ДОПОЛНИТЕЛЬНАЯ ГЛАВА 4

# Программирование с использованием элементов управления Windows Forms

В этой главе предлагается путеводитель по элементам управления, определенным в пространстве имен `System.Windows.Forms`. Ранее у вас была возможность поработать с некоторыми элементами управления, встроенными в главную форму, такими как `MenuStrip`, `ToolStrip` и `StatusStrip`. Однако в настоящей главе вы познакомитесь с различными типами, которые обычно размещаются в клиентской области `Form` (например, `Button`, `MaskedTextBox`, `WebBrowser`, `MonthCalendar`, `TreeView` и им подобными). Изучив основные виджеты пользовательского интерфейса, вы затем займетесь изучением процесса построения специальных пользовательских элементов управления Windows Forms, которые интегрируются в среду разработки Visual Studio.

Затем мы исследуем процесс построения пользовательских диалоговых окон и роль *наследования форм*, которое позволяет строить иерархии взаимосвязанных типов форм. Завершится эта глава дискуссией о поведении *стыковки* (docking) и *прикрепления* (anchoring) семейства типов GUI, а также роли типов `FlowControlPanel` и `TableControlPanel`, предлагаемых в .NET 2.0.

## Мир элементов управления Windows Forms

Пространство имен `System.Windows.Forms` содержит множество типов, представляющих распространенные виджеты GUI, обычно используемые для того, чтобы позволить реагировать на пользовательский ввод в приложении Windows Forms. Многие из этих элементов управления, с которыми придется работать повседневно (`Button`, `TextBox` и `Label`), достаточно интуитивно понятны. Другие, более экзотические элементы управления и компоненты (вроде `TreeView`, `ErrorProvider` и `TabControl`), требуют некоторых дополнительных объяснений.

Как вам известно из главы 19, тип `System.Windows.Forms.Control` является базовым классом для всех унаследованных виджетов. Напомним, что `Control` обеспечивает возможность обработки событий мыши и клавиатуры, установки физических размеров и положения виджета с использованием различных свойств (`Height`, `Width`, `Left`,

Right, Location и т.п.), манипулирования цветами фона и переднего плана, установки активного шрифта/курсора и поведения стыковки (объясняется в конце этой главы).

В процессе чтения этой главы помните, что рассматриваемые вами виджеты значительную долю своей функциональности наследуют от базового класса Control. Поэтому мы сосредоточимся (более или менее) на уникальных членах каждого виджета. Имейте в виду, что здесь не предпринимается попытка полностью описать все члены абсолютно всех элементов управления (эту задачу решает документация по .NET Framework 2.0 SDK). Однако будьте уверены, что по прочтении этой главу, вы не будете иметь проблем с пониманием виджетов, которые не были описаны непосредственно.

---

**На заметку!** Windows Forms предоставляет множество элементов управления, которые позволяют отображать реляционные данные (DataGridView, BindingResource и т.п.). Некоторые из этих ориентированных на данные элементов управления рассматриваются при описании ADO.NET.

---

# Добавление элементов управления к формам вручную

Независимо от типа элемента управления, который выбран для помещения в Form, всегда должен быть выполнен один и тот же набор шагов. Прежде всего, понадобится определить переменные-члены, представляющие элементы управления. Затем внутри конструктора Form (или внутри вспомогательного метода, вызываемого конструктором), необходимо настроить внешний вид и поведение каждого элемента управления, используя свойства, методы и события. И, наконец (что наиболее важно), как только элемент управления приведен в начальное состояние, его потребуется добавить во внутреннюю коллекцию элементов управления Form, используя унаследованное свойство Controls. Если забыть об этом заключительном шаге, виджет не будет видим во время выполнения.

Чтобы проиллюстрировать процесс добавления элементов управления к Form, начнем с построения типа формы “без мастеров”, используя только текстовый редактор и компилятор командной строки C#. Создайте новый файл C# по имени ControlsByHand.cs со следующим кодом класса MainWindow:

```
using System;
using System.Drawing;
using System.Windows.Forms;
namespace ControlsByHand
{
   class MainWindow : Form
   {
      // Переменные-члены формы, хранящие ссылки на виджеты.
      private TextBox firstNameBox = new TextBox();
      private Button btnShowControls = new Button();
      public MainWindow()
      {
         // Сконфигурировать Form.
         this.Text = "Simple Controls";
         this.Width = 300;
         this.Height = 200;
         CenterToScreen();
         // Добавить новое текстовое поле к Form.
         firstNameBox.Text = "Hello";
         firstNameBox.Size = new Size(150, 50);
         firstNameBox.Location = new Point(10, 10);
         this.Controls.Add(firstNameBox);
```

```
            // Добавить новую кнопку к Form.
            btnShowControls.Text = "Click Me";
            btnShowControls.Size = new Size(90, 30);
            btnShowControls.Location = new Point(10, 70);
            btnShowControls.BackColor = Color.DodgerBlue;
            btnShowControls.Click += new EventHandler(btnShowControls_Clicked);
            this.Controls.Add(btnShowControls);
        }
        private void btnShowControls_Clicked(object sender, EventArgs e)
        {
            // Вызвать ToString() на каждом элементе управления
            // из коллекции Controls формы.
            string ctrlInfo= "";
            foreach (Control c in this.Controls)
            {
                ctrlInfo += string.Format("Control: {0}\n",
                c.ToString());
            }
            MessageBox.Show(ctrlInfo, "Controls on Form");
        }
    }
}
```

Теперь добавьте второй класс в пространство имен `ControlsByHand`, реализующий метод `Main()`:

```
class Program
{
    public static void Main(string[] args)
    {
        Application.Run(new MainWindow());
    }
}
```

После этого скомпилируйте файл #C в командной строке с помощью следующей команды:

```
csc /target:winexe *.cs
```

Когда вы запустите программу и щелкнете на кнопке формы, то увидите окно сообщения с перечислением всех элементов формы (рис. 4.1).

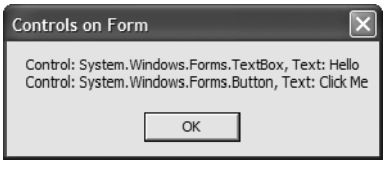


**Рис. 4.1.** Взаимодействие с коллекцией элементов управления `Form`

## Тип `Control.ControlCollection`

Хотя процесс добавления нового виджета в `Form` довольно прост, свойство `Controls` имеет смысл рассмотреть более подробно. Это свойство возвращает ссылку на вложенный класс по имени `ControlCollection`, определенный внутри класса `Control`. Этот вложенный тип `ControlCollection` поддерживает вхождения для каждого виджета, помещенного в `Form`. Всякий раз, когда нужно пройтись по списку дочерних элементов, необходимо получать ссылку на эту коллекцию:

```
// Получить доступ к вложенной коллекции ControlCollection для данной формы.
Control.ControlCollection coll = this.Controls;
```

Имея ссылку на эту коллекцию, можно манипулировать его содержимым, используя члены, перечисленные в табл. 4.1.

**Таблица 4.1. Члены `ControlCollection`**

| Член | Назначение |
|---|---|
| `Add()`<br>`AddRange()` | Используется для вставки нового типа-наследника `Control` (или массива типов) в коллекцию. |
| `Clear()` | Удалить все вхождения в коллекцию. |
| `Count` | Возвращает количество элементов в коллекции. |
| `GetEnumerator()` | Возвращает интерфейс `IEnumerator` для коллекции. |
| `Remove()`<br>`RemoveAt()` | Используются для удаления элемента управления из коллекции. |

Учитывая, что `Form` поддерживает коллекцию своих элементов управления, в Windows Forms очень просто создавать, удалять или иным образом манипулировать визуальными элементами. Например, предположим, что вы хотите сделать недоступными все элементы типа `Button` в данной форме (или сделать что-то другое, например, изменить цвет фона элементов `TextBox`). Для этого можно использовать ключевое слово C# `is` для определения того, кто есть кто, и соответствующего изменения состояния требуемых виджетов:

```
private void DisableAllButtons()
{
    foreach (Control c in this.Controls)
    {
        if (c is Button)
            ((Button)c).Enabled = false;
    }
}
```

**Исходный код.** Проект `ControlsByHand` включен в подкаталог `Bonus Chapter 4`.

# Добавление элементов управления в `Form` с использованием Visual Studio

Теперь, когда вы понимаете процесс ручного добавления элементов управления в форму, давайте посмотрим, как Visual Studio может помочь автоматизировать этот процесс. Создайте новый проект Windows Application и назовите его произвольным образом. Подобно процессу проектирования меню, панелей инструментов или линеек состояния, когда вы перетаскиваете элемент управления из панели Toolbox в дизайнер форм, IDE-среда реагирует на это автоматическим добавлением правильной переменной-члена к файлу `*.Designer.cs`. К тому же, при проектировании внешнего вида и поведения виджета с применением окна Properties (Свойства), соответствующие изменения кода добавляются в функцию-член `InitializeComponent()` (также расположенную в файле `*.Designer.cs`).

**На заметку!** Напомним, что окно Properties также позволяет обрабатывать события определенного элемента управления, для чего нужно щелкнуть на пиктограмме с изображением молнии. Просто выберите виджет в раскрывающемся списке и введите имя метода, который будет вызван в ответ на возникновение интересующего события (или просто дважды щелкните на событии, чтобы сгенерировать имя обработчика события по умолчанию).

Предположим, что вы добавили элементы типа `TextBox` и типа `Button` в дизайнере форм. Обратите внимание, что при перемещении элемента управления в дизайнере IDE-среда Visual Studio предоставляет визуальные подсказки относительно промежутков и выравнивания текущего виджета (рис. 4.2).

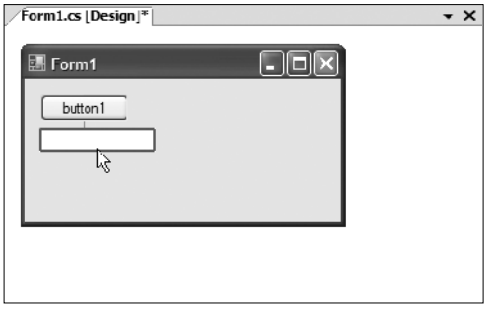


**Рис. 4.2.** Подсказки по промежуткам и выравниванию

Поместив `Button` и `TextBox` в дизайнер, загляните в сгенерированный код метода `InitializeComponent()`. Здесь вы увидите, что новые элементы создаются ключевым словом `new` и автоматически вставляются в `ControlCollection` формы (вдобавок в любым установкам, которые были сделаны в окне Properties):

```
private void InitializeComponent()
{
    this.btnMyButton = new System.Windows.Forms.Button();
    this.txtMyTextBox = new System.Windows.Forms.TextBox();
...
    // MainWindow
    //
    ...
    this.Controls.Add(this.txtMyTextBox);
    this.Controls.Add(this.btnMyButton);
...
}
```

Как видите, такой инструмент, как Visual Studio, просто экономит время на ввод кода. Рассмотрим некоторые наиболее интересные аспекты следующих базовых элементов пользовательского интерфейса:

- `Label`, `TextBox` и `MaskedTextBox`
- `Button`
- `CheckBox`, `RadioButton` и `GroupBox`
- `CheckedListBox`, `ListBox` и `ComboBox`

Как только вы освоитесь с этими базовыми типами-наследниками `Control`, мы переключим внимание на более экзотические виджеты — вроде `MonthCalendar`, `TabControl`, `TrackBar`, `WebBrowser` и т.д.

## Упражнения с Label

Элемент управления Label может содержать информацию, доступную только для чтения (текстовую или на основе изображения), которая поясняет пользователям назначение других элементов управления. Предположим, что вы создали новый проект Windows Forms под названием LabelsAndTextBoxes. Определите метод по имени CreateLabelControl в типе-наследнике Form, который создает и конфигурирует элемент типа Label, а затем добавляет его в коллекцию элементов управления Form:

```
private void CreateLabelControl()
{
    // Создать и конфигурировать Label.
    Label lblInstructions = new Label();
    lblInstructions.Name = "lblInstructions";
    lblInstructions.Text = "Please enter values in all the text boxes";
    lblInstructions.Font = new Font("Times New Roman", 9.75F, FontStyle.Bold);
    lblInstructions.AutoSize = true;
    lblInstructions.Location = new System.Drawing.Point(16, 13);
    lblInstructions.Size = new System.Drawing.Size(240, 16);

    // Добавить в коллекцию элементов управления Form.
    Controls.Add(lblInstructions);
}
```

Вызвав эту вспомогательную функцию внутри конструктора формы, вы увидите, что сообщение появится в верхней части главного окна:

```
public MainWindow()
{
    InitializeComponent();
    CreateLabelControl();
    CenterToScreen();
}
```

В отличие от большинства других виджетов, элемент управления Label не может принимать фокус по нажатию клавиши <Tab>. Тем не менее, в .NET 2.0 можно создавать *мнемонические ключи* для любого Label, устанавливая свойство UseMnemonic в true (что является установкой по умолчанию). После этого в свойстве Text элемента Label можно определить горячую клавишу (с помощью символа &), используемую для перехода к следующему элементу управления по клавише <Tab>.

---

**На заметку!** Далее в этой главе вы узнаете больше о конфигурировании порядка обхода по клавише табуляции, а пока знайте, что порядок обхода по <Tab> элемента управления устанавливается свойством TabIndex. По умолчанию TabIndex элемента управления устанавливается в соответствии с порядком, в котором он добавлялся в дизайнере формы. Таким образом, если добавить Label, за которым следует TextBox, то для Label свойство TabIndex устанавливается в 0, а для TextBox — в 1.

---

Для иллюстрации давайте воспользуемся дизайнером форм для построения пользовательского интерфейса, включающего набор из трех Label и трех TextBox (оставьте достаточно места в верхней части формы под элемент Label, динамически генерируемый в методе CreateLabelControl()). Обратите внимание на рис. 4.3, что каждая метка включает подчеркнутую букву, идентифицированную символом & в значении, присвоенном свойству Text (как известно, символы, специфицированные добавлением &, указывают клавишу, которая позволяет активизировать соответствующий элемент нажатием сочетания <Alt+клавиша>).

Запустите проект; теперь можно переключаться между элементами TextBox, нажимая сочетания клавиш <Alt+p>, <Alt+m> и <Alt+u>.

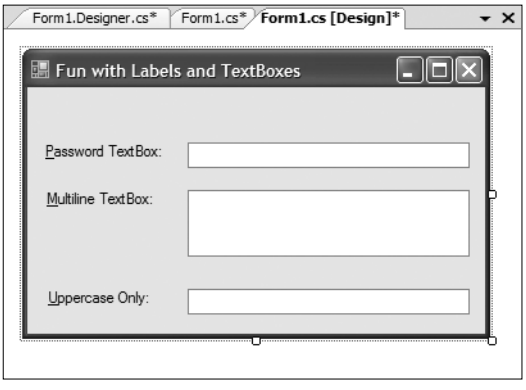


**Рис. 4.3.** Назначение мнемоник элементам управления

## Упражнения с TextBox

В отличие от Label, элемент управления TextBox обычно не является доступным только для чтения (хотя может быть таковым, если установить свойство ReadOnly в true), и применяется для того, чтобы позволить пользователю вводить текстовые данные. Тип TextBox может быть сконфигурирован для хранения одной строки или множества строк текста, настроен для ввода паролей (с отображением всех символов в виде, например, *), и может также поддерживать линейки прокрутки в случае многострокового текста. В дополнение к поведению, унаследованному от базовых классов, TextBox определяет несколько свойств, представляющих особый интерес (см. табл. 4.2).

