString();
                     };

  dateTimeCtl.Width = 200;
  dateTimeCtl.DisplayStyle = ToolStripItemDisplayStyle.Text;
  toolStrip1.Items.Add(_dateTimeCtl);
}
```

Это конструктор формы frmMain из кода примера. Сначала объявляется и создается экземпляр ToolStripControlHost. Обратите внимание, что когда создается этот экземпляр, элемент управления, который должен быть в нем размещен, передается конструктору. Следующая строка кода устанавливает обработчик события ValueChanged для элемента управления DateTimePicker. Он может быть доступен через свойство Control класса ToolStripHostControl, потому его нужно привести обратно к правильному типу элемента управления. Когда это сделано, можно обращаться к свойствам и методам размещенного внутри ToolStripHostControl элемента управления.

Другой способ сделать это, обеспечивающий лучшую инкапсуляцию — создать новый класс, унаследованный от ToolStripControlHost. Приведенный ниже код представляет другую версию включения DateTimePicker в ToolStrip как нового класса ToolStripDateTimePicker.

```
namespace FormsSample.SampleControls
{
  public class DTPickerToolStrip: System.Windows.Forms.ToolStripControlHost
  {
     public event EventHandler ValueChanged;
     public DTPickerToolStrip () : base(new DateTimePicker())
     {
     }
     public new DateTimePicker Control
     {
        get{return (DateTimePicker)base.Control;}
     }
     public DateTime Value
     {
        get { return Control.Value; }
     }
     protected override void OnSubscribeControlEvents(Control control)
     {
        base.OnSubscribeControlEvents(control);
        ((DateTimePicker)control).ValueChanged +=
             new EventHandler(ValueChangedHandler);
     }
```

```
        protected override void OnUnsubscribeControlEvents(Control control)
        {
           base.OnSubscribeControlEvents(control);
           ((DateTimePicker)control).ValueChanged -=
                new EventHandler(ValueChanged);
        }
        private void ValueChangedHandler (object sender, EventArgs e)
        {
           if (ValueChanged != null)
                ValueChanged(this, e);
        }
    }
}
```

Бо́льшая часть того, что делает этот класс — это показ избранных свойств, методов и событий класса `DateTimePicker`. Таким образом, ссылка на лежащий в основе элемент управления не должна будет поддерживаться приложением-хостом. Процесс показа событий несколько более сложен. Метод `OnSubscribeControlEvents` применяется для синхронизации событий размещенного элемента управления, в данном случае — `DateTimePicker`, с классом, происходящим от `ToolStripControlHost`, которым здесь является `ToolStripDateTimePicker`. В этом примере событие `ValueChanged` передается `DTPickerToolStrip`.

Это дает возможность пользователю элемента управления так настроить событие в приложении-хосте, как если бы `DTPickerToolStrip` был наследником `DateTimePicker` вместо `ToolStripControlHost`. Следующий пример кода демонстрирует это. В коде использован `DTPickerToolStrip`.