**Таблица 4.2. Свойства TextBox**

| Свойство | Назначение |
|---|---|
| AcceptsReturn | Получает или устанавливает значение, указывающее на то, создает многострочковый TextBox новую строку текста или же активизирует "кнопку по умолчанию" формы. |
| CharacterCasing | Получает или устанавливает режим модификации регистра вводимых символов. |
| PasswordChar | Получает или устанавливает символ, используемый для маскировки символов в однострочковом TextBox, используемом для ввода паролей. |
| ScrollBars | Получает или устанавливает наличие линеек прокрутки в многострочковом элементе управления TextBox. |
| TextAlign | Получает или устанавливает режим выравнивания текста в элементе управления TextBox с использованием перечисления HorizontalAlignment. |

Чтобы проиллюстрировать некоторые аспекты TextBox, давайте сконфигурируем три элемента TextBox в текущей форме. Первый из них (txtPassword) должен быть настроен как текстовое поле для ввода паролей, т.е. вводимые в него символы не должны быть непосредственно видимы, а вместо этого маскироваться одним символом, заданным в свойстве PasswordChar.

Второй элемент TextBox (txtMultiline) будет многостроковой текстовой областью, сконфигурированной для приема обработки клавиши <Enter> и отображения вертикальной линейки прокрутки при вводе текста за пределами видимой области TextBox. И, наконец, третий TextBox (txtUpperCase) будет настроен на трансляцию введенных символьных данных в верхний регистр.

Сконфигурируйте каждый TextBox соответствующим образом в окне Properties, воспользовавшись в качестве руководства следующей (частичной) реализацией InitializeComponent():

```
private void InitializeComponent()
{
...
   // txtPassword
   //
  this.txtPassword.PasswordChar = '*';
...
   // txtMultiline
   //
   this.txtMultiline.Multiline = true;
   this.txtMultiline.ScrollBars = System.Windows.Forms.ScrollBars.Vertical;
...
   // txtUpperCase
   //
   this.txtUpperCase.CharacterCasing =
      System.Windows.Forms.CharacterCasing.Upper;
...
}
```

Обратите внимание, что свойство ScrollBars получает значение из перечисления ScrollBars, которое определяет следующий список значений:

```
public enum System.Windows.Forms.ScrollBars
{
   Both, Horizontal, None, Vertical
}
```

Свойство CharacterCasing работает в сочетании с перечислением CharacterCasing, определенным так:

```
public enum System.Windows.Forms.CharacterCasing
{
   Normal, Upper, Lower
}
```

Теперь предположим, что на форму помещен элемент Button (по имени btnDisplay Data) и к нему добавлен обработчик события Click. Реализация этого метода просто отображает значение каждого TextBox внутри окна сообщения:

```
private void btnDisplayData_Click(object sender, EventArgs e)
{
   // Извлечь данные из всех элементов TextBox.
   string textBoxData = "";
   textBoxData += string.Format("MultiLine: {0}\n", txtMultiline.Text);
   textBoxData += string.Format("\nPassword: {0}\n", txtPassword.Text);
   textBoxData += string.Format("\nUppercase: {0}\n", txtUpperCase.Text);
   // Отобразить все данные.
   MessageBox.Show(textBoxData, "Here is the data in your TextBoxes");
}
```

На рис. 4.4 показан один возможный сеанс (чтобы видеть мнемоники меток, необходимо удерживать нажатой клавишу <Alt>).

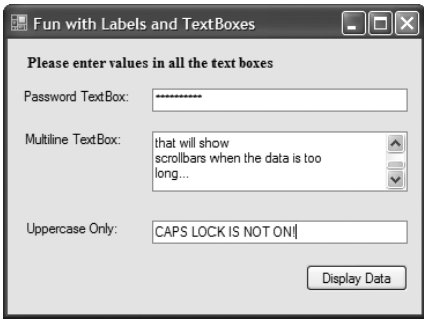


**Рис. 4.4.** Многоликий тип `TextBox`

На рис. 4.5 можно видеть результат щелчка на элементе `Button`.


**Рис. 4.5.** Извлечение значений из элементов `TextBox`

## Упражнения с `MaskedTextBox`

С выходом .NET 2.0 стало доступным *маскированное* текстовое поле, позволяющее специфицировать правильную последовательность символов, которые будут приняты областью ввода (номер карточки социального страхования, номер телефона с кодом региона, почтовый индекс и т.п.). Маска, по которой нужно проверять ввод (называемая *маскирующим выражением*), устанавливается с использованием специфических лексем, встраиваемых в строковый литерал. После создания маскирующее выражение присваивается свойству `Mask`. В табл. 4.3 документированы некоторые допустимые маскирующие лексемы.

**Таблица 4.3. Маскирующие лексемы**

| Лексема маски | Назначение |
|---|---|
| `0` | Представляет обязательную десятичную цифру от 0 до 9. |
| `9` | Представляет необязательную десятичную цифру или пробел. |
| `L` | Представляет букву A–Z (в верхнем или нижнем регистре). |
| `?` | Представляет необязательную букву A–Z (в верхнем или нижнем регистре). |
| `,` | Представляет разделитель тысяч. |
| `:` | Представляет разделитель времени. |
| `/` | Представляет разделитель даты. |
| `$` | Представляет символ валюты. |

**На заметку!** Символы, воспринимаемые `MaskedTextBox`, не отображаются непосредственно на синтаксис регулярных выражений. Хотя .NET предоставляет пространства имен для работы с регулярными выражениями (`System.Text.RegularExpressions` и `System.Web.RegularExpressions`), `MaskedTextBox` применяет синтаксис на основе унаследованного элемента управления `MaskedEdit` из VB6 COM.

В дополнение к свойству `Mask`, класс `MaskedtextBox` имеет дополнительные члены, которые определяют то, как элемент управления должен реагировать на ввод пользователем недопустимых данных. Например, `BeepOnError` заставит элемент издать звуковой сигнал, когда вводится символ, не соответствующий маске, и предотвращает дальнейшую обработку недопустимого символа.

Чтобы проиллюстрировать применение `MaskedTextBox`, добавьте еще один элемент `Label` и `MaskedTextBox` к текущей форме. Хотя шаблон маски можно строить прямо в коде, окно Properties предлагает кнопку с многоточием для ввода свойства `Mask`, при нажатии которой появится диалоговое окно с множеством предопределенных масок (рис. 4.6).

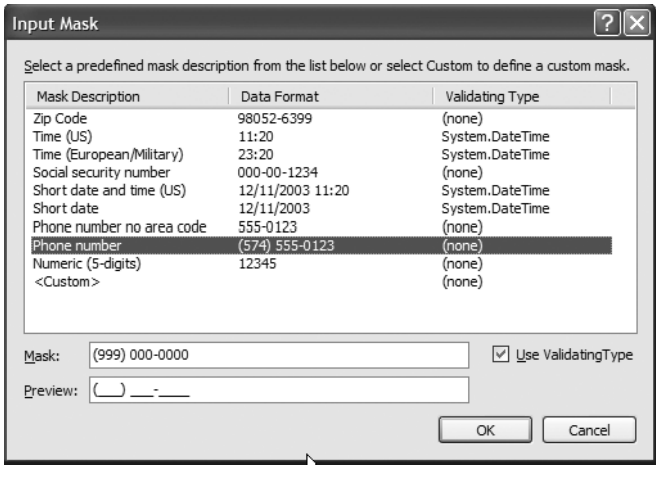


**Рис. 4.6.** Предопределенные значения масок для свойства `Mask`

Найдите подходящий шаблон маски (вроде Phone number (Телефонный номер)), включите свойство `BeepOnError` и снова запустите программу для тестирования. Буквенные символы (в случае выбора маски Phone number) ввести не удастся.

Как и можно было ожидать, `MaskedTextBox` на протяжении своего времени жизни посылает разнообразные события, одно из которых, `MaskedInputRejected`, инициируется, когда конечный пользователь осуществляет ошибочный ввод. Обработайте это событие в окне Properties; обратите внимание, что второй входящий аргумент сгенерированного обработчика события имеет тип `MaskInputRejectedEventArgs`. Этот тип поддерживает свойство по имени `RejectionHint`, содержащее краткое описание ошибки ввода. Для целей данного примера просто отобразите ошибку в заголовке формы:

```
private void txtMaskedTextBox_MaskInputRejected(object sender,
   MaskInputRejectedEventArgs e)
{
   this.Text = string.Format("Error: {0}", e.RejectionHint);
}
```

Чтобы гарантировать, что эта ошибка не будет отображаться при вводе корректных данных, обработайте событие KeyDown в MaskedTextBox и реализуйте обработчик события так, чтобы он сбрасывал заголовок формы в значение по умолчанию:

```
private void txtMaskedTextBox_KeyDown(object sender, KeyEventArgs e)
{
   this.Text = "Fun with Labels and TextBoxes";
}
```

**Исходный код.** Проект LabelsAndTextBoxes включен в подкаталог Bonus Chapter 4.

## Упражнения с Button

Задача типа System.Windows.Forms.Button состоит в том, чтобы обеспечить возможность пользователю подтверждать ввод, обычно в результате щелчка кнопкой мыши или нажатия какой-либо клавиши на клавиатуре. Класс Button — прямой наследник абстрактного типа по имени ButtonBase, который обеспечивает большую часть ключевого поведения своим типам-наследникам (таким как CheckBox, Radioutton и Button). В табл. 4.4 перечислены некоторые основные свойства ButtonBase.

**Таблица 4.4. Свойства ButtonBase**

| Свойство | Назначение |
|---|---|
| FlatStyle | Получает или устанавливает "плоский" стиль отображения элемента управления Button, используя члены перечисления FlatStyle. |
| Image | Конфигурирует (дополнительное) изображение, отображаемое где-то в пределах типа-наследника ButtonBase. Напомним, что класс Control также определяет свойство BackgroundImage, используемое для визуализации изображения на всей поверхности виджета. |
| ImageAlign | Устанавливает выравнивание изображения на элементе управления Button с использованием перечисления ContentAlignment. |
| TextAlign | Получает или устанавливает выравнивание текста на элементе управления Button с использованием перечисления ContentAlignment. |

Свойство TextAlign класса ButtonBase исключительно упрощает позиционирование текста почти в любом положении. Для установки позиции метки элемента Button применяйте перечисление ContentAlignment (определенное в пространстве имен System.Drawing). Как вы увидите, то же самое выравнивание может быть использовано для размещения дополнительного изображения на элементе типа Button:

```
public enum System.Drawing.ContentAlignment
{
   BottomCenter, BottomLeft, BottomRight,
   MiddleCenter, MiddleLeft, MiddleRight,
   TopCenter, TopLeft, TopRight
}
```

FlatStyle — еще одно интересное свойство. Оно используется для управления общим внешним видом и поведением элемента управления Button и может принимать любое значение из перечисления FlatStyle (определенного в пространстве имен System.Windows.Forms):

```
public enum System.Windows.Forms.FlatStyle
{
   Flat, Popup, Standard, System
}
```

Чтобы проиллюстрировать работу с типом Button, создайте новое приложение Windows Forms по имени Buttons. В дизайнере форм добавьте три элемента Button (btnFlat, btnPopup и btnStandard) и соответствующим образом установите для каждого свойство FlatStyle (FlatStyle.Flat, FlatStyle.Popup или FlatStyle.Standard). Кроме того, установите свойства Text каждой кнопки в соответствующее значение и обработайте событие Click для кнопки btnStandard. Когда пользователь щелкнет на этой кнопке, вы измените положение ее текста с использованием свойства TextAlign.

Теперь добавьте последний четвертый элемент Button (по имени btnImage), поддерживающий фоновое изображение (устанавливаемое через свойство BackgroundImage) и маленькую пиктограмму с "бычьим глазом" (установленную через свойство Image), которая также будет динамически перемещена при щелчке на btnStandard. Свойствам BackgroundImage и Image можно присвоить любые файлы изображений; использованные в примере файлы изображений можно найти в загружаемом коде для этой главы.

Учитывая, что дизайнер генерирует весь необходимый подготовительный код пользовательского интерфейса внутри InitializeComponent(), в остальном коде используется перечисление ContentAlignment для изменения положения текста на btnStandard и пиктограммы на btnImage. Обратите внимание в следующем коде, что для получения списка имен из перечисления ContentAlignment вызывается статический метод Enum.GetValues():

```
partial class MainWindow : Form
{
    // Используется для хранения текущего значения выравнивания текста.
    ContentAlignment currAlignment = ContentAlignment.MiddleCenter;
    int currEnumPos = 0;
    public MainWindow()
    {
        InitializeComponent();
        CenterToScreen();
    }
    private void btnStandard_Click (object sender, EventArgs e)
    {
        // Получить все возможные значения перечисления ContentAlignment.
        Array values = Enum.GetValues(currAlignment.GetType());

        // Сдвинуть текущую позицию перечисления.
        // и проверить на выход за границу.
        currEnumPos++;
        if(currEnumPos >= values.Length)
            currEnumPos = 0;

        // Получить текущее значение перечисления.
        currAlignment = (ContentAlignment)Enum.Parse(currAlignment.GetType(),
            values.GetValue(currEnumPos).ToString());

        // Нарисовать значение перечисления и выровнять текст на btnStandard.
        btnStandard.TextAlign = currAlignment;
            btnStandard.Text = currAlignment.ToString();

        // Теперь установить местоположение пиктограммы на btnImage.
        btnImage.ImageAlign = currAlignment;
    }
}
```

Запустите программу. Щелчок на средней кнопке приводит к установке ее текста в текущее имя и позицию переменной-члена currAlignment. К тому же положение пиктограммы внутри btnImage изменится в зависимости от того же значения. Вывод показан на рис. 4.7.


**Рис. 4.7.** Многоликий тип `Button`

**Исходный код.** Проект `Buttons` включен в подкаталог `Bonus Chapter 4`.

## Упражнения с `CheckBox`, `RadioButton` и `GroupBox`

Пространство имен `System.Windows.Forms` определяет множество других типов, расширяющих `ButtonBase`, в частности, `CheckBox` (флажок, который может поддерживать три возможных состояния) и `RadioButton` (переключатель, который может быть либо выбран, либо нет). Подобно `Button`, эти типы также получают большую часть функциональности от базового класса `Control`. Однако каждый класс определяет некоторую дополнительную функциональность. Рассмотрим основные свойства виджета `CheckBox`, описанные в табл. 4.5.

**Таблица 4.5. Свойства `CheckBox`**

| Свойство | Назначение |
|---|---|
| `Appearance` | Конфигурирует внешний вид элемента управления `CheckBox`, используя перечисление `Appearance`. |
| `AutoCheck` | Получает или устанавливает значение, указывающее, изменяются ли автоматически значения `Checked` или `CheckState` и внешний вид `CheckBox` при щелчке. |
| `CheckAlign` | Получает или устанавливает горизонтальное и вертикальное выравнивание метки элемента управления `CheckBox` с использованием перечисления `ContentAlignment` (подобно типу `Button`). |
| `Checked` | Возвращает булевское значение, представляющее состояние `CheckBox` (отмечен или не отмечен). Если свойство `ThreeState` установлено в `true`, свойство `Checked` возвращает `true` как при помеченном, так и неопределенном состоянии. |
| `CheckState` | Получает или устанавливает значение, указывающее, отмечен ли `CheckBox`, используя перечисление `CheckState` вместо булевского значения. |
| `ThreeState` | Конфигурирует поддержку `CheckBox` трех состояний выбора (как специфицировано в перечислении `CheckState`) вместо двух. |

Тип RadioButton требует небольших комментариев, учитывая, что он представляет собой некоторую вариацию типа CheckBox. Фактически члены RadioButton почти идентичны членам типа CheckBox. Единственное существенное отличие связано с событием CheckedChanged, которое (что не удивительно) инициируется при изменении значения Checked. К тому же тип RadioButton не поддерживает свойство ThreeState, так как RadioButton должен быть либо включен, либо выключен.