```
public mdiParent()
{
  DTPickerToolStrip otherDateTimePicker = new DTPickerToolStrip();
  otherDateTimePicker.Width = 200;
  otherDateTimePicker.ValueChanged +=
  new EventHandler(otherDateTimePicker_ValueChanged);
  toolStrip1.Items.Add(otherDateTimePicker);
}
```

Отметим, что когда устанавливается обработчик события `ValueChanged`, используется ссылка на класс `DTPickerToolStrip` вместо `DateTimePicker`, как в предыдущем примере. Видно, насколько яснее этот код выглядит по сравнению с предыдущим. Мало того, поскольку `DateTimePicker` помещен в оболочку другого класса, степень инкапсуляции значительно возросла, а `DTPickerToolStrip` стало гораздо проще использовать в других частях приложения и других проектах.

MenuStrip

Элемент управления `MenuStrip` — это контейнер для структур меню в приложении. Как упоминалось ранее, класс `MenuStrip` унаследован от `ToolStrip`. Система меню строится добавлением объектов `ToolStripMenu` к `MenuStrip`. Это можно сделать в коде или в дизайнере Visual Studio. Для этого нужно перетащить элемент управления `MenuStrip` на форму в дизайнере, и этот `MenuStrip` позволит вводить текст меню непосредственно в элементы меню.

Элемент управления `MenuStrip` включает лишь пару дополнительных свойств. `GripStyle` использует перечисление `ToolStripGripStyle` для установки видимости. Свойство `MdiWindowListItem` принимает и возвращает `ToolStripMenuItem`. Этот `ToolStripMenuItem` будет представлять меню, которое отображают все открытые окна в MDI-приложении.

ContextMenuStrip

Класс ContextMenuStrip применяется для показа контекстного меню, или меню, отображаемого по нажатию правой кнопки мыши. Подобно MenuStrip, ContextMenuStrip является контейнером объектов ToolStripMenuItem. Однако он унаследован от ToolStripDropDownMenu. Элемент ContextMenuStrip создается так же, как MenuStrip. К нему добавляются элементы ToolStripMenuItem и определяются события Click каждого элемента для выполнения специфического действия. Контекстное меню назначается конкретному элементу управления. Это делается установкой свойства ContextMenuStrip элемента управления. Когда пользователь щелкает правой кнопкой мыши в поле элемента управления, отображается упомянутое меню.

ToolStripMenuItem

ToolStripMenuItem — класс, служащий для построения структур меню. Каждый объект ToolStripMenuItem представляет отдельный пункт в системе меню.

Каждый ToolStripMenuItem владеет коллекцией ToolStripItemCollection, поддерживающей дочерние меню. Эта функциональность унаследована от ToolStripDropDownItem.

Поскольку ToolStripMenuItem наследуется от ToolStripItem, к нему применимы все те же самые свойства форматирования. Изображения появляются как маленькие пиктограммы справа от текста меню. Элементы меню могут иметь флажки для отметки, находящиеся рядом с ними, определяемые с помощью свойств Checked и CheckState.

Каждому пункту меню могут быть назначены горячие клавиши. Обычно это сочетание двух клавиш, как, например, <Ctrl+C> (обычное сокращение для операции копирования). Когда горячая клавиша назначена, она может быть необязательно отображена в меню установкой значения свойства ShowShortCutKey в true.

Чтобы быть полезным, пункт меню должен что-то делать, когда пользователь щелкает на нем или нажимает сочетание горячих клавиш. Чаще всего для этого следует обработать событие Click. Если используется свойство Checked, то события CheckStateChanged и CheckedChanged могут применяться для определения изменения состояния метки.

ToolStripManager

Структуры меню и панелей инструментов могут вырасти до такого размера, что ими становится трудно управлять. Класс ToolStripManager предоставляет возможность создания маленьких, более управляемых фрагментов структур меню или панели инструментов с тем, чтобы потом при необходимости их комбинировать. Примером этого может служить форма, содержащая несколько элементов управления. Каждый из них должен отображать контекстное меню. Несколько пунктов меню должны быть доступны всем элементам управления, но каждый из них также содержит пару своих уникальных пунктов. Общие пункты меню могут быть определены в одном ContextMenuStrip. Каждый из уникальных пунктов меню может быть определен предварительно или создан во время выполнения. Для каждого элемента, которому требуется контекстное меню, общее меню клонируется, и к нему добавляются уникальные пункты с помощью метода ToolStripManager.Merge. Результирующее меню назначается свойству ContextMenuStrip элемента управления.

ToolStripContainer

Элемент управления ToolStripContainer используется для стыковки элементов управления, основанных на ToolStrip. Добавление ToolStripContainer и установка свойству Docked значения Fill добавляет ToolStripPanel к каждой стороне формы, а ToolStripContainerPanel — в середину формы. Любой ToolStrip (ToolStrip, MenuStrip или StatusStrip) может быть добавлен к любой из панелей ToolStripPanel. Пользователь может переместить ToolStrip с помощью мыши на любую сторону или в нижнюю часть формы. Установив значение false свойству Visible для любой из панелей ToolStripPanel, можно запретить помещение на панель ToolStrip. Панель ToolStripContainerPanel в центре формы может быть использована для размещения других элементов управления, которые ей понадобятся.

Формы

Ранее в этой главе уже было показано, как создавать простое Windows-приложение. Пример содержал один класс, унаследованный от System.Windows.Forms.Form. Согласно документации .NET Framework, "форма — это представление любого окна в вашем приложении". Если у вас есть опыт разработки на Visual Basic, то термин *форма* вам должен быть знаком. Если же ваш опыт лежит в области C++ с применением MFC, то, вероятно, вы привыкли называть формы окнами, диалоговыми окнами или фреймами. Независимо от этого, можно сказать, что форма — это основное средство взаимодействия с пользователем. Ранее в этой главе уже раскрывались некоторые из наиболее часто используемых свойств, методов и событий класса Control, а поскольку класс Form является наследником Control, все те же методы, свойства и события присутствуют также в классе Form. Класс Form добавляет значительный объем функциональности к той, что обеспечена классом Control, и мы поговорим о ней в этом разделе.

Класс Form

Клиентские Windows-приложения могут содержать от одной до сотен различных форм. Формы могут быть основаны на SDI (Single Document Interface — однодокументный интерфейс) или же на MDI (Multiple Document Interface — многодокументный интерфейс). Независимо от этого, сердцем Windows-клиента остается класс System.Windows.Forms.Form. Класс Form унаследован от ContainerControl, который, в свою очередь, унаследован от ScrollableControl — прямого потомка Control. Отсюда можно предположить, что форма может служить контейнером для других элементов управления, предоставлять возможность прокрутки содержимого, когда оно не умещается в клиентской области, а также обладать множеством тех же свойств, методов и событий, которые присущи другим элементам управления. Все это делает класс Form достаточно сложным. В данном разделе мы рассмотрим бо́льшую часть его функциональности.

Создание и уничтожение экземпляра формы

Важно хорошо понять процесс создания формы. То, что вы хотите сделать, зависит от того, где вы напишете инициализационный код. При создании экземпляра формы события происходят в следующем порядке:

- конструктор
- Load
- Activated
- Closing
- Closed
- Deactivate

Первые три события касаются инициализации. То, какой тип инициализации вы предпочитаете, может определить событие, в которое должен быть помещен необходимый код. Конструктор класса срабатывает при создании экземпляра объекта. Событие `Load` происходит после создания экземпляра, но перед появлением формы на экране. Разница между ними связана с существованием формы. Когда возникает событие `Load`, форма уже существует, хотя пока еще и невидима. Во время выполнения конструктора форма находится в процессе создания. Событие `Activated` происходит, когда форма становится видимой и текущей.

Бывают ситуации, когда этот порядок может быть слегка изменен. Если во время конструирования свойство `Visible` установить в `true` или вызвать метод `Show` (устанавливающий `Visible` в `true`), то событие `Load` возбуждается немедленно. А поскольку это также делает форму видимой и текущей, также сразу возбуждается событие `Activated`. Если имеется код после установки `Visible` в `true`, он выполняется. Поэтому последовательность событий будет выглядеть примерно так:

- конструктор, до выражения `Visible = true`
- `Load`
- `Activate`
- конструктор, после выражения `Visible = true`

Потенциально это может привести к некоторым нежелательным результатам. С точки зрения передового опыта может показаться, что выполнить как можно больше работ по инициализации в конструкторе — хорошая идея.

Но что происходит, когда форма закрывается? Событие `Closing` предоставляет возможность прервать процесс. Событие `Closing` принимает параметр `CancelEventArgs`. Этот аргумент имеет свойство `Cancel`, которое, будучи установленным в `true`, прерывает событие и оставляет форму открытой. Событие `Closing` случается при попытке закрыть форму, в то время как событие `Closed` — после ее закрытия. Оба они дают возможность выполнить необходимую очистку. Отметим, что событие `Deactivate` происходит уже после закрытия формы. Это еще один потенциальный источник трудноуловимых ошибок. Всегда нужно убедиться, что вы не делаете ничего такого в `Deactivate`, что помешало бы нормальной очистке формы сборщиком мусора. Так, например, установка ссылки на другой объект может стать причиной того, что форма останется в памяти. Если выполняется вызов метода `Application.Exit()`, когда открыта одна или более форм, события `Closing` и `Closed` не возбуждаются. Это важное обстоятельство, особенно если существуют открытые файлы или подключения к базам данных, которые необходимо очистить. Метод `Dispose` вызывается обязательно, поэтому, возможно, имеет смысл помещать большую часть кода очистки в этот метод.

Некоторые свойства, имеющие отношение к запуску формы — это `StartPosition`, `ShowInTaskbar` и `TopMost`.

`StartPosition` может принимать значения из перечисления `FormStartPosition`:

- `CenterParent` — форма центрируется в клиентской области родительской формы;
- `CenterScreen` — форма центрируется на текущем экране;
- `Manual` — местоположение формы основано на свойстве `Location`;
- `WindowsDefaultBounds` — форма располагается в позиции по умолчанию, определяемой Windows, и имеет размеры по умолчанию;
- `WindowsDefaultLocation` — форма располагается в позиции по умолчанию, определяемой Windows, но ее размеры определяются свойством `Size`.

Свойство `ShowInTaskbar` определяет, должна ли форма быть доступной в панели задач (taskbar). Это касается только тех случаев, когда форма является дочерней, а вы хо-

тите, чтобы в панели задач отображалась только родительская форма. Свойство TopMost сообщает форме, что она должна находиться в верхней позиции Z-порядка приложения. Это справедливо даже для тех случаев, когда форма не получает фокус немедленно.

Чтобы пользователи могли взаимодействовать с приложением, они должны видеть форму. Это обеспечивается методами Show и ShowDialog. Метод Show просто делает форму видимой пользователю. В следующем фрагменте кода показано, как создать форму и отобразить ее пользователю. Предположим, что класс формы, которую нужно отобразить, называется MyFormClass:

```
MyFormClass myForm = new MyFormClass();
myForm.Show();
```

Это — простейший путь. Единственный недостаток состоит в том, что вызывающий код не получает никакого уведомления о том, что форма myForm завершила работу и была закрыта. Иногда это не представляет проблемы, и метод Show будет работать нормально. Если же все-таки требуется какое-то уведомление, лучше воспользоваться методом ShowDialog.

Когда вызывается метод Show, то код, следующий за этим вызовом, выполняется немедленно. Когда же вызывается ShowDialog, то вызывающий код блокируется до тех пор, пока форма, чей метод ShowDialog был вызван, не будет закрыта. Но при этом не только вызывающий код блокируется, но еще форма необязательно возвращает значение DialogResult. Перечисление DialogResult представляет собой список идентификаторов, описывающих причину закрытия формы. Они включают OK, Cancel, Yes, No и ряд других. Для того чтобы форма возвратила DialogResult, должно быть установлено ее свойство DialogResult, либо же свойство DialogResult должно быть установлено для одной из кнопок формы.

Например, предположим, что часть приложения запрашивает телефонный номер клиента. Форма включает текстовое поле для ввода этого номера и две кнопки: одну с меткой OK и другую — с меткой Cancel (Отмена). Если установить значение свойства DialogResult кнопки OK равным DialogResult.OK, а значение свойства DialogResult кнопки Cancel — DialogResult.Cancel, то когда любая их этих кнопок будет выбрана, форма станет невидимой и вернет вызвавшей ее форме соответствующее значение DialogResult. Теперь представим, что форма не уничтожена, а только свойство Visible установлено в false. Это может понадобиться для того, чтобы получить какую-то информацию от формы. Например, тот же телефонный номер. За счет создания в форме свойства для хранения телефонного номера появляется возможность в родительской форме получить это значение и вызвать метод Close дочерней формы. Код такой дочерней формы может выглядеть следующим образом:

```
namespace FormsSample.DialogSample
{
  partial class Phone : Form
  {
     public Phone()
     {
       InitializeComponent();
       btnOK.DialogResult = DialogResult.OK;
       btnCancel.DialogResult = DialogResult.Cancel;
     }
     public string PhoneNumber
     {
       get { return textBox1.Text; }
       set { textBox1.Text = value; }
     }
  }
}
```

Первое, что необходимо отметить — здесь нет никакого кода, обрабатывающего щелчки на кнопках. Поскольку свойство DialogResult устанавливается для каждой из кнопок, форма исчезает с экрана после щелчка либо на OK, либо на Cancel. Единственное добавленное свойство — PhoneNumber. В следующем коде показан метод родительской формы, вызывающей диалог Phone.

```
Phone frm = new Phone();
frm.ShowDialog();
if (frm.DialogResult == DialogResult.OK)
{
  label1.Text = "Номер телефона: " + frm.PhoneNumber;
}
else if (frm.DialogResult == DialogResult.Cancel)
{
  label1.Text = "Форма отменена.";
}
frm.Close();
```

Это выглядит достаточно просто. Создается новый объект Phone с именем frm. Когда вызывается метод frm.ShowDialog(), выполнение кода родительской формы останавливается и ожидает возврата формы из Phone. После этого можно проверить свойство DialogResult формы Phone. Поскольку в этот момент она еще не уничтожена, а только стала невидимой, вы по-прежнему можете обращаться к ее общедоступным свойствам, одним из которых является свойство PhoneNumber. После получения необходимых данных можно вызвать метод формы Close.

Все работает хорошо, но что если возвращенный номер телефона сформатирован некорректно? Если поместить ShowDialog внутрь цикла, то можно вызывать его повторно и запрашивать повторный ввод значения. Таким образом, в конце концов, получаем правильное значение. Не забудьте, что нужно также обрабатывать DialogResult.Cancel, если пользователь щелкает на кнопке Cancel.

```
Phone frm = new Phone();
while (true)
{
  frm.ShowDialog();
  if (frm.DialogResult == DialogResult.OK)
  {
     label1.Text = "Номер телефона: " + frm.PhoneNumber;
     if (frm.PhoneNumber.Length == 8 | frm.PhoneNumber.Length == 12)
     {
       break;
     }
     else
     {
       MessageBox.Show("Неверный формат номера телефона. Пожалуйста, исправьте. ");
     }
  }
  else if (frm.DialogResult == DialogResult.Cancel)
  {
     label1.Text = "Форма отменена. ";
     break;
  }
}
frm.Close();
```

Теперь, если номер телефона не проходит простую проверку длины, форма Phone появляется вновь, чтобы пользователь смог исправить ошибку. Метод ShowDialog не создает нового экземпляра формы. Любой текст, введенный в форму, останется в ней, поэтому если форма должна быть очищена, это должен будет сделать сам пользователь.

Внешний вид

Первое, что видит пользователь — это форма приложения. Она должна появляться первой и быть максимально функциональной. Если приложение не решает никаких бизнес-задач, то на самом деле не важно, как она выглядит. Это не значит, однако, что форма и весь дизайн графического интерфейса пользователя не должны быть приятными глазу. Такие простые вещи, как комбинация цветов, размеры шрифтов и окон могут значительно облегчить работу для пользователя.

Иногда не нужно, чтобы пользователь имел доступ к системному меню. Имеется в виду меню, которое появляется по щелчку на пиктограмме, расположенной в верхнем левом углу окна. Обычно оно содержит такие элементы, как `Restore`, `Minimize`, `Maximize` и `Close`. Свойство `ControlBox` позволяет установить видимость системного меню. Видимость кнопок `Minimize` и `Maximize` можно задать свойствами `MaximizeBox` и `MinimizeBox`. Если удалить все кнопки и затем присвоить свойству `Text` пустую строку (`""`), то полоса заголовка исчезнет полностью.

Если установить свойство `Icon` формы и не установить `ControlBox` в `false`, то в левом верхнем углу появится пиктограмма. Обычно ее устанавливают в `app.ico`. Это присваивает каждой форме пиктограмму, совпадающую с пиктограммой приложения.

Свойство `FormBorderStyle` устанавливает тип рамки, окружающей форму через значения из перечисления `FormBorderStyle`:

- `Fixed3D`
- `FixedDialog`
- `FixedSingle`
- `FixedToolWindow`
- `None`
- `Sizable`
- `SizableToolWindow`

Бо́льшая часть приведенных выше значений не требует пояснений, за исключением двух рамок инструментальных (`Tool`) окон. Окна `Tool` не появляются в панели задач, независимо от установки `ShowInTaskBar`. Кроме того, такие окна не отображаются в списке окон, когда нажимается комбинация <Alt+Tab>. По умолчанию установлен тип рамки `Sizeable`.

Если только не предъявлены специальные требования, цвета большинства элементов графического интерфейса пользователя должны быть установлены системными, а не какими-то конкретными. При этом если какой-нибудь пользователь предпочитает иметь зеленые кнопки с красным текстом, он может это настроить для всей системы и приложение примет такие настройки. Чтобы установить для элемента управления некоторый системный цвет, нужно вызвать метод `FromKnownColor` класса `System.Drawing.Color`. Этот метод принимает значение из перечисления `KnownColor`. Многие цвета определены в этом перечислении как цвета различных элементов графического интерфейса пользователя — например, `Control`, `ActiveBorder` и `Desktop`. Поэтому если, например, цвет фона формы должен всегда соответствовать цвету `Desktop`, нужно применить следующий код:

```
myForm.BackColor = Color.FromKnownColor(KnownColor.Desktop);
```

Теперь, если пользователь изменит цвет своего рабочего стола, цвет фона формы изменится вместе с ним. Это правильный, дружественный подход к настройке внешнего вида приложения. Иногда пользователи выбирают довольно странные цветовые сочетания для своих рабочих столов, однако, это их выбор.

В операционной системе Windows XP было введено средство под названием визуальных стилей. Визуальные стили изменяют способ отображения кнопок, текстовых полей, меню и других элементов управления, а также вид указателя мыши.

Разрешить визуальные стили для своих приложений можно с помощью метода `Application.EnableVisualStyles`. Этот метод должен быть вызван перед созданием элементов графического интерфейса любого рода. По этой причине обычно он вызывается в методе `Main`, как показано в следующем примере:

```
[STAThread]
static void Main()
{
  Application.EnableVisualStyles();
  Application.Run(new Form1());
}
```

Этот код позволяет многим элементам управления, поддерживающим визуальные стили, воспользоваться их преимуществами. Из-за последствий применения метода `EnableVisualStyles()` сразу после его вызова может понадобиться вызвать метод `Application.DoEvents()`. Это решит проблему исчезновения пиктограмм и панелей инструментов во время выполнения. К тому же метод `EnableVisualStyles` доступен только в .NET Framework 1.1.

Вам доведется решать еще одну задачу, имеющую отношение к элементам управления. Бо́льшая часть элементов управления имеет свойство `FlatStyle`, принимающее значения из перечисления `FlatStyle`:

- `Flat` — аналогично `flat`, за исключением того, что когда указатель мыши находится на элементе управления, он принимает трехмерную форму;
- `Standard` — элемент управления имеет трехмерную форму;
- `System` — внешний вид элемента управления управляется системой.

Чтобы включить визуальные стили, свойство `FlatStyle` элемента управления должно быть установлено равным `FlatStyle.System`. После этого приложение будет иметь внешний вид XP и поддерживать темы XP.

Многодокументный интерфейс (MDI)

Приложения в стиле MDI используются, когда нужно отображать либо множество экземпляров однотипных форм, либо разных форм, которые должны быть включены в приложение одинаковым образом. Примером множества экземпляров однотипных форм может служить текстовый редактор, одновременно отображающий множество окон редактирования. Примером приложений второго типа может служить Microsoft Access. В нем можно одновременно иметь открытыми окна запросов, окна дизайнера и окна таблиц. Все эти окна никогда не выходят за пределы главного окна приложения Access.

Проект, содержащий примеры для этой главы, сам является примером MDI-приложения. Форма `mdiParent` в этом проекте представляет собой родительскую форму MDI. Установка свойству `IsMdiContainer` значения `true` делает любую форму родительской формой MDI. Если форма находится в дизайнере, это можно сразу увидеть, поскольку ее цвет изменяется на темно-серый. Это говорит о том, что данная форма стала родительской формой MDI. В нее по-прежнему можно добавлять элементы управления, но обычно это не рекомендуется.

Для того чтобы форма вела себя как дочерняя форма MDI, она должна знать, какая форма является для нее родительской. Это делается присвоением свойству `MdiParent` ссылки на родительскую форму. В примере все дочерние формы создаются методом `ShowMdiChild`, принимающим ссылку на дочернюю форму, которая должна быть пока-

зана. После установки свойства MdiParent на this, т.е. ссылку на форму mdiParent, дочерняя форма отображается. Код метода ShowMdiChild выглядит следующим образом:

```
private void ShowMdiChild(Form childForm)
{
  childForm.MdiParent = this;
  childForm.Show();
}
```

Одна из особенностей MDI-приложений состоит в том, что в любой момент времени может быть открыто несколько дочерних форм. Ссылка на текущую активную дочернюю форму может быть получена из свойства ActiveMdiChild родительской формы. Это демонстрирует пункт Windows-меню Current Active. Выбор его отобразит окно сообщения с именем формы и текстовым значением.

Дочерние формы могут быть размещены вызовом метода LayoutMdi. Он принимает параметр типа перечисления MdiLayout, возможные значения которого включают Cascade, TileHorizontal и TileVertical.

Заказные элементы управления

Применение элементов управления и компонентов — это то, что делает разработку с помощью такого пакета, как Windows Forms, настолько продуктивной. Возможность создавать собственные элементы управления и компоненты делает ее еще более продуктивной. Создавая элементы управления, можно инкапсулировать функциональность в пакеты, которые могут повторно использоваться вновь и вновь.

Элементы управления создаются различными способами. Можно начать с нуля, унаследовав новый класс от Control, ScrollableControl или ContainerControl. При этом нужно будет переопределить событие Paint, чтобы полностью выполнять все рисование, не говоря уже о добавлении функциональности, которую ваш элемент управления должен обеспечивать. Если элемент управления задуман как расширенная версия некоторого существующего, то потребуется объявить его наследником класса расширяемого элемента управления. Например, если необходим элемент управления TextBox, который будет менять цвет фона при установке свойства ReadOnly, то полное создание совершенно нового TextBox будет напрасной тратой времени. В этом случае просто унаследуйте новый класс от TextBox и переопределите свойство ReadOnly. Поскольку свойство ReadOnly класса TextBox не помечено словом override, в объявлении нужно использовать конструкцию new. В следующем коде показан пример такого переопределения свойства ReadOnly.