Обычно несколько элементов RadioButton логически иди физически группируются вместе, чтобы работать как единое целое. Например, пусть имеется набор из четырех элементов типа RadioButton, представляющих выбор цвета для определенного автомобиля, и необходимо обеспечить, чтобы только один из четырех был выбран в каждый отдельный момент времени. Вместо написания соответствующего программного кода воспользуйтесь элементом управления GroupBox, чтобы гарантировать, что все входящие в него элементы RadioButton были взаимоисключающими.

Чтобы проиллюстрировать работу с типами CheckBox, RadioButton и GroupBox, давайте создадим новое приложение Windows Forms по имени CarConfig. Главная форма позволит пользователю вводить (и подтверждать) информацию о новом автомобиле, который он желает приобрести. Итоговая информация заказа будет отображаться в элементе типа Label после щелчка на кнопке Confirm Order (Подтвердить заказ). Начальный пользовательский интерфейс представлен на рис. 4.8.

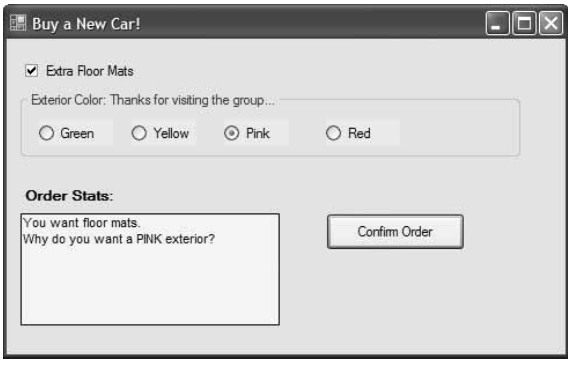


**Рис. 4.8.** Начальный пользовательский интерфейс CarConfig

Исходя из того, что для построения пользовательского интерфейса применяется дизайнер форм, должно появиться множество переменных-членов, представляющих каждый виджет GUI. Вдобавок соответствующим образом обновится метод InitializeComponent(). Первый интересный момент — конструирование элемента типа CheckBox. Подобно любому типу-наследнику Control, как только его внешность и поведение установлены, он должен быть вставлен во внутреннюю коллекцию элементов управления формы:

```
private void InitializeComponent()
{
...
    // checkFloorMats
    //
    this.checkFloorMats.Name = "checkFloorMats";
    this.checkFloorMats.TabIndex = 0;
    this.checkFloorMats.Text = "Extra Floor Mats";
...
    this.Controls.Add(this.checkFloorMats);
}
```

Затем потребуется сконфигурировать GroupBox и содержащиеся в нем элементы типа RadioButton. Для помещения элемента управления во владение GroupBox необходимо добавить его в коллекцию Controls элемента GroupBox (точно так же, как добавляются виджеты в коллекцию Controls формы). Чтобы сделать задачу более интересной, воспользуйтесь окном Properties для обработки событий Enter и Leave, отправляемых объектом GroupBox, как показано ниже:

```
private void InitializeComponent()
{
...
    // radioRed
    //
    this.radioRed.Name = "radioRed";
    this.radioRed.Size = new System.Drawing.Size(64, 23);
    this.radioRed.Text = "Red";
    //
    // groupBoxColor
    //
...
    this.groupBoxColor.Controls.Add(this.radioRed);
    this.groupBoxColor.Text = "Exterior Color";
    this.groupBoxColor.Enter += new System.EventHandler(this.groupBoxColor_Enter);
    this.groupBoxColor.Leave += new System.EventHandler(this.groupBoxColor_Leave);
...
}
```

Имейте в виду, что перехватывать события Enter или Leave для GroupBox не обязательно. Однако для целей иллюстрации обработчики событий, обновляющие текст заголовка GroupBox, показаны ниже:

```
// Определение нахождения фокуса в группен.
private void groupBoxColor_Leave(object sender, EventArgs e)
{
    groupBoxColor.Text = "Exterior Color: Thanks for visiting the group...";
}
private void groupBoxColor_Enter(object sender, EventArgs e)
{
    groupBoxColor.Text = "Exterior Color: You are in the group...";
}
```

И последние виджеты GUI в этой форме (типов Label и Button) также будут сконфигурированы и вставлены в коллекцию Controls формы в методе InitializeComponent(). Элемент Label используется для отображения подтверждения заказа, который формируется в обработчике события Click кнопки Order, как показано ниже:

```
private void btnOrder_Click (object sender, System.EventArgs e)
{
    // Построить строку для отображения информации.
    string orderInfo = "";
    if(checkFloorMats.Checked)
        orderInfo += "You want floor mats.\n";
    if(radioRed.Checked)
        orderInfo += "You want a red exterior.\n";
    if(radioYellow.Checked)
        orderInfo += "You want a yellow exterior.\n";
    if(radioGreen.Checked)
        orderInfo += "You want a green exterior.\n";
    if(radioPink.Checked)
        orderInfo += "Why do you want a PINK exterior?\n";
    // Отправить эту строку в Label.
    infoLabel.Text = orderInfo;
}
```

Обратите внимание, что и CheckBox, и RadioButton поддерживают свойство Checked, которое позволяет исследовать состояние виджета. И, наконец, вспомните, что если CheckBox был сконфигурирован с тремя состояниями, его состояние должно проверяться с помощью свойства CheckState.

## Упражнения с CheckListBox

Теперь, когда вы познакомились с базовыми виджетами на основе Button, давайте перейдем к набору типов выбора из списка, в частности — к CheckedListBox, ListBox и ComboBox. Виджет CheckedListBox позволяет группировать взаимосвязанные опции CheckBox в прокручиваемый списковый элемент управления. Предположим, что вы добавили такой элемент к форме CarConfig, который позволяет пользователям конфигурировать множество опций, касающихся аудиосистемы автомобиля (рис. 4.9).


**Рис. 4.9.** Тип CheckedListBox

Для вставки новых элементов в CheckedListBox вызывайте метод Add() для каждого элемента либо AddRange() с параметром — массивом объектов (строк, чтобы быть точным), представляющим полный набор помечаемых элементов. Имейте в виду, что вы можете заполнять любые списковые типы во время дизайна, используя свойство Items в окне Properties (просто щелкните на кнопке с многоточием и введите нужные строковые значения). Ниже показан соответствующий код метода InitializeComponent(), который конфигурирует CheckedListBox:

```
private void InitializeComponent()
{
...
    // checkedBoxRadioOptions
    //
    this.checkedBoxRadioOptions.Items.AddRange(new object[] {
        "Front Speakers", "8-Track Tape Player",
        "CD Player", "Cassette Player",
        "Rear Speakers", "Ultra Base Thumper"});
...
    this.Controls.Add (this.checkedBoxRadioOptions);
}
```

Теперь обновите логику обработчика событий Click кнопки Confirm Order. Опросите CheckedListBox на предмет отмеченных позиций списка и добавьте их в строку orderInfo. Вот необходимые изменения кода:

```
private void btnOrder_Click (object sender, EventArgs e)
{
    // Построить строку для отображения информации.
    string orderInfo = "";
    ...
    orderInfo += "---------------------------------\n";

    // Для каждого элемента в CheckedListBox:
    for(int i = 0; i < checkedBoxRadioOptions.Items.Count; i++)
    {
        // Элемент отмечен?
        if(checkedBoxRadioOptions.GetItemChecked(i))
        {
            // Получить текст отмеченного элемента и добавить к строке orderinfo.
            orderInfo += "Radio Item: ";
            orderInfo += checkedBoxRadioOptions.Items[i].ToString();
            orderInfo += "\n";
        }
    }
    ...
}
```

Последнее замечание относительно типа CheckedListBox состоит в том, что он поддерживает использование множественных столбцов через унаследованное свойство MultiColumn. Таким образом, добавив следующий код:

```
checkedBoxRadioOptions.MultiColumn = true;
```

можно увидеть многостолбцовый CheckedListBox, показанный на рис. 4.10.

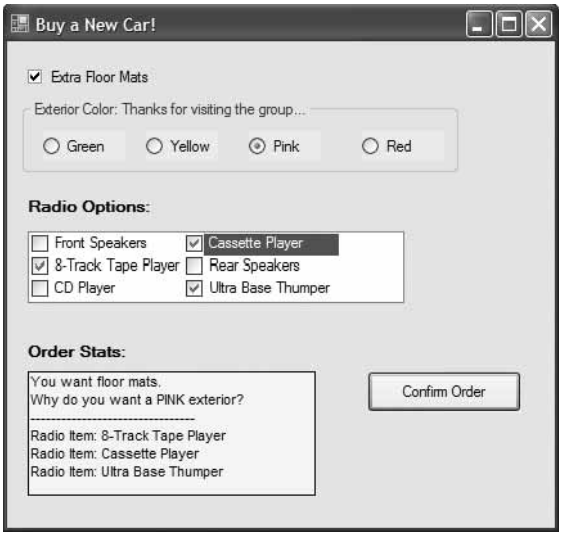


**Рис. 4.10.** Многостолбцовый элемент CheckedListBox

## Упражнения с ListBox

Как упоминалось ранее, тип CheckListBox наследует большую часть своей функциональности от типа ListBox. Чтобы проиллюстрировать применение типа ListBox, добавим еще одно средство к текущему приложению CarConfig — возможность выбирать производителя (BMW, Yugo и т.п.) автомобиля. На рис. 4.11. показан желаемый пользовательский интерфейс.


**Рис. 4.11.** Тип ListBox

Как всегда, начнем с создания переменной-члена для манипулирования типом (в данном случае — ListBox). Затем сконфигурируем внешний вид и поведение, используя следующий снимок InitializeComponent():

```
private void InitializeComponent()
{
...
    // carMakeList
    //
    this.carMakeList.Items.AddRange(new object[] {
        "BMW", "Caravan", "Ford", "Grand Am",
        "Jeep", "Jetta", "Saab", "Viper", "Yugo"});
...
    this.Controls.Add (this.carMakeList);
}
```

Обновление обработчика событий btnOredr_Click() также просто:

```
private void btnOrder_Click (object sender, EventArgs e)
{
    // Построить строку для отображения информации.
    string orderInfo = "";
    ...
    // Получить текущий выбранный элемент (а не его индекс).
    if(carMakeList.SelectedItem != null)
      orderInfo += "Make: " + carMakeList.SelectedItem + "\n";
...
}
```

## Упражнения с ComboBox

Подобно ListBox, элемент ComboBox позволяет пользователям производить выбор из известного набора возможных элементов. Однако тип ComboBox уникален в том отношении, что пользователи также могут вставлять дополнительные элементы. Напомним, что ComboBox наследуется от ListBox (который наследуется от Control). Чтобы проиллюстрировать их использование, добавим еще один виджет GUI к форме CarConfig, который позволит пользователю вводить имя предпочтительного продавца. Одно из возможных обновлений пользовательского интерфейса показано на рис. 4.12 (при желании добавьте свои собственные имена продавцов).


**Рис. 4.12.** Тип ComboBox

Эта модификация начинается с конфигурирования самого ComboBox. Как видите, логика выглядит идентично той, что применялась с ListBox:

```
private void InitializeComponent()
{
...
    // comboSalesPerson
    //
    this.comboSalesPerson.Items.AddRange(new object[] {
        "Baby Ry-Ry", "Dan \'the Machine\'", "Danny Boy", "Tommy Boy"});
...
    this.Controls.Add (this.comboSalesPerson);
}
```

Обновление обработчика событий btnOredr_Click() снова не сложно, как показано ниже:

```
private void btnOrder_Click (object sender, EventArgs e)
{
    // Построить строку для отображения информации.
    string orderInfo = "";
    ...
    // Использовать свойство Text для вывода продавца.
    if(comboSalesPerson.Text != "")
        orderInfo += "Sales Person: " + comboSalesPerson.Text + "\n";
    else
        orderInfo += "You did not select a sales person!" + "\n";
    ...
}
```

# Настройка порядка обхода по клавише табуляции

Теперь, когда создана довольно интересная форма, давайте формализуем проблему порядка обхода по клавише табуляции. Как известно, когда форма содержит множество виджетов GUI, пользователь ожидает возможности перемещения фокуса между ними с помощью клавиши <Tab>. Настройка порядка обхода по клавише <Tab> для набора элементов управления требует понимания двух ключевых свойств: `TabStop` и `TabIndex`.

Свойство `TabStop` может быть установлено в `true` или `false`, в зависимости от того, хотите ли вы, чтобы данный элемент графического интерфейса был доступен по нажатию <Tab>. Предположим, что свойство `TabStop` установлено в `true` для определенного виджета; тогда свойство `TabIndex` устанавливает порядок активизации этого элемента в последовательности обхода по <Tab> (начиная с нуля). Рассмотрим пример:

```
// Конфигурирование свойств табуляции.
txtMake.TabIndex = 2;
txtMake.TabStop = true;
```

## Мастер порядка обхода по клавише табуляции

В интегрированной среде разработки Visual Studio имеется мастер порядка обхода по клавише табуляции (Tab Order Wizard), который доступен через пункт меню View⇨Tab Order (Вид⇨Порядок обхода); учтите, что этот пункт меню виден, только если активен дизайнер форм. После запуска мастера дизайнер форм отобразит текущее значение `TabIndex` для каждого виджета. Чтобы изменить эти значения, прощелкайте на элементах в нужном порядке (рис. 4.13).

Для выхода из мастера Tab Order Wizard нажмите клавишу <Esc>.

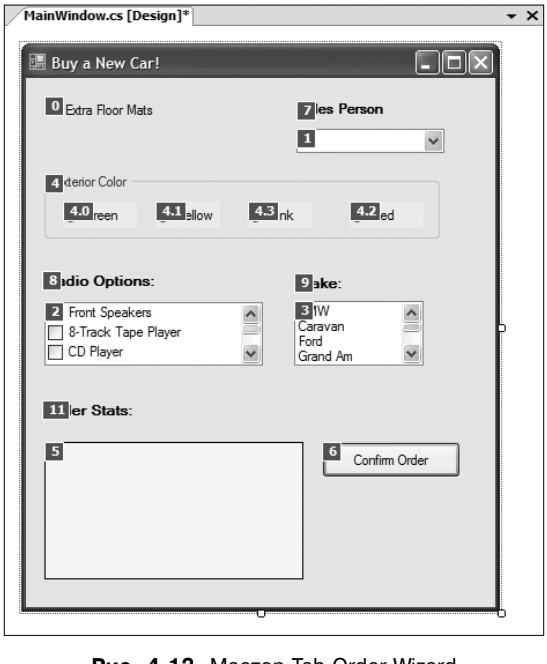


**Рис. 4.13.** Мастер Tab Order Wizard

# Установка в форме кнопки ввода по умолчанию

Многие формы пользовательского ввода (особенно диалоговые окна) имеют специальный элемент `Button`, который автоматически реагирует на нажатие пользователем клавиши <Enter>. Если необходимо обеспечить для текущей формы, чтобы при нажатии клавиши <Enter> пользователем срабатывал обработчик события `Click` кнопки `btnOrder`, установите свойство `AcceptButton`, как показано ниже (та же установка может быть выполнена через окно Properties):

```
// Когда нажата клавиша <Enter>, это все равно, как
// если бы пользователь щелкнул на кнопке btnOrder.
this.AcceptButton = btnOrder;
```

**На заметку!** Некоторые формы требуют возможности эмуляции щелчка на кнопке Cancel (Отмена), когда пользователь нажимает клавишу <Esc>. Это можно сделать, присвоив свойству `CancelButton` формы ссылку на объект `Button`, представляющий кнопку Cancel.