```
public new bool ReadOnly
{
  get { return base.ReadOnly;}
  set {
     if(value)
       this.BackgroundColor = Color.Red;
     else
       this.BackgroundColor = Color.FromKnowColor(KnownColor.Window);
     base.ReadOnly = value;
  }
}
```

Здесь видим, что get возвращает то, что и базовый объект. Опрос свойства никак не связан с нашей задачей изменения цвета фона при установке ReadOnly, поэтому мы просто передаем функциональность базовому объекту. При установке же свойства следует проверить переданное новое значение — false или true. Если оно равно true, то изменяем цвет (в данном случае на красный), а если false — устанавливаем

`BackgroundColor` в цвет по умолчанию. После этого значение передается базовому объекту, чтобы текстовое поле действительно стало доступным в соответствии со значением параметра. Как видим, переопределение одного простого свойства позволяет добавить новую функциональность к элементу управления.

Атрибуты элемента управления

Заказному элементу управления можно добавить атрибуты, которые расширяют его возможности времени проектирования. В табл. 31.5 описаны некоторые из наиболее часто используемых атрибутов.

Таблица 31.5. Часто используемые атрибуты заказных элементов управления

Имя атрибута	Описание
`BindableAttribute`	Используется во время проектирования для определения того, поддерживает ли свойство двустороннюю привязку данных.
`BrowsableAttribute`	Определяет, должно ли свойство отображаться в визуальном дизайнере.
`CategoryAttribute`	Определяет, под какой категорией свойство отображается в окне свойств. Использует предопределенные категории либо позволяет создавать новые. По умолчанию — `Misc`.
`DefaultEventAttribute`	Специфицирует событие по умолчанию для класса.
`DefaultPropertyAttribute`	Специфицирует свойство по умолчанию для класса.
`DefaultValueAttribute`	Специфицирует значение по умолчанию для свойства. Обычно это начальное значение.
`DecriptionAttribute`	Текст, появляющийся в нижней части окна дизайнера при выборе свойства.
`DesignOnlyAttribute`	Помечает свойство как редактируемое только в режиме проектирования.

Доступны также другие атрибуты, имеющие отношение к редактору, используемому в процессе проектирования, а также другие расширенные возможности времени проектирования. Почти всегда должны быть добавлены атрибуты `Category` и `Description`. Это помогает другим разработчикам, использующим элемент управления, лучше понимать назначение свойства. Чтобы добавить поддержку средства IntelliSense, следует добавлять XML-комментарий к каждому свойству, методу и событию. Когда элемент управления компилируется с опцией `/doc`, сгенерированный XML-файл комментариев представляет элементу управления поддержку IntelliSense.

Заказной элемент управления на базе `TreeView`

В этом разделе мы продемонстрируем разработку заказного элемента управления на базе стандартного `TreeView`. Он будет отображать файловую структуру дискового устройства. Мы добавим свойства, устанавливающие базовую или корневую папку и определяющие, как должны отображаться папки и файлы. Мы также воспользуемся различными атрибутами, о которых говорилось в предыдущем разделе.

Как и для любого другого проекта, необходимо определить требования к новому элементу управления. Вот список базовых требований, которые нужно будет реализовать:

- читать папки и файлы и отображать их пользователю;
- отображать структуру папок в древовидном иерархическом представлении;
- необязательно скрывать файлы от представления;
- определять папку, которая будет служить базовой корневой папкой;
- возвращать текущую выбранную папку;
- предоставлять возможность отложить загрузку файловой структуры.

Это может служить хорошей начальной точкой. Одно из требований удовлетворяется самим фактом наследования нового элемента управления от TreeView.

Элемент управления TreeView отображает данные в иерархическом виде. Он показывает текст, описывающий объект в списке, и необязательно — пиктограмму. Список может открываться и закрываться щелчком на объекте или нажатием клавиш со стрелками.

Создадим в Visual Studio .NET новый проект Windows Control Library с именем FolderTree и удалим из него класс UserControl1. Добавим новый класс и назовем его FolderTree. Поскольку FolderTree будет наследоваться от TreeView, изменим объявление класса

```
public class FolderTree
```

на

```
public class FolderTree : System.Windows.Forms.TreeView
```

В этот момент мы уже получаем полностью функциональный работающий элемент управления FolderTree. Он делает все, что может делать TreeView, но не более того. Элемент управления TreeView поддерживает коллекцию объектов TreeNode. Мы не можем загружать файлы и папки непосредственно в этот элемент управления. Существует несколько возможностей отобразить узлы TreeNode, которые загружаются в коллекцию Nodes, принадлежащую TreeView, на папки и файлы, которые они представляют.

Например, когда обрабатывается каждая папка, создается новый объект TreeNode, и его свойство text принимает значение имени папки или файла. Если в какой-то момент понадобится некоторая дополнительная информация о файле или папке, можно выполнить еще одно обращение к диску, чтобы получить эту информацию, или же сохранить дополнительные данные относительно файла или папки в свойстве Tag.

Другой метод заключается в создании нового класса-наследника TreeNode. В дополнение к его базовой функциональности можно добавить ряд новых методов и свойств. Именно таким путем мы и пойдем в этом примере. Это обеспечит возможность более гибкого дизайна. Если нужны новые свойства, их можно легко добавлять, не затрагивая существующего кода.

В элемент управления нужно загружать два типа объектов: папки и файлы. Каждый из них обладает своими собственными характеристиками. Например, каждая папка включают объект DirectoryInfo, который содержит дополнительную информацию, а файл — объект FileInfo. Из-за этого отличия мы будем использовать два разных класса для загрузки элемента управления TreeView: FileNode и FolderNode. Добавим эти два класса в проект, при этом унаследуем каждый из них от TreeNode. Ниже показан код FileNode.

```
namespace FormsSample.SampleControls
{
  public class FileNode : System.Windows.Forms.TreeNode
  {
     string _fileName = "";
     FileInfo _info;
     public FileNode(string fileName)
     {
        _fileName = fileName;
        _info = new FileInfo(_fileName);
        base.Text = _info.Name;
        if (_info.Extension.ToLower() == ".exe")
             this.ForeColor = System.Drawing.Color.Red;
     }
     public string FileName
     {
```

```
        get { return _fileName; }
        set { _fileName = value; }
      }
      public FileInfo FileNodeInfo
      {
        get { return _info; }
      }
    }
}
```

Имя обрабатываемого файла передается конструктору `FileNode`. В конструкторе создается объект `FileInfo` для файла и присваивается переменной-члену `_info`. Свойство `base.Text` устанавливается равным имени файла. Поскольку мы наследуем данный класс от `TreeNode`, это устанавливает свойство `Text` класса `TreeNode`. Это — текст, который будет отображен в элементе управления `TreeView`.

Далее добавляются два свойства для извлечения данных. `FileName` возвратит имя файла, а `FileNodeInfo` — объект `FileInfo`, описывающий файл.

Ниже представлен код класса `FolderNode`. Он очень похож на структуру класса `FileNode`. Разница состоит в том, что вместо свойства `FileInfo` здесь представлено свойство `DirectoryInfo`, а вместо `FileName` — `FolderPath`.

```
namespace FormsSample.SampleControls
{
  public class FolderNode : System.Windows.Forms.TreeNode
  {
     string _folderPath = "";
     DirectoryInfo _info;

     public FolderNode(string folderPath)
     {
       folderPath = folderPath;
       info = new DirectoryInfo(folderPath);
       this.Text = _info.Name;
     }
     public string FolderPath
     {
       get { return _folderPath; }
       set { _folderPath = value; }
     }
     public DirectoryInfo FolderNodeInfo
     {
       get { return _info; }
     }
  }
}
```

Теперь можно конструировать элемент управления `FolderTree`. В соответствии с требованиями, нам понадобится свойство для чтения и установки `RootFolder`. Также понадобится свойство `ShowFiles` для определения того, должны ли отображаться файлы в дереве. Свойство `SelectedFolder` вернет текущую выделенную папку в дереве. Таким образом, код элемента управления `FolderTree` будет выглядеть, как показано ниже:

```
using System;
using System.Windows.Forms;
using System.IO;
using System.ComponentModel;
namespace FolderTree
{
```

```
    /// <summary>
    /// Summary description for FolderTreeCtrl.
    /// </summary>
    public class FolderTree : System.Windows.Forms.TreeView
    {
       string _rootFolder = "";
       bool _showFiles = true;
       bool _inInit = false;
       public FolderTree()
       {
       }
       [Category("Behavior"),
         Description("Gets or sets the base or root folder of the tree"),
         DefaultValue("C:\\ ")]
       public string RootFolder
       {
         get {return _rootFolder;}
         set
         {
            _rootFolder = value;
            if(!_inInit)
            InitializeTree();
         }
       }
       [Category("Behavior"),
         Description("Indicates whether files will be seen in the list. "),
         DefaultValue(true)]
       public bool ShowFiles
       {
         get {return _showFiles;}
         set {_showFiles = value;}
       }
       [Browsable(false)]
       public string SelectedFolder
       {
         get
         {
            if(this.SelectedNode is FolderNode)
              return ((FolderNode)this.SelectedNode).FolderPath;
            return "";
         }
       }
    }
}
```

Здесь добавлены три свойства: `ShowFiles`, `SelectedFolder` и `RootFolder`. Обратите внимание на атрибуты этих свойств. Мы установили `Category`, `Description` и `DefaultValues` для `ShowFiles` и `RootFolder`. Эти два свойства появятся в браузере свойств в режиме проектирования. `SelectedFolder` на самом деле не имеет смысла во время проектирования, поэтому для него установлен атрибут `Browsable=false`. `SelectedFolder` не появится в браузере свойств, однако, поскольку это общедоступное свойство, оно появится в IntelliSense и будет доступным в коде.

Далее нужно инициализировать загрузку файловой системы. Инициализация элемента управления может оказаться непростой. Инициализация — как во время проектирования, так и во время выполнения — должна быть хорошо продуманной. Когда элемент управления устанавливается в конструкторе, он на самом деле запускается. Если, например, в конструкторе имеется обращение к базе данных, то это обращение выполнится, когда вы перетащите элемент управления в поле конструктора. В случае `FolderTree` это может повлечь за собой определенные последствия.

Вот как будет выглядеть метод, который в действительности загружает файлы:

```
private void LoadTree(FolderNode folder)
{
  string[] dirs = Directory.GetDirectories(folder.FolderPath);
  foreach(string dir in dirs)
  {
     FolderNode tmpfolder = new FolderNode(dir);
     folder.Nodes.Add(tmpfolder);
     LoadTree(tmpfolder);
  }
  if(_showFiles)
  {
     string[] files = Directory.GetFiles(folder.FolderPath);
     foreach(string file in files)
     {
       FileNode fnode = new FileNode(file);
       folder.Nodes.Add(fnode);
     }
  }
}
```

showFiles — булевская переменная-член класса, которая устанавливается через свойство ShowFiles. Если она равна true, файлы отображаются в дереве. Единственный вопрос — когда следует вызывать LoadFiles? Существует несколько вариантов. Этот метод может быть вызван при установке свойства RootFolder. В некоторых ситуациях это желательно, но только не во время проектирования. Вспомним, элементы управления в дизайнере "живые", поэтому, когда будет устанавливаться свойство RootFolder, наш элемент управления попытается выполнить загрузку из файловой системы.

Решить эту проблему можно, проверив свойство DesignMode. Оно возвращает true, если элемент управления находится в данный момент в дизайнере. Теперь можно записать код инициализации так:

```
private void InitializeTree()
{
  if(!this.DesignMode)
  {
     FolderNode rootNode = new FolderNode(_rootFolder);
     LoadTree(rootNode);
     this.Nodes.Clear();
     this.Nodes.Add(rootNode);
  }
}
```

Если элемент управления не пребывает в режиме проектирования, и rootFolder не равен пустой строке, начнется загрузка дерева. Первым делом создается и передается методу LoadTree корневой узел Root.

Другой способ состоит в том, чтобы реализовать общедоступный метод Init. В методе Init можно осуществить вызов LoadTree. Проблема такого способа связана с тем, что разработчик, использующий элемент управления, должен вызвать метод Init. Но в некоторых случаях это может оказаться приемлемым решением.

Для дополнительной гибкости может быть реализован интерфейс ISupportInitialize. Этот интерфейс имеет два метода — BeginInit и EndInit. Когда элемент управления реализует ISupportInitialize, методы BeginInit и EndInit вызываются автоматически в сгенерированном коде InitializeComponent. Это позволяет отложить процесс инициализации до того момента, когда будут установлены все свойства.

ISupportInitialize позволит коду в родительской форме также отложить инициализацию. Если в коде будет устанавливаться свойство RootNode, то вызов BeginInit позволит сначала установить свойство RootNode, как и все прочие свойства, или выполнить необходимые действия прежде, чем наш элемент управления загрузит файловую систему. Когда вызывается EndInit, произойдет инициализация. Вот как могут выглядеть BeginInit и EndInit:

```
#region ISupportInitialize Members
public void ISupportInitialize.BeginInit()
{
     _inInit = true;
}
public void ISupportInitialize.EndInit()
{
  if(_rootFolder != "")
  {
     InitializeTree();
  }
  _inInit = false;
}
#endregion
```

Все, что делается в методе BeginInit — это присваивание переменной _inInit значения true. Этот флаг применяется для того, чтобы определить, что элемент управления находится в процессе инициализации, и используется в свойстве RootFolder. Если свойство RootFolder устанавливается вне класса InitializeComponent, это значит, что дерево нуждается в повторной инициализации. В свойстве RootFolder мы проверяем, чему равно ли _inInit — true или false. Если true, то это значит, что нам не нужно проходить через процесс инициализации. Если же флаг _inInit равен false, мы вызываем InitializeTree. Можно также иметь общедоступный метод Init и в нем выполнять ту же задачу.

В методе EndInit мы проверим, находится ли наш элемент управления в режиме проектирования и что с _rootFolder ассоциирован корректный путь. Только после этого вызывается InitializeTree.

Чтобы придать нашему элементу управления профессиональный вид, необходимо добавить битовое изображение. Это будет пиктограмма, отображаемая в панели инструментов, когда элемент управления будет добавлен в проект. Битовое изображение должно быть размером 16×16 пикселей и иметь 16 цветов. Файл этого изображения можно создать в любом графическом редакторе, позволяющем выдержать такой размер и цветовую глубину. Это можно сделать даже в Visual Studio .NET. Для этого потребуется щелкнуть правой кнопкой мыши на проекте и выбрать в контекстном меню пункт Add New Item (Добавить новый элемент). Выберите в списке пункт Bitmap File (Файл битового изображения), чтобы открыть графический редактор. После создания файла битового изображения добавьте его в проект, убедившись, что он находится в том же пространстве имен и с тем же именем, как у нашего элемента управления.

И, наконец, установите свойство BuildAction для этого ресурса в Embedded Resource: щелкните правой кнопкой мыши на файле изображения в проводнике решений (Solution Explorer) и выберите в контекстном меню пункт Properties (Свойства). Выберите Embedded Resource для свойства BuildAction.

Чтобы протестировать разработанный элемент управления, создадим в том же решении проект TestHarness. Это должно быть простое приложение Windows Forms с единственной формой. В разделе ссылок добавим ссылку на проект FolderTreeCtl. В окне Toolbox добавим ссылку на FolderTreeCtl.DLL. После этого FolderTreeCtl должен появиться в панели инструментов с добавленным ранее изображением в виде

пиктограммы. Щелкнем на пиктограмме и перетащим ее на форму `TestHarness`. Установим `RootFolder` на одну из доступных папок и запустим решение.

Это, без сомнения, полноценный новый элемент управления. Но еще кое-что можно сделать для того, чтобы он был более полнофункциональным и готовым к промышленному применению. Например, ему можно добавить перечисленные ниже вещи.

- **Исключения** — если элемент управления пытается загрузить папку, к которой пользователь не имеет доступа, должно возбуждаться исключение.
- **Фоновую загрузку —** загрузка большого дерева папок может потребовать значительного времени. Усовершенствовав процесс инициализации так, чтобы воспользоваться преимуществами многопоточности для фоновой загрузки может быть хорошей идеей.
- **Кодирование с помощью цвета —** можно выводить имена файлов определенных типов в разных цветах.
- **Пиктограммы —** можно добавить элемент управления `ImageList`, в который поместить пиктограммы каждого загруженного файла или папки.

Пользовательские элементы управления

Пользовательские элементы управления — одно из наиболее мощных средств Windows Forms. Они позволяют инкапсулировать пользовательский интерфейс в симпатичные повторно используемые пакеты, которые можно легко подключать к различным проектам. Нередко организации имеют набор библиотек с часто используемыми пользовательскими элементами управления собственной разработки. Пользовательские элементы управления могут содержать в себе не только функциональность интерфейса пользователя, но также некоторые общие функции проверки достоверности данных — такие как форматирование телефонных номеров или номеров идентификаторов. Пользовательские элементы управления могут иметь встроенные в себя списки элементов, применяемых для быстрой загрузки в интерфейсные элементы — окна списков и комбинированные списки. Коды штатов и стран также входят в эту категорию. Включение как можно большего объема функциональности, не зависящей от конкретного приложения, в пользовательские элементы управления позволяют сделать их более полезными для данной организации.

В этом разделе мы создадим простой пользовательский элемент управления для ввода адреса. Мы также добавим различные события, которые обеспечат возможность привязки данных. Этот элемент управления будет состоять из текстовых полей для ввода двух строк адреса, города, штата и почтового кода.

Чтобы создать пользовательский элемент управления в текущем проекте, щелкните правой кнопкой мыши в Solution Explorer и выберите в контекстном меню пункт Add⇨Add New User Control (Добавить⇨Добавить новый пользовательский элемент управления). Можно также создать новый проект Control Library (Библиотека элемента управления) и добавить в него этот пользовательский элемент управления. После того, как новый пользовательский элемент управления будет запущен, мы увидим в дизайнере форму без рамок. Сюда мы поместим элементы управления, из которых будет состоять наш пользовательский элемент. Напомним, что пользовательский элемент управления состоит из одного или более элементов управления, добавленных в контейнер, поэтому все это напоминает создание формы. Для ввода адреса нам понадобится пять текстовых полей (`TextBox`) и три метки (`Label`). Расположить их можно любым подходящим образом (рис. 31.4).

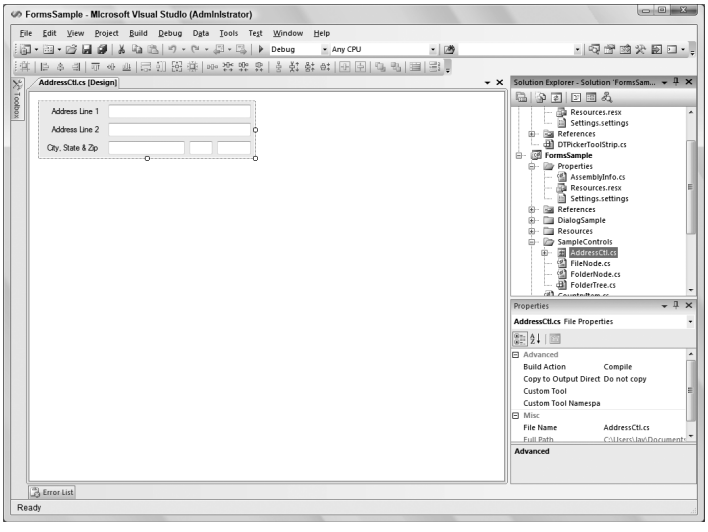

Рис. 31.4. Пример компоновки пользовательского элемента управления

Имена текстовых полей в нашем примере будут такими:

- `txtAddress1`
- `txtAddress2`
- `txtCity`
- `txtState`
- `txtZip`

После того, как текстовые поля будут размещены и получат корректные имена, добавим общедоступные свойства. У вас может возникнуть соблазн сделать составляющие текстовые поля общедоступными (`public`), а не приватными (`private`). Однако это плохая идея, поскольку нарушает принцип инкапсуляции функциональности, которую вы можете пожелать добавить к свойствам.

Ниже приведен код свойств, которые следует добавить к нашему пользовательскому элементу управления.