# Работа с экзотическими элементами управления

К данному моменту вы уже видели, как работать с базовыми элементами управления Windows Forms (`Label`, `TextBox` и т.п.). Наша следующая задача — исследовать некоторые виджеты GUI с более мощной функциональностью. К счастью, то, что элемент управления может показаться "более экзотичным", еще не означает, что с ним трудно работать. На нескольких следующих разделах мы рассмотрим следующие элементы управления:

- `MonthCalendar`
- `ToolTip`
- `TabControl`
- `TrackBar`
- `Panel`
- `UpDown`
- `ErrorProvider`
- `TreeView`
- `WebBrowser`

Для начала дополним проект `CarConfig` элементами управления `MonthCalendar` и `ToolTip`.

## Упражнения с `MonthCalendar`

Пространство имен `System.Windows.Forms` предоставляет исключительно полезный элемент управления `MonthCalendar`, который позволяет пользователю выбирать дату (или диапазон дат), используя дружественный графический интерфейс. Чтобы продемонстрировать этот новый элемент в работе, модифицируем существующее приложение `CarConfig`, позволив пользователю вводить дату поставки нового автомобиля.

На рис. 4.14 показана обновленная и слегка реорганизованная форма.


**Рис. 4.14.** Элемент `MonthCalendar`

Хотя элемент управления `MonthCalendar` предлагает развитую функциональность, программно получить диапазон дат, указанный пользователем, очень просто. Поведение этого типа по умолчанию состоит в том, что текущая дата (сегодня) всегда выбирается автоматически. Для получения выбранной даты в коде нужно обновить обработчик событий `Click` для кнопки Confirm Order следующим образом:

```
private void btnOrder_Click (object sender, EventArgs e)
{
    // Построить строку для отображения информации.
    string orderInfo = "";
    ...
    // Извлечь дату поставки.
    DateTime d = monthCalendar.SelectionStart;
    string dateStr = string.Format("{0}/{1}/{2}", d.Month, d.Day, d.Year);
    orderInfo += "Car will be sent: " + dateStr;
    ...
}
```

Обратите внимание на возможность запроса у `MonthCalendar` текущей выбранной даты с помощью свойства `SelectionStart`. Это свойство возвращает ссылку `DateTime`, которая сохраняется в локальной переменной. Используя несколько свойств типа `DateTime`, можно извлечь требуемую информацию в заданном формате.

Пока что предполагается, что пользователь специфицирует один день, когда должен быть доставлен автомобиль. Однако что если необходимо позволить пользователю выбрать диапазон возможных дат поставки? В этом случае все, что пользователю понадобится сделать — это перетащить курсор по диапазону желаемых дат. Вы уже видели, как получить начало выбора через свойство `SelectionStart`. Конец диапазона может быть определен с помощью свойства `SelectionEnd`. Вот необходимое изменение кода:

```
private void btnOrder_Click (object sender, EventArgs e)
{
   // Построить строку для отображения информации.
   string orderInfo = "";
   ...
   // Получить диапазон дат поставки....
   DateTime startD = monthCalendar.SelectionStart;
   DateTime endD = monthCalendar.SelectionEnd;
   string dateStartStr =
      string.Format("{0}/{1}/{2}", startD.Month, startD.Day, startD.Year);
   string dateEndStr =
      string.Format("{0}/{1}/{2}", endD.Month, endD.Day, endD.Year);
   // Тип DateTime поддерживает перегруженные операции!
   if(dateStartStr != dateEndStr)
   {
      orderInfo += "Car will be sent between "
         + dateStartStr + " and\ n" + dateEndStr;
   }
   else // Выбрана одна дата.
      orderInfo += "Car will be sent on " + dateStartStr;
   ...
}
```

**На заметку!** Набор инструментов Windows Forms также включает элемент `DateTimePicker`, который отображает `MonthCalendar` в элементе управления `DropDown`.

## Упражнения с `ToolTip`

Если говорить о форме `CarConfig`, то тут остается еще один интересный момент. Большинство современных пользовательских интерфейсов поддерживают *всплывающие подсказки* (tool tips). В пространстве имен `System.Windows.Forms` эту функциональность предлагает тип `ToolTip`. Этот виджет представляет собой маленькое плавающее окно, которое отображает полезное сообщение при наведении курсора на определенный элемент.

Для иллюстрации добавим всплывающую подсказку к типу `Calendar` из `CarConfig`. Начните с перетаскивания нового элемента управления `ToolTip` из панели Toolbox на поверхность дизайнера форм, после чего переименуйте его в `calendarTip`. С помощью окна Properties настройте внешний вид и поведение виджета `ToolTip`, например:

```
private void InitializeComponent()
{
...
   // calendarTip
   //
   this.calendarTip.IsBalloon = true;
   this.calendarTip.ShowAlways = true;
   this.calendarTip.ToolTipIcon = System.Windows.Forms.ToolTipIcon.Info;
...
}
```

Чтобы ассоциировать `ToolTip` с определенным элементом управления, выберите нужный элемент, который должен активизировать `ToolTip`, и установите свойство `ToolTip on` (рис. 4.15).

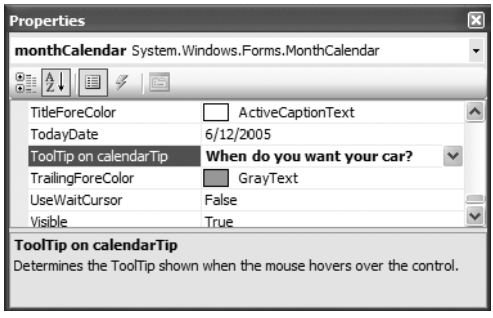


**Рис. 4.15.** Ассоциирование `ToolTip` с заданным виджетом

На этом проект `CarConfig` готов. На рис. 4.16 показан элемент `ToolTip` в действии.

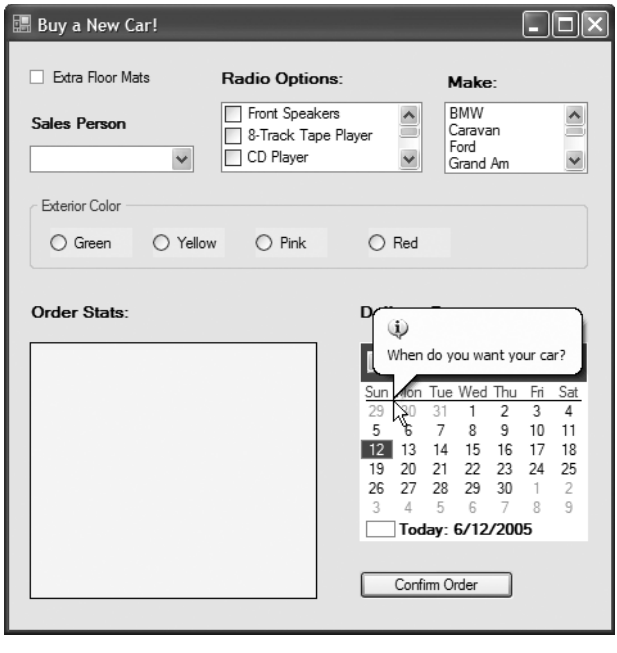


**Рис. 4.16.** `ToolTip` в действии

**Исходный код.** Проект `CarConfig` включен в подкаталог `Bonus Chapter 4`.

## Упражнения с `TabControl`

Для иллюстрации остальных экзотических элементов управления построим новую форму, содержащую `TabControl`. Как должно быть известно, `TabControl` позволяет выборочно скрывать или показывать страницы взаимосвязанного содержимого GUI посредством выбора определенной вкладки. Для начала создайте новое приложение Windows Forms по имени `ExoticControls` и переименуйте начальную форму в `MainWindow`.

Затем добавьте `TabControl` в дизайнер форм и в окне Properties откройте редактор страниц через коллекцию `TabPages` (просто щелкните на кнопке с многоточием).

Отобразится окно конфигурирования. Добавьте шесть страниц, установив для каждой из них свойства `Text` и `Name`, как показано на рис. 4.17.

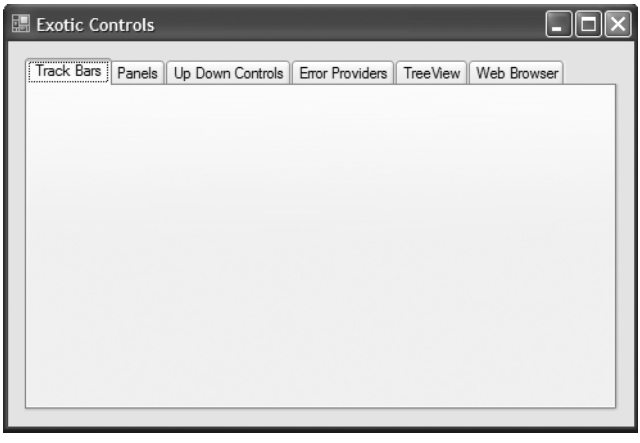


**Рис. 4.17.** Многостраничный `TabControl`

При проектировании `TabControl` имейте в виду, что каждая его страница представлена объектом `TabPage`, который вставлен во внутреннюю коллекцию `TabControl`. Как только `TabControl` сконфигурирован, этот объект (как любой другой виджет GUI внутри `Form`), помещается в коллекцию `Controls` формы. Рассмотрим следующий частичный код `InitializeComponent()`:

```
private void InitializeComponent()
{
...
   // tabControlExoticControls
   //
   this.tabControlExoticControls.Controls.Add(this.pageTrackBars);
   this.tabControlExoticControls.Controls.Add(this.pagePanels);
   this.tabControlExoticControls.Controls.Add(this.pageUpDown);
   this.tabControlExoticControls.Controls.Add(this.pageErrorProvider);
   this.tabControlExoticControls.Controls.Add(this.pageTreeView);
   this.tabControlExoticControls.Controls.Add(this.pageWebBrowser);
   this.tabControlExoticControls.Location = new System.Drawing.Point(13, 13);
   this.tabControlExoticControls.Name = "tabControlExoticControls";
   this.tabControlExoticControls.SelectedIndex = 0;
   this.tabControlExoticControls.Size = new System.Drawing.Size(463, 274);
   this.tabControlExoticControls.TabIndex = 0;
...
   this.Controls.Add(this.tabControlExoticControls);
}
```

Имея форму, поддерживающую несколько вкладок, можно строить каждую страницу для иллюстрации остальных экзотических элементов управления. Прежде всего, рассмотрим назначение `TrackBar`.

---

**На заметку!** Виджет `TabControl` поддерживает события `Selected`, `Selecting`, `Deselected` и `Deselecting`. Они могут быть полезны, когда необходимо динамически генерировать элементы внутри определенной страницы.

---

## Упражнения с TrackBar

Элемент управления TrackBar позволяет пользователям выбирать из диапазона значений, используя механизм, подобный линейке прокрутки. При работе с этим типом нужно установить минимальное и максимальное значения диапазона, минимальные и максимальные инкременты и начальное положение движка. Каждый из этих аспектов может быть установлен с использованием свойств, описанных в табл. 4.6.

**Таблица 4.6. Свойства TrackBar**

| Свойства | Назначение |
|---|---|
| LargeChange | Количество "тиков", на которое изменяется TrackBar при возникновении события, считающегося большим изменением (например, щелчок кнопкой мыши, пока курсор находится в диапазоне перемещения ползунка, либо нажатие клавиши <Page Up> или <Page Down>). |
| Maximum<br>Minimum | Конфигурируют верхнюю и нижнюю границы диапазона TrackBar. |
| Orientation | Ориентация TrackBar. Допустимые значения берутся из перечисления Orientation (т.е. горизонтальная и вертикальная). |
| SmallChange | Количество "тиков", на которое изменяется TrackBar при возникновении события, считающегося малым изменением (например, нажатие клавиш со стрелками). |
| TickFrequency | Показывает количество нарисованных "тиков". Для TrackBar с верхней границей 200 непрактично отображать все 200 тиков в элементе длиной 2 дюйма. Если вы установите свойство TickFrequency равным 5, то TrackBar нарисует всего 20 тиков (каждый представляет 5 единиц). |
| TickStyle | Устанавливает, каким образом TrackBar рисует себя. Это касается как места рисования тиков относительно перемещаемого ползунка, так и способ отображения самого ползунка (с использованием перечисления TickStyle). |
| Value | Получает или устанавливает текущее значение TrackBar. Используйте это свойство для получения числового значения, содержащегося в TrackBar, для использования в приложении. |

Для иллюстрации добавьте на первую вкладку TabControl три элемента TrackBar, каждый с верхним пределом диапазона 255 и нижним — 0 единиц. По мере перемещения ползунка пользователем приложение перехватывает событие Scroll и динамически строит новый объект типа System.Drawing.Color на основе значения каждого из трех ползунков. Этот объект Color используется для отображения цвета внутри виджета PictureBox (по имени colorBox) и значений RGB внутри элемента Label (под названием lblCurrentColor). На рис. 4.18 показана готовая первая страница в действии.

Поместите три элемента TrackBar на первую вкладку, используя дизайнер форм, и соответствующим образом переименуйте переменные-члены (redTrackBar, greenTrackBar и blueTrackBar). Затем обработайте событие Scroll для каждого из элементов TrackBar.

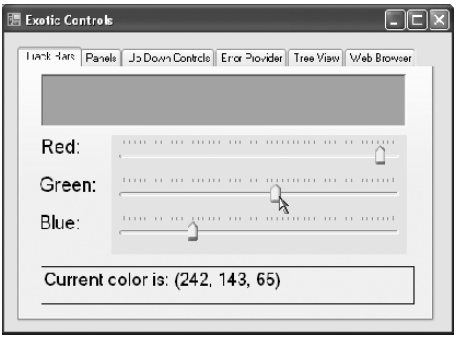


**Рис. 4.18.** Страница с элементами `TrackBar`

Ниже приведен соответствующий код для `blueTrackBar` (остальные почти такие же, за исключением имени обработчика события `Scroll`):

```
private void InitializeComponent()
{
...
    //
    // blueTrackBar
    //
    this.blueTrackBar.Maximum = 255;
    this.blueTrackBar.Name = "blueTrackBar";
    this.blueTrackBar.TickFrequency = 5;
    this.blueTrackBar.TickStyle = System.Windows.Forms.TickStyle.TopLeft;
    this.blueTrackBar.Scroll += new System.EventHandler(this.blueTrackBar_Scroll);
...
}
```

Обратите внимание, что минимальное значение `TrackBar` по умолчанию равно 0 и потому не нуждается в явной установке. В обработчиках событий для каждого `TrackBar` вызывается пока не написанная вспомогательная функция по имени `UpdateColor()`:

```
private void blueTrackBar_Scroll (object sender, EventArgs e)
{
    UpdateColor();
}
```

`UpdateColor()` отвечает за решение двух основных задач. Первая: вы читаете текущее значение каждого `TrackBar` и применяете его данные для построения новой переменной `Color`, используя `Color.FromArgb()`. После получения нового цвета переменная-член `PictureBox` (`colorBox`) обновляется текущим фоновым цветом. И вторая: `UpdateColor()` форматирует значение ползунка в строку, помещаемую на `Label` (`lblCurrColor`), как показано ниже:

```
private void UpdateColor()
{
    // Получить новый цвет на основе положения ползунков.
    Color c = Color.FromArgb(redTrackBar.Value,
        greenTrackBar.Value, blueTrackBar.Value);
    // Изменить цвет PictureBox.
    colorBox.BackColor = c;
    // Установить метку цвета.
    lblCurrColor.Text = string.Format("Current color is: (R:{0}, G:{1}, B:{2})",
        redTrackBar.Value, greenTrackBar.Value,
        blueTrackBar.Value);
}
```

Последняя деталь — установка начальных значений каждого ползунка при запуске формы и визуализация текущего цвета, как показано ниже:

```
public MainWindow()
{
    InitializeComponent();
    CenterToScreen();
    // Установка начального положения каждого ползунка.
    redTrackBar.Value = 100;
    greenTrackBar.Value = 255;
    blueTrackBar.Value = 0;
    UpdateColor();
}
```

## Упражнения с Panel

Как вы уже видели ранее в главе, элемент управления GroupBox можно применять для логической группировки других элементов управления (таких как RadioButton), чтобы они функционировали совместно. Тесно связан с GroupBox элемент управления Panel. Он также используется для объединения связанных элементов в логическую единицу. Единственное отличие состоит в том, что тип Panel унаследован от класса ScrollableControl, а потому может поддерживать линейки прокрутки, что невозможно в GroupBox.

Элементы Panel могут применяться для предохранения компоновки экрана. Например, если есть группа элементов управления, занимающая всю нижнюю половину формы, ее можно поместить в Panel размером в половину формы и установить свойство AutoScroll в true. Таким образом, пользователь сможет применить линейку прокрутки, чтобы увидеть полный набор элементов. Более того, если свойство BorderStyle в Panel установлено в None, этот тип можно применять просто для объединения в группу набора элементов, которые могут быть легко показаны или скрыты из виду прозрачным для пользователя способом.

Для иллюстрации давайте обновим вторую страницу TabControl, добавив два элемента Button (btnShowPanel и btnHidePanel) и одну Panel, содержащую пару текстовых полей (txtNormalText и txtUpperText) и Label с инструкцией. (Имейте в виду, что виджеты, помещенные в Panel, не имеют особого значения в данном примере). На рис. 4.19 показан финальный графический интерфейс.

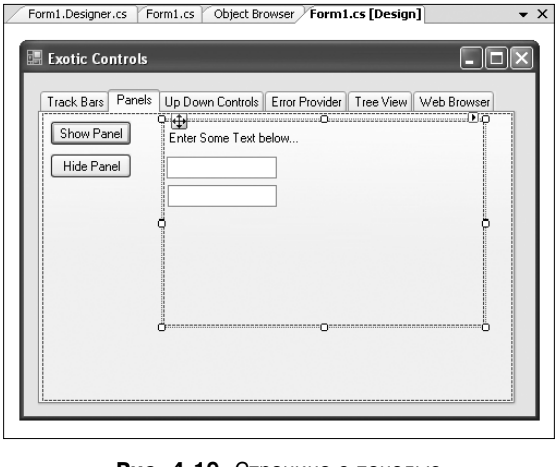


**Рис. 4.19.** Страница с панелью

В окне Properties обработайте событие TextChanged для первого TextBox, и внутри сгенерированного обработчика поместите версию текста, введенного в txtNormalText и txtUpperText, но в верхнем регистре:

```
private void txtNormalText_TextChanged(object sender, EventArgs e)
{
   txtUpperText.Text = txtNormalText.Text.ToUpper();
}
```

Теперь обработайте событие Click для каждой кнопки. Как и можно было ожидать, элемент Panel будет просто скрываться или отображаться (вместе со всеми содержащимися в нем интерфейсными элементами):

```
private void btnShowPanel_Click(object sender, EventArgs e)
{
  panelTextBoxes.Visible = true;
}
private void btnHidePanel_Click(object sender, EventArgs e)
{
   panelTextBoxes.Visible = false;
}
```

Если теперь запустить программу и щелкнуть на обеих кнопках, содержимое Panel будет отображаться или скрываться. Несмотря на простоту примера, вы сумеете оценить потенциальные возможности. Например, может существовать пункт меню, который позволяет пользователю видеть "простой" или "сложный" вид страницы. Вместо того чтобы вручную устанавливать свойство Visible в false для множества виджетов, их можно сгруппировать в элемент Panel и устанавливать Visible в нем.

## Упражнения с UpDown

Windows Forms предоставляет два виджета, которые функционируют как *элементы управления со счетчиком*. Подобно типам ComboBox и ListBox, эти элементы также позволяют пользователю выбирать значения из диапазона возможных значений. Отличие в том, что когда применяется элемент DomainUpDown или NumericUpDown, информация выбирается с помощью пары маленьких стрелок вверх и вниз (рис. 4.20).

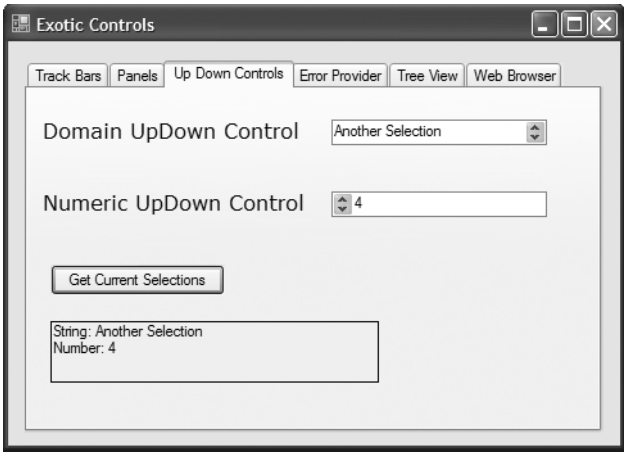


**Рис. 4.20.** Работа с UpDown

Учитывая, что вы уже работали с подобными типами, вы найдете работу с виджетами UpDown несложной. Виджет DomainUpDown позволяет пользователю выбирать из множества строковых данных. NumericUpDown позволяет выбирать из диапазона числовых данных. Каждый виджет наследуется от общего базового класса UpDownBase.

В табл. 4.7 описаны важнейшие свойства этого класса.

**Таблица 4.7. Свойства UpDownBase**

| Свойство | Назначение |
|---|---|
| InterceptArrowKeys | Получает или устанавливает значение, указывающее на то, может ли пользователь использовать клавиши со стрелками вверх и вниз для выбора значений. |
| ReadOnly | Получает и устанавливает значение, указывающее, может ли текст изменяться только посредством стрелок вверх и вниз либо также посредством ввода с клавиатуры для быстрого поиска нужной строки. |
| Text | Получает или устанавливает текущий текст, отображаемый в элементе управления. |
| TextAlign | Получает или устанавливает выравнивание текста в элементе управления. |
| UpDownAlign | Получает или устанавливает выравниванием стрелок вверх и вниз в элементе управления, используя перечисление LeftRightAlingment. |

Элемент управления DomainUpDown добавляет небольшой набор свойств (см. табл. 4.8), которые позволяют конфигурировать и манипулировать текстовыми данными в виджете.

**Таблица 4.8. Свойства DomainUpDown**

| Свойство | Назначение |
|---|---|
| Items | Позволяет получить доступ к набору элементов, хранящихся в виджете. |
| SelectedIndex | Возвращает индекс (начиная с нуля) текущего выбранного элемента (значение -1 указывает на отсутствие выбранного элемента). |
| SelectedItem | Возвращает сам выбранный элемент (а не его индекс). |
| Sorted | Указывает на необходимость сортировки строк по алфавиту. |
| Wrap | Управляет режимом перехода от последнего элемента к первому, если пользователь выходит за конец списка. |

Тип NumericUpDown не сложнее (см. табл. 4.9).

**Таблица 4.9. Свойства NumericUpDown**

| Свойство | Назначение |
|---|---|
| DecimalPlaces<br>ThousandsSeparator<br>Hexadecimal | Используются для конфигурирования отображения числовых данных. |
| Increment | Устанавливает числовую величину, на которую увеличивается или уменьшается значение в элементе управления при щелчке на стрелках вверх и вниз. По умолчанию значение изменяется на 1. |
| Minimum<br>Maximum | Устанавливают верхнюю и нижнюю границу значений в элементе управления. |
| Value | Возвращает текущее значение в элементе управления. |

Приведем частичный код `InitializeComponent()`, который конфигурирует виджеты страницы `NumericUpDown` и `DomainUpDown`:

```
private void InitializeComponent()
{
...
    //
    // numericUpDown
    //
...
    this.numericUpDown.Maximum = new decimal(new int[] {5000, 0, 0, 0});
    this.numericUpDown.Name = "numericUpDown";
    this.numericUpDown.ThousandsSeparator = true;
    //
    // domainUpDown
    //
    this.domainUpDown.Items.Add("Another Selection");
    this.domainUpDown.Items.Add("Final Selection");
    this.domainUpDown.Items.Add("Selection One");
    this.domainUpDown.Items.Add("Third Selection");
    this.domainUpDown.Name = "domainUpDown";
    this.domainUpDown.Sorted = true;
...
}
```

Обработчик события `Click` для элемента `Button` этой страницы просто опрашивает текущее значение каждого элемента и помещает его в соответствующий элемент `Label` (`lblCurrSel`) в виде форматированной строки, как показано ниже:

```
private void btnGetSelections_Click (object sender, EventArgs e)
{
    // Получить значения от updowns...
    lblCurrSel.Text = string.Format("String: {0}\nNumber: {1}",
       domainUpDown.Text, numericUpDown.Value);
}
```

## Упражнения с `ErrorProvider`

Большинство приложений Windows Forms нуждается в проверке ввода пользователя тем или иным образом. Это особенно качается диалоговых окон, поскольку вы должны информировать пользователя о необходимости исправить ошибку, прежде чем продолжить работу. Тип `ErrorProvider` применяется для выдачи визуального указания на ошибку ввода. Например, предположим, что есть `Form` с виджетами `TextBox` и `Button`. Если пользователь введет более пяти символов в `TextBox` и `TextBox` утратит фокус, то будет отображена информация об ошибке, показанная на рис. 4.21.

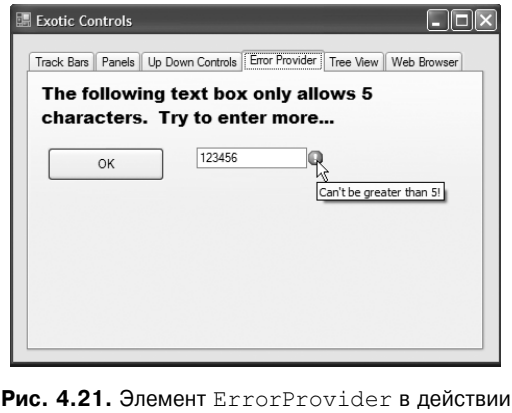


**Рис. 4.21.** Элемент `ErrorProvider` в действии

Здесь вы обнаружили, что пользователь ввел более пяти символов, на что указывает маленькая пиктограмма с восклицательным знаком (!) рядом с объектом TextBox. Когда пользователь наводит курсор на пиктограмму, появляется всплывающее текстовое описание ошибки. Также этот ErrorProvider сконфигурирован так, что несколько раз мигает, чтобы привлечь внимание (чего вы не увидите, не запустив приложение).

Если необходим такой контроль ввода, понадобится изучить свойства класса Control, перечисленные в табл. 4.10.

**Таблица 4.10. Свойства и события Control**

| Свойство | Назначение |
|---|---|
| CausesValidation | Указывает, вызывает ли проверку достоверности выбор этого элемента на элементах, требующих проверки. |
| Validated | Случается, когда элемент управления завершает выполнение своей логики проверки достоверности. |
| Validating | Случается, когда элемент управления проверяет ввод пользователя (например, когда элемент теряет фокус). |

Каждый виджет GUI может устанавливать свойство CausesValidation в true или false (по умолчанию — true). Если вы установите этот флаг состояния в true, то элемент управления принуждает другие элементы в Form выполнять проверку достоверности при получении фокуса. Как только такой элемент управления получает фокус, в нем инициируются события Validating и Validated. В контексте обработчика событий Validating настраивается соответствующий ErrorProvider. Дополнительно можно обработать событие Validated для определения того, когда элемент управления завершил цикл проверки достоверности.

Тип ErrorProvider имеет небольшой набор членов. Наиболее важный для данных целей — свойство BlinkStyle, которое может быть установлено в любое из значений перечисления ErrorBlinkStyle, описанного в табл. 4.11.

**Таблица 4.11. Значения ErrorBlinkStyle**

| Значение | Назначение |
|---|---|
| AlwaysBlink | Заставляет пиктограмму ошибки мигать, когда ошибка впервые отображается, или когда устанавливается новое описание ошибки для элемента управления, а пиктограмма ошибки уже отображается. |
| BlinkIfDifferentError | Заставляет пиктограмму ошибки мигать, только если пиктограмма ошибки уже отображается, а для элемента управления установлена новая строка ошибки. |
| NeverBlink | Указывает, что пиктограмма ошибки никогда не должна мигать. |

В целях иллюстрации обновите пользовательский интерфейс страницы Error Provider, добавив Button, TextBox и Label, как показано на рис. 20.21. Затем перетащите виджет ErrorProvider по имени tooManyCharactersErrorProvider на поверхность дизайнера. Вот код настройки из InitializeComponent():

```
private void InitializeComponent()
{
    ...
    //
    // tooManyCharactersErrorProvider
    //
```

```
    this.tooManyCharactersErrorProvider.BlinkRate = 500;
    this.tooManyCharactersErrorProvider.BlinkStyle =
       System.Windows.Forms.ErrorBlinkStyle.AlwaysBlink;
    this.tooManyCharactersErrorProvider.ContainerControl = this;
    ...
}
```

После настройки внешнего вида и поведение `ErrorProvider` вы привязываете ошибку к `TextBox` в контексте его обработчика события `Validating`, как показано ниже:

```
private void txtInput_Validating (object sender, CancelEventArgs e)
{
    // Проверить длину текста, чтобы она не превышала 5.
    if(txtInput.Text.Length > 5)
    {
       errorProvider1.SetError( txtInput, "Can't be greater than 5!");
    }
    else // Все нормально, ничего не отображать.
       errorProvider1.SetError(txtInput, "");
}
```

## Упражнения с `TreeView`

Элементы управления `TreeView` очень полезны в том отношении, что позволяют визуально отображать иерархические данные вроде структуры каталогов или любого типа с отношением "родительский–дочерний". Как и можно было ожидать, элемент управления `TreeView` в Windows Forms может тонко настраиваться. Если хотите, можете добавлять свои изображения, цвета узлов, вложенные элементы управления узлов и прочие визуальные усовершенствования (см. документацию по .NET Framework 2.0 SDK).

Чтобы проиллюстрировать базовое применение `TreeView`, следующая страница `TabControl` программно сконструирует элемент `TreeView`, определяющий серию узлов верхнего уровня, которые представляют набор элементов типа `Car`. Каждый узел `Car` имеет два подузла, представляющие текущую скорость выбранного автомобиля и любимую радиостанцию. Обратите внимание на рис. 4.22, что выбранный элемент подсвечен. Также отметьте, что если выбранный узел имеет родителя (или соседний узел), его имя представлено в виджете `Label`.