```
public string AddressLine1
{
  get{return txtAddress1.Text;}
  set{
     if(txtAddress1.Text != value)
     {
       txtAddress1.Text = value;
       if(AddressLine1Changed != null)
            AddressLine1Changed(this, EventArgs.Empty);
     }
  }
}
public string AddressLine2
{
  get{return txtAddress2.Text;}
  set{
     if(txtAddress2.Text != value)
     {
```

```
            txtAddress2.Text = value;
            if(AddressLine2Changed != null)
                  AddressLine2Changed(this, EventArgs.Empty);
         }
      }
   }
   public string City
   {
      get{return txtCity.Text;}
      set{
         if(txtCity.Text != value)
         {
            txtCity.Text = value;
            if(CityChanged != null)
                  CityChanged(this, EventArgs.Empty);
         }
      }
   }
   public string State
   {
      get{return txtState.Text;}
      set{
         if(txtState.Text != value)
         {
            txtState.Text = value;
            if(StateChanged != null)
                  StateChanged(this, EventArgs.Empty);
         }
      }
   }
   public string Zip
   {
      get{return txtZip.Text;}
      set{
         if(txtZip.Text != value)
         {
            txtZip.Text = value;
            if(ZipChanged != null)
                  ZipChanged(this, EventArgs.Empty);
         }
      }
   }
```

Экземпляры свойства `get` достаточно прямолинейны. Они возвращают значения соответствующих текстовых свойств элемента управления `TextBox`. Напротив, экземпляры свойства `set` выполняют немного больше работы. Все они функционируют единообразно. Сначала выполняется проверка того, изменяется ли значение свойства. Если новое значение совпадает со старым, то ничего не происходит. Если же новое значение отличается, то оно присваивается текстовому свойству элемента `TextBox`, после этого проверяется, существует ли экземпляр события. Речь идет о событиях изменения свойств, которые имеют специальный формат имени — *propertyname*`Changed`, где *propertyname* — имя свойства. В случае свойства `AddressLine1` событие называется `AddressLine1Changed`. События объявляются так, как показано ниже:

```
public event EventHandler AddressLine1Changed;
public event EventHandler AddressLine2Changed;
public event EventHandler CityChanged;
public event EventHandler StateChanged;
public event EventHandler ZipChanged;
```

Назначение этих событий — уведомлять привязку о том, что свойство изменилось. Выполнив проверку допустимости, привязка обеспечит синхронизацию нового значения свойства с объектом, к которому оно привязано. Еще один шаг должен быть выполнен для поддержки привязки. Изменение в текстовом поле, выполненное пользователем, не установит новое значение свойства непосредственно. Поэтому необходимо также возбудить событие *propertyname*Changed при изменении значения в текстовом поле. Простейший способ сделать это — отслеживать событие TextChanged элемента управления TextBox. В нашем примере будет только один обработчик событий TextChanged, и все текстовые поля будут использовать его. Имя элемента управления проверяется, чтобы увидеть, в каком именно элементе произошло изменение, и затем возбудить соответствующее событие propertyNameChanged. Ниже представлен код этого обработчика событий.

```
private void controls_TextChanged(object sender, System.EventArgs e)
{
  switch(((TextBox)sender).Name)
  {
     case "txtAddress1" :
       if(AddressLine1Changed != null)
            AddressLine1Changed(this, EventArgs.Empty);
       break;
     case "txtAddress2" :
       if(AddressLine2Changed != null)
            AddressLine2Changed(this, EventArgs.Empty);
       break;
     case "txtCity" :
       if(CityChanged != null)
            CityChanged(this, EventArgs.Empty);
       break;
     case "txtState" :
       if(StateChanged != null)
            StateChanged(this, EventArgs.Empty);
       break;
     case "txtZip" :
       if(ZipChanged != null)
            ZipChanged(this, EventArgs.Empty);
       break;
  }
}
```

Здесь мы применяем простой оператор switch для определения того, какое из текстовых полей вызвало событие TextChanged. После этого выполняется проверка, чтобы убедиться, что событие корректно и не равно null.

Затем возбуждается событие Changed. Обратите внимание, что при этом отправляется пустой EventArgs (EventArgs.Empty). Тот факт, что эти события были добавлены к свойствам для поддержки привязки данных, не значит, что единственный способ использования нашего элемента управления лежит через привязку данных. Они добавлены так, чтобы пользовательский элемент управления мог использовать привязку в случае ее наличия. Это лишь один из способов обеспечения максимальной гибкости пользовательского элемента управления, чтобы его можно было применять в самых разнообразных ситуациях.

Памятуя о том, что пользовательский элемент управления — это, по сути, элемент управления с дополнительными возможностями, все связанное с применением его в среде дизайнера, что мы обсуждали в предыдущем разделе, также применимо и здесь. Инициализация пользовательского элемента управления может привести к тем же

последствиям, что мы видели в примере FolderTree. При проектировании пользовательских элементов управления необходимо позаботиться о том, чтобы избежать обращения к хранилищам данных, которые могут оказаться недоступными разработчикам, использующим ваши элементы управления.

Еще одна вещь, подобная созданию элементов управления — это атрибуты, применяемые к пользовательским элементам управления. Общедоступные свойства и методы пользовательского элемента управления отображаются в окне свойств, когда он помещается в дизайнер. В примере с адресом неплохо будет добавить атрибуты Category, Description и DefaultValue к свойствам адреса. Можно создать новую категорию AddressData со значением по умолчанию "". Ниже показан пример применения этих атрибутов к свойству AddressLine1.

```
[Category("AddressData"),
     Description("Gets or sets the AddressLine1 value"),
     DefaultValue("")]
public string AddressLine1
{
   get{return txtAddress1.Text;}
   set{
     if(txtAddress1.Text != value)
     {
          txtAddress1.Text = value;
          if(AddressLine1Changed != null)
          AddressLine1Changed(this, EventArgs.Empty);
     }
   }
}
```

Как видите, все, что нужно сделать для добавления новой категории — это установить текст в атрибуте Category. Это автоматически добавляет новую категорию.

Помимо того, что было описано выше, остается большой простор для совершенствования. Например, можно включить список названий штатов и их сокращений. Вместо одного свойства штата пользовательский элемент управления можно расширить свойствами, позволяющими вводить и хранить как имя штата, так и его аббревиатуру. Также можно добавить обработку исключений. Можно подумать о том, должно ли свойство AddressLine2 быть необязательным, куда следует вводить номер квартиры и комнаты и тому подобное.

Резюме

В этой главе было дано представление о базовых принципах построения клиентских Windows-приложений. Были описаны стандартные элементы управления, образующие иерархию классов пространства имен Windows.Forms, и рассмотрены различные их свойства и методы.

Также здесь было продемонстрировано, как можно создать базовый заказной элемент управления, а также базовый пользовательский элемент управления. Мощь и гибкость, обеспечиваемую ими, переоценить невозможно. Благодаря возможностям создания собственных наборов пользовательских элементов управления, значительно облегчается задача разработки и тестирования клиентских Windows-приложений, поскольку вновь и вновь можно использовать одни и те же тщательно протестированные компоненты.

В следующей главе речь пойдет о том, каким образом можно привязать источник данных к элементам управления, расположенным на форме. За счет этого вы получите возможность создавать формы, которые будут автоматически обновлять данные, и сможете синхронизировать данные на форме.