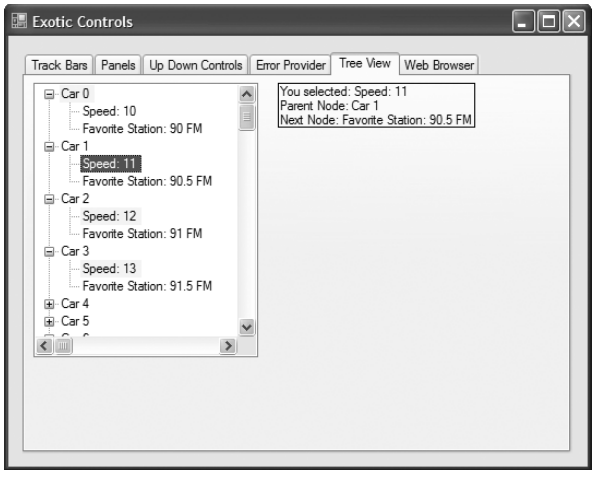


**Рис. 4.22.** Элемент `TreeView` в действии

Предполагая, что пользовательский интерфейс TreeView состоит из элемента управления TreeView (по имени treeViewCars) и Label (по имени lblNodeInfo), вставьте новый файл C# в проект ExoticControls, который моделирует простой Car, имеющий Radio:

```
namespace ExoticControls
{
    class Car
    {
        public Car(string pn, int cs)
        {
            petName = pn;
            currSp = cs;
        }
        public string petName;
        public int currSp;
        public Radio r;
    }
    class Radio
    {
        public double favoriteStation;
        public Radio(double station)
        { favoriteStation = station; }
    }
}
```

Тип-наследник Form будет поддерживать обобщенную коллекцию List<> (по имени listCars) из 100 объектов Car, которая будет наполнена в конструкторе по умолчанию типа MainForm. К тому же конструктор вызовет новый вспомогательный метод по имени BuildCarTreeView(), не принимающий аргументов и возвращающий void. Вот начальное обновление:

```
public partial class MainWindow : Form
{
    // Создать новый обобщенный List для хранения объектов Car.
    private List<Car> listCars = new List<Car>();
    public MainWindow()
    {
...
        // Заполнить List<> и построить TreeView.
        double offset = 0.5;
        for (int x = 0; x < 100; x++)
        {
            listCars.Add(new Car(string.Format("Car {0}", x), 10 + x));
            offset += 0.5;
            listCars[x].r = new Radio(89.0 + offset);
        }
        BuildCarTreeView();
    }
...
}
```

Обратите внимание, что petName каждого автомобиля основан на текущем значении x (Car 0, Car 1, Car 2 и т.п.). Кроме того, текущая скорость устанавливается смещением x на 10 (от 10 миль в час до 109 миль в час), в то время как любимая радиостанция устанавливается смещением значения 89.0 на 0.5 (90, 90.5, 91, 91.5 и т.д.).

Теперь, имея список Car, нужно отобразить эти значения на узлы элемента управления TreeView. Наиболее важный аспект для понимания при работе с виджетом

TreeView состоит в том, что каждый узел верхнего уровня и его подузлы представлены объектом System.Windows.Forms.TreeNode, унаследованным непосредственно от MarshalByRefObject. Вот некоторые из интересных свойств TreeNode:

```
public class TreeNode : MarshalByRefObject,
   ICloneable, ISerializable
{
...
   public Color BackColor { get; set; }
   public bool Checked { get; set; }
   public virtual ContextMenu ContextMenu { get; set; }
   public virtual ContextMenuStrip ContextMenuStrip { get; set; }
   public Color ForeColor { get; set; }
   public int ImageIndex { get; set; }
   public bool IsExpanded { get; }
   public bool IsSelected { get; }
   public bool IsVisible { get; }
   public string Name { get; set; }
   public TreeNode NextNode { get; }
   public Font NodeFont { get; set; }
   public TreeNodeCollection Nodes { get; }
   public TreeNode PrevNode { get; }
   public string Text { get; set; }
   public string ToolTipText { get; set; }
...
}
```

Как видите, каждому узлу можно присваивать графические образы, цвета, шрифты, всплывающие подсказки и контекстные меню. К тому же TreeNode предоставляет члены для навигации к следующему (предыдущему) TreeNode. Учитывая это, рассмотрим начальную реализацию BuildCarTreeView():

```
private void BuildCarTreeView()
{
   // Не рисовать TreeView, пока все TreeNode не будут созданы.
   treeViewCars.BeginUpdate();
   // Очистить TreeView от всех текущих TreeNode.
   treeViewCars.Nodes.Clear();
   // Добавить TreeNode для каждого объекта Car из List<>.
   foreach (Car c in listCars)
   {
      // Добавить текущий Car в качестве самого верхнего узла.
      treeViewCars.Nodes.Add(new TreeNode(c.petName));
      // Теперь получить только что добавленный Car
      // для построения двух подузлов для скорости
      // и встроенного объекта Radio.
      treeViewCars.Nodes[listCars.IndexOf(c)].Nodes.Add(
         new TreeNode(string.Format("Speed: {0}",
         c.currSp.ToString())));
      treeViewCars.Nodes[listCars.IndexOf(c)].Nodes.Add(
         new TreeNode(string.Format("Favorite Station: {0} FM",
         c.r.favoriteStation)));
   }
   // Нарисовать TreeView.
   treeViewCars.EndUpdate();
}
```

Как видите, конструирование узлов TreeView осуществляется между вызовами BeginUpdate() и EndUpdate(). Это может быть полезно при наполнении объемного

TreeView большим количеством узлов, учитывая, что виджет задержит отображение элементов до тех пор, пока не будет завершено наполнение коллекции Nodes. Таким образом, конечный пользователь не увидит постепенной визуализации элементов TreeView.

Узлы верхнего уровня добавляются к TreeView в процессе итерации по обобщенному типу List<> и вставки нового объекта TreeNode в коллекцию Nodes элемента TreeView. Как только узел верхнего уровня добавлен, можно добавлять подузлы (которые также представлены объектами TreeNode). Как и можно было предположить, для добавления подузлов к текущему подузлу нужно наполнить его внутреннюю коллекцию узлов через свойство Nodes.

Следующая задача для этой страницы элемента TabControl — подсветить текущий выбранный узел (через свойство BackColor) и отобразить выбранный элемент (вместе с родительским узлом и подузлами) внутри Label. Все это можно сделать, обрабатывая событие AfterSelect из TreeView в окне Properties. Это событие инициируется после того, как пользователь выбрал узел щелчком кнопкой мыши или навигацией с помощью клавиатуры. Ниже приведена полная реализация обработчика событий AfterSelect:

```
private void treeViewCars_AfterSelect(object sender, TreeViewEventArgs e)
{
    string nodeInfo = "";
    // Построить информацию о выбранном узле.
    nodeInfo = string.Format("You selected: {0}\n", e.Node.Text);
    if (e.Node.Parent != null)
        nodeInfo += string.Format("Parent Node: {0}\n", e.Node.Parent.Text);
    if (e.Node.NextNode != null)
        nodeInfo += string.Format("Next Node: {0}", e.Node.NextNode.Text);
    // Показать информацию и подсветить узел.
    lblNodeInfo.Text = nodeInfo;
    e.Node.BackColor = Color.AliceBlue;
}
```

Входной параметр типа TreeViewEventArgs содержит свойство по имени Node, которое возвращает объект TreeNode, представляющий текущий выбор. Отсюда можно извлечь имя узла (через свойство Text) вместе с родительским и следующим узлами (из свойств Parent/NextNode). Обратите внимание, что вы явно проверяете объекты TreeNode, возвращенные Parent/NextNode на равенство null — на случай, если пользователь выбрал первый узел верхнего уровня или самый последний подузел (если этого не сделать, может возникнуть исключение NullReferenceException).

## Добавление изображений к узлу

Чтобы завершить исследование типа TreeView, давайте украсим текущий пример, определив три изображения *.bmp, которые будут присвоены каждому типу узлов. Для этого добавьте новый компонент ImageList (по имени imageListTreeView) в дизайнере к типу MainForm. Затем добавьте три новых битовых изображения к проекту с помощью пункта меню Project⇨Add New Item (можно воспользоваться файлами *.bmp из загружаемого кода для этой главы), представляющие (или по крайней мере, похожие) на автомобиль, радиоприемник и символ скорости. Обратите внимание, что каждый из этих файлов *.bmp имеет размер 16×16 пикселей (устанавливается в окне Properies), чтобы прилично выглядеть внутри элемента TreeView.

Создав эти файлы изображений, выберите ImageList в дизайнере и заполните свойство Images каждым из трех изображений, упорядоченных, как показано на рис. 4.23, чтобы гарантировать назначение правильного ImageIndex (0, 1 или 2) каждому узлу.

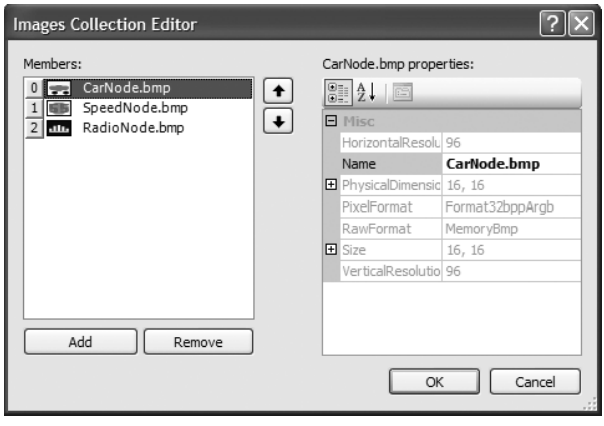


**Рис. 4.23.** Наполнение `ImageList`

Вспомните, что при включении ресурсов (наподобие битовых изображений) в решение Visual Studio соответствующий файл `*.resx` обновляется автоматически. Таким образом, эти изображения будут встраиваться в сборку без каких-либо дополнительных усилий с вашей стороны. Теперь, используя окно Properties, установите свойство `ImageList` элемента управления `TreeView` равным переменной-члену `ImageList` (рис. 4.24).

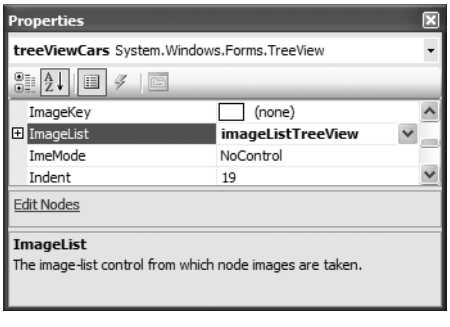


**Рис. 4.24.** Ассоциированием `ImageList` с `TreeView`

И последнее: модифицируйте метод `BuildCarTreeView()`, чтобы специфицировать корректный `ImageIndex` (через аргументы конструктора) при создании `TreeNode`:

```
private void BuildCarTreeView()
{
...
   foreach (Car c in listCars)
   {
      treeViewCars.Nodes.Add(new TreeNode(c.petName, 0, 0));
      treeViewCars.Nodes[listCars.IndexOf(c)].Nodes.Add(
         new TreeNode(string.Format("Speed: {0}",
         c.currSp.ToString()), 1, 1));
      treeViewCars.Nodes[listCars.IndexOf(c)].Nodes.Add(
         new TreeNode(string.Format("Favorite Station: {0} FM",
         c.r.favoriteStation), 2, 2));
   }
...
}
```

Обратите внимание, что вы специфицируете каждый `ImageIndex` дважды. Причина в том, что данный `TreeNode` может иметь два уникальных образа, назначенных ему: один для отображения в обычном состоянии, а другой — в выбранном состоянии. Чтобы не усложнять, вы специфицируете одно и то же изображение для обеих ситуаций. В любом случае, на рис. 4.25 показан обновленный тип `TreeView`.

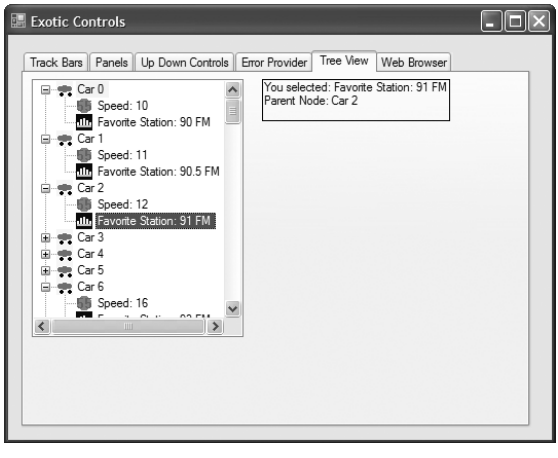


**Рис. 4.25.** Элемент `TreeView` с изображениями

## Упражнения с `WebBrowser`

Последняя страница настоящего примера использует виджет `System.Windows.Forms.WebBrowser`, который появился в .NET 2.0. Этот виджет — в высшей степени конфигурируемый веб-браузер, который может встраиваться в любой тип-наследник `Form`. Как и можно было ожидать, этот элемент управления определяет свойство `Url`, которому можно присвоить любой URL, формально представленный типом `System.Uri`. На страницу Web Browser (Веб-браузер) добавьте элемент `WebBrowser` (сконфигурированный по вашему усмотрению), `TextBox` (для ввода URL) и `Button` (для выполнения запроса HTTP). На рис. 4.26 показано поведение `WebBrowser` со свойством `Url`, установленным в `http://www.intertechtraining.com`.


**Рис. 4.26.** Элемент `WebBrowser`, отображающий домашнюю страницу Intertech Training

Единственный код, необходимый для отображения в WebBrowser входящего запроса HTTP, состоит в присваивании значения свойству Url, как показано в следующем обработчике события Click кнопки Button:

```
private void btnGO_Click(object sender, EventArgs e)
{
    // Установить URL по значению из элемента управления TextBox формы.
    myWebBrowser.Url = new System.Uri(txtUrl.Text);
}
```

На этом изучение виджетов из пространства имен System.Windows.Forms завершено. Хотя здесь не были описаны все возможные элементы пользовательского интерфейса, вы не должны иметь проблем с самостоятельным их исследованием. А теперь давайте посмотрим на процесс построения специальных (пользовательских) элементов управления Windows Forms.

---

**Исходный код.** Проект ExoticControls включен в подкаталог Bonus Chapter 4.

---

# Построение специальных элементов управления Windows Forms

Платформа .NET предлагает разработчикам очень простой способ построения специальных элементов пользовательского интерфейса. В отличие от элементов управления ActiveX, элементы управления Windows Forms не требуют громоздкой инфраструктуры COM или сложного управления памятью. Вместо этого разработчики .NET просто строят новый класс, унаследованный от UserControl, и наполняют его необходимым набором свойств, методов и событий. Чтобы продемонстрировать процесс, на следующих нескольких страницах вы сконструируете новый элемент управления под названием CarControl, используя Visual Studio.

---

**На заметку!** Как и в отношении любого приложения .NET, построить специальный элемент управления Windows Forms можно, не используя ничего помимо компилятора командной строки и простого текстового редактора. Специальные элементы управления располагаются в сборках *.dll, поэтому нужно специфицировать опцию /target:dll для csc.exe.

---

Для начала запустите Visual Studio и выберите новое рабочее пространство Windows Control Library (Библиотека элементов управления Windows) под названием CarControlLibrary (рис. 4.27).

Переименуйте начальный класс C# в CarControl. Подобно рабочему пространству проекта Windows Application, специальный элемент управления состоит из двух частичных классов. Файл *.Designer.cs содержит весь сгенерированный дизайнером код, а начальное определение частичного класса определяет тип-наследник System.Windows.Forms.UserControl:

```
namespace CarControlLibrary
{
    public partial class CarControl : UserControl
    {
        public CarControl()
        {
            InitializeComponent();
        }
    }
}
```

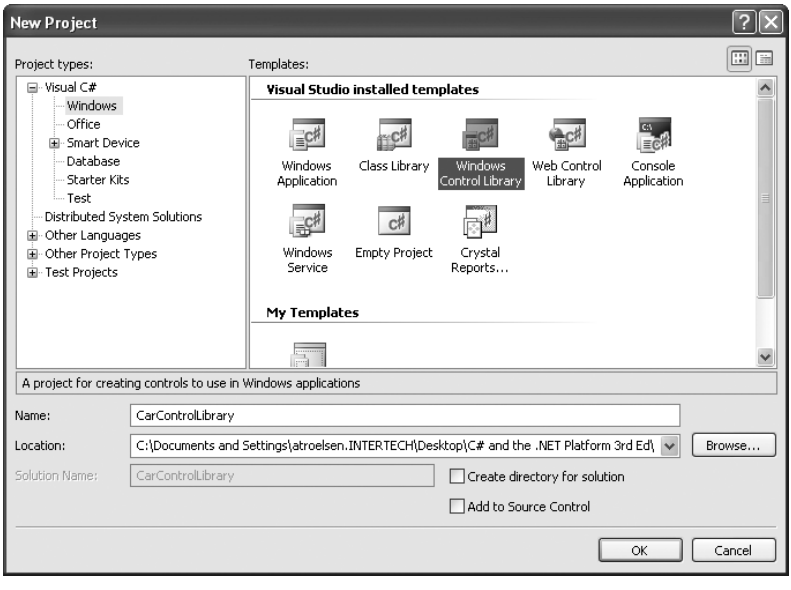


**Рис. 4.27.** Создание нового рабочего пространства Windows Control Library

Давайте очертим общую картину того, чего хотим достичь в этом примере. Тип `CarControl` отвечает за анимацию с помощью серии изображений, которые будут сменять друг друга в зависимости от внутреннего состояния автомобиля. Если текущая скорость автомобиля не превышает предельно допустимую, то `CarControl` выполняет цикл по трем битовым изображениям, которые визуализируют безопасное управление автомобилем. Если текущая скорость ниже предельной, менее чем на 10 миль в час, `CarControl` выполняет цикл по четырем изображениям, причем четвертое показывает разбивающийся автомобиль. И, наконец, если автомобиль превысит максимальную скорость, `CarControl` выполняет цикл по пяти изображениям, при этом пятое представляет разбитый автомобиль.

## Создание изображений

Исходя из приведенного описания дизайна, первой задачей будет создание набора из пяти файлов `*.bmp`, которые будут использоваться в цикле анимации. Если вы хотите создать собственные изображения, начните с выбора пункта меню Project⇨Add New Item и добавления пяти файлов битовых изображений. Можно воспользоваться файлами из загружаемого кода для этой главы. Первые три картинки (`Lemon1.bmp`, `Lemon2.bmp` и `Lemon3.bmp`) изображают автомобиль, который едет по дороге в спокойной и безопасной манере. Последние две (`AboutToBlow.bmp` и `EngineBlown.bmp`) представляют автомобиль, несущийся с предельной скоростью, и разбитый автомобиль.

## Построение пользовательского интерфейса времени проектирования

Следующий шаг — применение редактора времени проектирования для типа `CarControl`. Он похож на дизайнер форм, представляющий клиентскую область разрабатываемого элемента управления. Используя окно Toolbox, добавьте элемент типа `ImageList` для хранения каждого изображения (по имени `carImages`). Не беспокойтесь

о конфигурировании размера или местоположения элемента типа PictureBox, поскольку этот виджет вы позиционируете программно в пределах CarControl. Однако не забудьте установить свойство SizeMode элемента PictureBox в значение StretchImage через окно Properties. На рис. 4.28 показано, как это должно выглядеть.

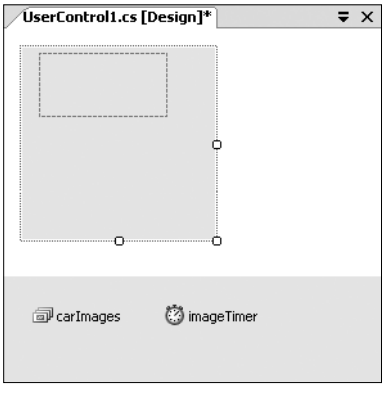


**Рис. 4.28.** Создание пользовательского интерфейса времени проектирования

Теперь, используя окно Properties, сконфигурируйте коллекцию Images элемента ImageList, добавив все картинки в список. Имейте в виду, что добавлять их нужно последовательно (Lemon1.bmp, Lemon2.bmp, Lemon3.bmp, AboutToBlow.bmp и EngineBlown.bmp), чтобы обеспечить линейный цикл анимации. Также имейте в виду, что ширина и высота файлов *.bmp по умолчанию, вставляемых Visual Studio, составляет 47×47 пикселей. Поэтому ImageSize и ImageList должны также иметь размер 47×47 (иначе вы получите визуализацию с искажениями). И, наконец, настройте состояние компонента Timer так, чтобы его свойство Interval было равно 200 и изначально отключено.

## Реализация основного CarControl

Выполнив эту предварительную подготовку пользовательского интерфейса, можете приступать к реализации членов типа. Для начала создайте новое общедоступное перечисление под названием AnimFrames, которое включает член, представляющий каждый элемент из списка ImageList. Это перечисление используется для определения текущей рамки для визуализации в PictureBox:

```
// Вспомогательное перечисление для изображений.
public enum AnimFrames
{
    Lemon1, Lemon2, Lemon3,
    AboutToBlow, EngineBlown
}
```

Тип CarControl поддерживает порядочное количество приватных членов данных, представляющих логику анимации. Вот их список:

```
public partial class CarControl : UserControl
{
    // Данные состояния.
    private AnimFrames currFrame = AnimFrames.Lemon1;
    private AnimFrames currMaxFrame = AnimFrames.Lemon3;
    private bool IsAnim;
    private int currSp = 50;
    private int maxSp = 100;
```

```
    private string carPetName= "Lemon";
    private Rectangle bottomRect = new Rectangle();
    public CarControl()
    {
        InitializeComponent();
    }
}
```

Как видите, есть элементы данных, представляющих текущую и максимальную скорость, название автомобиля и два члена типа `AnimFrames`. Переменная `curFrame` используется, чтобы специфицировать, какой член `ImageList` должен быть визуализирован в данный момент. Переменная `curMaxFrame` служит для отметки текущего верхнего предела в списке `ImageList` (напомним, что `CarControl` выполняет цикл по трем–пяти изображениям, в зависимости от текущей скорости). Элемент данных `IsAnim` используется для определения того, находится ли автомобиль в режиме анимации. И, наконец, имеется член `Rectangle` (`bottomRect`), который используется для представления нижней области типа `CarControl`. Позднее вы визуализируете здесь название автомобиля.

Чтобы разделить `CarControl` на две прямоугольных области, создайте приватную вспомогательную функцию по имени `StretchBox()`. Роль этого члена — вычислять корректный размер члена `bottomRect` и обеспечивать, чтобы виджет `PictureBox` растягивался приблизительно на верхнюю треть элемента `CarControl`.

```
private void StretchBox()
{
    // Сконфигурировать рамку картинок.
    currentImage.Top = 0;
    currentImage.Left = 0;
    currentImage.Height = this.Height - 50;
    currentImage.Width = this.Width;
    currentImage.Image = carImages.Images[(int)AnimFrames.Lemon1];

    // Вычислить размер нижнего прямоугольника.
    bottomRect.X = 0;
    bottomRect.Y = this.Height - 50;
    bottomRect.Height = this.Height - currentImage.Height;
    bottomRect.Width = this.Width;
}
```

Вычислив размеры каждого прямоугольника, вызовите `StretchBox()` в конструкторе по умолчанию:

```
public CarControl()
{
    InitializeComponent();
    StretchBox();
}
```

## Определение специальных событий

Тип `CarControl` поддерживает два события, которые передаются включающей форме, в зависимости от текущей скорости автомобиля. Первое из них, `AboutToBlow`, посылается, когда скорость `CarControl` приближается к верхнему пределу. `BlewUp` отправляется контейнеру, когда текущая скорость превышает допустимый максимум. Каждое из этих событий использует специальный делегат (`CarEventHandler`), который может хранить адрес любого метода, возвращающего `void` и принимающего единственный `System.String` в качестве параметра. Очень скоро вы инициируете эти события, а пока добавьте следующие члены в общедоступный раздел `CarControl`:

```
// События и делегаты Car.
public delegate void CarEventHandler(string msg);
public event CarEventHandler AboutToBlow;
public event CarEventHandler BlewUp;
```

**На заметку!** Напомним, что "правильный" делегат должен специфицировать два аргумента, первый из которых — `System.Object` (представляющий отправителя), а второй — тип-наследник `System.EventArgs`. В этом примере, однако, мы нарушим это правило.

## Определение специальных свойств

Подобно любому типу класса, специальные элементы управления могут определять множество свойств, чтобы позволить внешнему миру взаимодействовать с состоянием виджета. Для текущих потребностей интересует определение только трех свойств. Для начала имеется `Animate`. Это свойство включает и отключает компонент `Timer`:

```
// Используется для конфигурирования внутреннего типа Timer.
public bool Animate
{
    get {return IsAnim;}
    set
    {
        IsAnim = value;
        imageTimer.Enabled = IsAnim;
    }
}
```

Свойство `PetName` особых комментариев не требует. Однако обратите внимание, что когда пользователь устанавливает имя автомобиля, вы вызываете `Invalidate()`, чтобы визуализировать имя `CarControl` в нижней прямоугольной области виджета.

```
// Конфигурирует название.
public string PetName
{
    get{return carPetName;}
    set
    {
        carPetName = value;
        Invalidate();
    }
}
```

Далее имеется свойство `Speed`. В дополнение к простой модификации члена данных `currSp`, `Speed` — это сущность, которая инициирует события `AboutToBlow` и `BlewUp` в зависимости от текущей скорости `CarControl`. Вот полная логика:

```
// Изменить currSp и currMaxFrame и инициировать события.
public int Speed
{
    get { return currSp; }
    set
    {
        // Скорость в безопасных пределах?
        if (currSp <= maxSp)
        {
            currSp = value;
            currMaxFrame = AnimFrames.Lemon3;
        }
```

```
            // Приближается к предельной?
            if ((maxSp - currSp) <= 10)
            {
                if (AboutToBlow != null)
                {
                    AboutToBlow("Slow down dude!");
                    currMaxFrame = AnimFrames.AboutToBlow;
                }
            }

            // Превышена?
            if (currSp >= maxSp)
            {
                currSp = maxSp;
                if (BlewUp != null)
                {
                    BlewUp("Ug...you're toast...");
                    currMaxFrame = AnimFrames.EngineBlown;
                }
            }
        }
    }
```

Как видите, если текущая скорость на 10 и менее миль в час приближается к максимальной, инициируется событие `AboutToBlow` и изменяется верхний лимит кадров анимации `AnimFrames.AboutToBlow`. Если пользователь превысил предел скорости автомобиля, инициируется событие `BlewUp` и устанавливается верхний лимит кадров анимации `AnimFrames.EngineBlown`. Если же скорость ниже предельно допустимой, верхним лимитом анимации остается `AnimFrames.Lemon3`.

## Управление анимацией

Следующая деталь, которой нужно уделить внимание — обеспечить работу типа `Timer` в пределах заданного предела анимации, чтобы визуализировать содержимое `PictureBox`. Вспомните, что количество кадров, которые нужно отобразить в цикле, зависит от текущей скорости автомобиля. Вы должны изменять картинку в `PictureBox`, только если свойство `Animate` установлено в `true`. Начните с обработки события `Tick` для типа `Timer` и реализуйте обработчик следующим образом:

```
private void imageTimer_Tick(object sender, EventArgs e)
{
    if(IsAnim)
        currentImage.Image = carImages.Images[(int)currFrame];

    // Сменить кадр.
    int nextFrame = ((int)currFrame) + 1;
    currFrame = (AnimFrames)nextFrame;
    if (currFrame > currMaxFrame)
        currFrame = AnimFrames.Lemon1;
}
```

## Визуализация имени автомобиля

До завершения работы над элементом управления осталась последняя деталь: визуализация названия автомобиля. Для этого обработайте событие `Paint` для `CarControl`, и внутри обработчика визуализируйте название `CarControl` в нижней прямоугольной части клиентской области:

```
private void CarControl_Paint(object sender, PaintEventArgs e)
{
    // Визуализировать название в нижней части элемента управления.
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.GreenYellow, bottomRect);
    g.DrawString(PetName, new Font("Times New Roman", 15),
       Brushes.Black, bottomRect);
}
```

На этом начальная версия `CarControl` готова. Соберите проект.

# Тестирование типа `CarControl`

Когда вы запустите проект Windows Control Library внутри Visual Studio, то UserControl TestContainer (управляемая замена старого ActiveX Control TestContainer) автоматически загрузит элемент управления в его текстовую среду дизайнера. Как показано на рис. 4.29, этот инструмент позволяет установить каждое специальное свойство (а также унаследованные свойства) для тестовых целей.

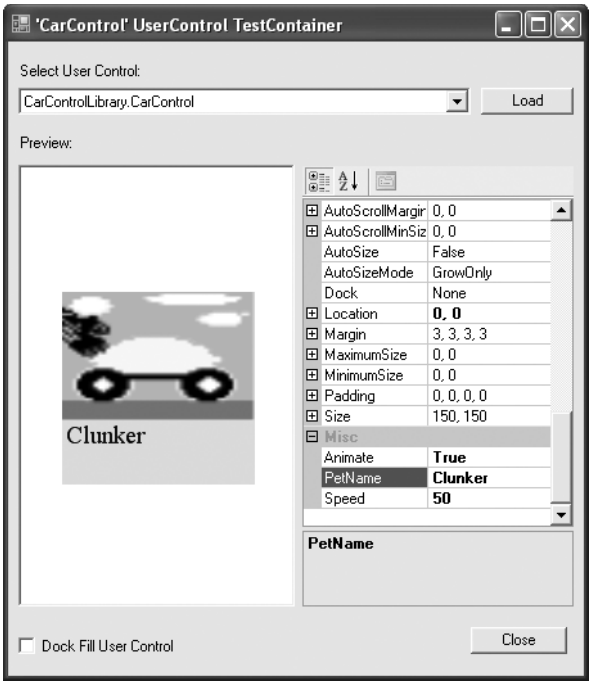


**Рис. 4.29.** Тестирование `CarControl` в среде UserControl TestContainer

После установки свойства `Animate` в `true` вы должны увидеть работающий цикл анимации `CarControl`, демонстрирующий первые три файла `*.bmp`. Что нельзя делать с этой утилитой тестирования — так это обрабатывать события. Чтобы протестировать этот аспект вашего элемента, придется построить специальную форму.

# Построение специальной формы-хоста для CarControl

Как и с типами .NET, теперь можно использовать новый специальный элемент управления из любого языка, нацеленного на CLR. Начните с закрытия текущего рабочего пространства и создания нового проекта C# Windows Application по имени CarControlTestForm. Чтобы обращаться к специальным элементам управления из интегрированной среды Visual Studio, выполните щелчок правой кнопкой мыши в любом месте окна Toolbox и выберите в контекстном меню пункт Choose Item (Выбрать элемент). Используя кнопку Browse (Обзор) на вкладке .NET Framework Components (Компоненты .NET Framework), перейдите к библиотеке CarControlLibrary.dll. После щелчка на кнопке OK в панели Toolbox появится новая пиктограмма под названием CarControl.

Затем поместите новый виджет CarControl в дизайнер форм. Обратите внимание, что свойства Animate, PetName и Speed доступны в окне Properties. Как и в UserControl TestContainer, элемент управления активен во время проектирования. Поэтому, установив свойство Animate в true, вы обнаружите, что автомобиль начнет анимироваться в дизайнере форм.

Завершив конфигурирование начального состояния CarControl, добавьте дополнительные виджеты GUI, которые позволят пользователю увеличивать и уменьшать скорость автомобиля и просматривать строковые данные, посылаемые входящими событиями, а также текущую скорость автомобиля (для этих целей отлично подходят элементы управления Label). Один из возможных вариантов дизайна GUI показан на рис. 4.30.

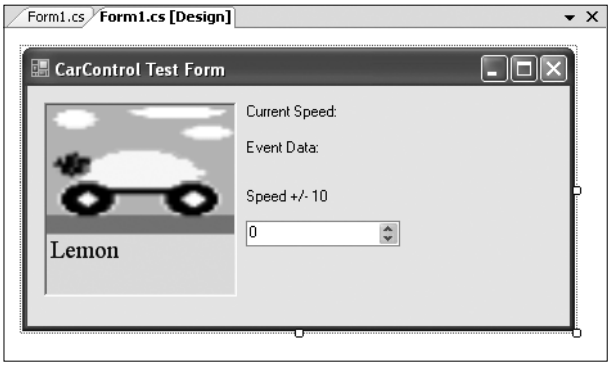


**Рис. 4.30.** GUI клиентской стороны

Если создали такой же GUI, как приведенный на рис. 4.30, то код типа-наследника Form будет достаточно простым (предполагается, что каждое событие CarControl было обработано в окне Properties):

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
        lblCurrentSpeed.Text = string.Format("Current Speed: {0}",
            this.myCarControl.Speed.ToString());
            numericUpDownCarSpeed.Value = myCarControl.Speed;
    }
```

```
    private void numericUpDownCarSpeed_ValueChanged(object sender, EventArgs e)
    {
        // Предполагаем, что минимальное значение NumericUpDown равно 0,
        // а максимальное - 300.
        this.myCarControl.Speed = (int)numericUpDownCarSpeed.Value;
        lblCurrentSpeed.Text = string.Format("Current Speed: {0}",
            this.myCarControl.Speed.ToString());
    }
    private void myCarControl_AboutToBlow(string msg)
    { lblEventData.Text = string.Format("Event Data: {0}", msg); }
    private void myCarControl_BlewUp(string msg)
    { lblEventData.Text = string.Format("Event Data: {0}", msg); }
}
```

После этого вы готовы запустить клиентское приложение и взаимодействовать с `CarControl`. Как было показано, построение и использование специальных элементов управления — довольно простая задача, если обладать достаточными знаниями ООП, системы типов .NET, GDI+ (т.е. `System.Drawing.dll`) и Windows Forms.

Хотя вы теперь имеете достаточно информации для продолжения исследования процесса разработки элементов управления .NET Windows Forms, есть еще один дополнительный программистский аспект, с которым надо поработать: функциональность времени проектирования. Вначале следует ознакомиться с ролью пространства имен `System.ConponentModel`.

# Роль пространства имен `System.ConponentModel`

Пространство имен `System.ConponentModel` определяет ряд атрибутов (помимо прочих типов), которые позволяют описать поведение специальных элементов управления во время проектирования. Например, вы можете предоставить текстовое описание каждого свойства, определить событие по умолчанию либо сгруппировать взаимосвязанные свойства или события в специальную категорию в целях отображения внутри окна Properties. Если вы заинтересованы в подобного рода модификациях, следует применить основные атрибуты, показанные в табл. 4.12.

**Таблица 4.12. Избранные атрибуты `System.ComponentModel`**

| Атрибут | Применение | Назначение |
|---|---|---|
| `BrowsableAttribute` | Свойства и события | Специфицирует, должно ли свойство или событие отображаться в браузере свойств. По умолчанию все специальные свойства и события могут просматриваться. |
| `CategoryAttribute` | Свойства и события | Специфицирует название категории, к которой следует отнести свойство или событие. |
| `DescriptionAttribute` | Свойства и события | Определяет небольшой блок текста для отображения внизу браузера свойств, когда пользователь выбирает свойство или событие. |
| `DefaultPropertyAttribute` | Свойства | Специфицирует свойство компонента по умолчанию. Это свойство выбирается в браузере свойств, когда пользователь выбирает элемент управления. |

*Окончание табл. 4.12*

| Атрибут | Применение | Назначение |
|---|---|---|
| `DefaultValueAttribute` | Свойства | Определяет значение свойства по умолчанию, которое будет применено, когда элемент управления "сбрасывается" внутри IDE-среды. |
| `DefaultEventAttribute` | События | Специфицирует событие по умолчанию для компонента. Когда программист выполняет двойной щелчок на элементе управления, автоматически генерируется заготовка кода для события по умолчанию. |

## Усовершенствование внешнего вида `CarControl` времени проектирования

Чтобы проиллюстрировать применение некоторых из этих новых атрибутов, закройте проект `CarControlTestForm` и откройте заново проект `CarControlLibrary`. Давайте создадим специальную категорию по имени `"Car Configuration"` (Конфигурация автомобиля), к которой будут относиться все свойства и события `CarControl`. Кроме того, добавьте к каждому свойству и событию типа `CarControl` атрибуты `[Category]`, `[DefaultValue]` и `[Description]`:

```
public partial class CarControl : UserControl
{
    ...
    [Category("Car Configuration"),
    Description("Sent when the car is approaching terminal speed.")]
    public event CarEventHandler AboutToBlow;
    ...
    [Category("Car Configuration"),
    Description("Name your car!"),
    DefaultValue("Lemon")]
    public string PetName {...}
    ...
}
```

Теперь давайте проясним, что означает присваивание свойству *значения по умолчанию*. Просто говоря, атрибут `[DefaultValue]` *не* гарантирует, что лежащее в основе свойство, оснащенное этим атрибутом, будет автоматически инициализировано этим самым значением по умолчанию. Поэтому, хотя для свойства `PetName` было указано значение по умолчанию "No Name", переменная-член `carPetName` не будет установлена в "Lemon", если только не сделать это в конструкторе типа или с использованием синтаксиса инициализации членов (как делалось до сих пор):

```
private string carPetName= "Lemon";
```

Вместо этого атрибут `[DefaultValue]` вступает в игру, когда программист "сбрасывает" значение данного свойства в окне Properties. Чтобы сбросить свойство с использованием Visual Studio, выберите интересующее свойство, выполните на нем щелчок правой кнопкой мыши и выберите в контекстном меню пункт Reset (Сброс). Обратите внимание на рис. 4.31, что значение `[Description]` появляется в нижней панели окна Properties.

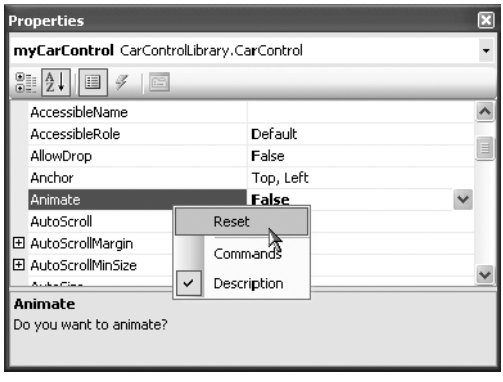


**Рис. 4.31.** Сброс свойства в значение по умолчанию

Атрибут `[Category]` будет реализован, только если программист выберет категоризированное представление окна Properties (в противоположность алфавитному представлению по умолчанию), как показано на рис. 4.32.

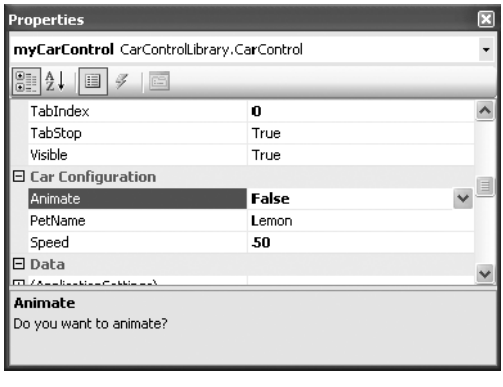


**Рис. 4.32.** Специальная категория

## Определение свойства по умолчанию и события по умолчанию

В дополнение к описанию группирования сходных свойств в общую категорию, может понадобится сконфигурировать элементы управления для поддержки поведения по умолчанию. Данный элемент управления может поддерживать свойство по умолчанию. Определив свойство по умолчанию для класса с помощью атрибута `[DefaultProperty]`, как показано ниже:

```
//Пометить свойство элемента управления по умолчанию.
[DefaultProperty("Animate")]
public partial class CarControl : UserControl
{...}
```

вы тем самым гарантируете, что когда пользователь выберет этот элемент управления во время проектирования, свойство `Animate` будет автоматически подсвечено в окне Properties. Аналогично, конфигурирование элемента управления событием по умолчанию, как здесь:

```
// Пометить событие и свойство по умолчанию для данного элемента управления.
[DefaultEvent("AboutToBlow"),
DefaultProperty("Animate")]
public partial class CarControl : UserControl
{...}
```

гарантирует, что при двойном щелчке пользователя на виджете во время проектирования, заготовка кода будет автоматически сгенерирована для обработки события по умолчанию (это объясняет, почему двойной щелчок на `Button` вызывает автоматическое создание обработчика события `Click`, двойной щелчок на `Form` автоматически генерирует обработчик события `Load`, и т.д.).

## Спецификация специальной пиктограммы Toolbox

Последняя возможность времени проектирования для любого тщательно спроектированного пользовательского элемента управления — специальная битовая пиктограмма для отображения в панели инструментов. В настоящее время, когда пользователь выберет `CarControl`, среда IDE отобразит этот тип в Toolbox, используя стандартную пиктограмму с шестерней. Если вместо этого нужно специфицировать какое-то специальное изображение, первым шагом будет вставка нового файла `*.bmp` в проект (`CarControl.bmp`), который должен иметь размер 16 16 пикселей (задается свойствами `Width` и `Height`). В текущем примере просто повторно используется картинка `Car` из примера `TreeView`.

Создав необходимую картинку, используйте атрибут `[ToolboxBitmap]` (применяемый на уровне типа), чтобы назначить эту картинку элементу управления. Первый аргумент конструктора атрибута — информация о типе самого элемента управления, а второй аргумент — дружественное имя файла `*.bmp`.

```
[DefaultEvent("AboutToBlow"),
DefaultProperty("Animate"),
ToolboxBitmap(typeof(CarControl), "CarControl")]
public partial class CarControl : UserControl
{...}
```

И последний шаг, который осталось предпринять — установить значение Build Action (Действие сборки) образа пиктограммы элемента управления в Embedded Resource (Встроенный ресурс) в окне Properties, чтобы обеспечить встраивание изображения в сборку (рис. 4.33).

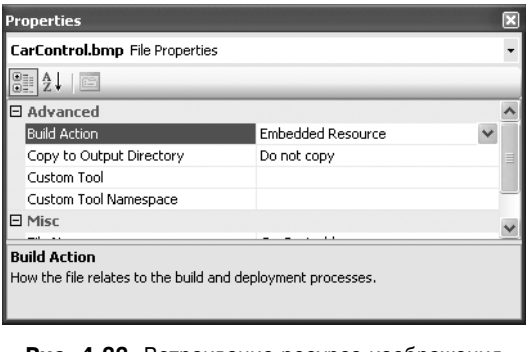


**Рис. 4.33.** Встраивание ресурса изображения

**На заметку!** Причина ручного встраивания файла `*.bmp` (в отличие от случая использования типа `ImageList`) в том, что файл `CarControl.bmp` элементу пользовательского интерфейса не назначается во время проектирования, а потому лежащий в основе файл `*.resx` не будет обновлен автоматически.

После перекомпиляции библиотеки элементов управления Windows можно загрузить предыдущий проект `CarControlTestForm`. Выполните щелчок правой кнопкой мыши на пиктограмме `CarControl` внутри Toolbox и выберите в контекстном меню пункт Delete (Удалить). Затем заново добавьте виджет `CarControl` в Toolbox (щелчком правой кнопкой мыши и выбором в контекстном меню пункта Choose Item). На этот раз вы должны увидеть специальную пиктограмму элемента в панели инструментов (рис. 4.34).


**Рис. 4.34.** Специальная пиктограмма элемента в панели инструментов

Итак, на этом рассмотрение процесса построения пользовательских элементов управления Windows Forms завершено. Представленный пример, посвященный вездесущей автомобильной теме, должен был привлечь внимание к разработке специальных элементов управления. Подумайте о пользе специального элемента управления, который отображает круговую диаграмму в зависимости от текущего содержимого определенной таблицы базы данных, или элемента, расширяющего функциональность стандартных виджетов.

**Исходный код.** Проекты `CarControlLibrary` и `CarControlTestForm` включены в подкаталог `Bonus Chapter 4`.

# Построение специальных диалоговых окон

Теперь, понимания устройство основных элементов управления Windows Forms и процесс построения специальных элементов управления, давайте рассмотрим процесс конструирования специальных пользовательских диалоговых окон. Все, что вы до сих пор изучили о Windows Forms, непосредственно применимо при программировании диалоговых окон. В основном, создание (и отображение) диалогового окна не сложнее вставки новой формы в проект.

В пространстве имен `System.Windows.Forms` не существует базового класса “Dialog”. Вместо этого диалоговые окна в Windows Forms являются просто типами-наследниками класса `Form`. Например, многие диалоговые окна намеренно не допускают изменения размера; поэтому обычно необходимо устанавливать свойство `FormBorderStyle` рав-

ным FormBorderStyle.FixedDialog. К тому же диалоговые окна обычно устанавливают свойства MinimizeBox и MaximizeBox в false. Таким образом, диалоговое окно конфигурируется с постоянным размером. И, наконец, установка свойства ShowInTaskbar в false, предотвращает отображение формы в панели задач Windows.

Чтобы проиллюстрировать построение и управление диалоговыми окнами, создайте новый проект Windows Application по имени SimpleModalDialog. Главный тип Form поддерживает MainStrip, содержащий пункт меню File⇨Exit (Файл⇨Выход), а также Tools⇨Configure (Сервис⇨Конфигурировать). Постройте этот пользовательский интерфейс и обработайте событие Click для упомянутых пунктов меню. Также определите в Form строковую переменную-член (userMessage) и визуализируйте эти данные внутри обработчика событий Paint главной формы. Ниже показан код внутри файла MainForm.cs:

```
public partial class MainWindow : Form
{
    private string userMessage = "Default Message";
    public MainWindow()
    {
        InitializeComponent();
    }
    private void exitToolStripMenuItem_Click(object sender, EventArgs e)
    {
        Application.Exit();
    }
    private void configureToolStripMenuItem_Click(object sender, EventArgs e)
    {
        // Этот метод реализуем чуть позже...
    }
    private void MainWindow_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.DrawString(userMessage, new Font("Times New Roman", 24), Brushes.DarkBlue,
            50, 50);
    }
}
```

Теперь добавьте новый тип Form в текущий проект, используя пункт меню Project⇨Add Windows Form (Проект⇨Добавить форму Windows), и назовите его UserMessageDialog.cs. Установите свойства ShowInTaskbar, MinimizeBox и MaximizeBox в false. Затем постройте пользовательский интерфейс, состоящий из двух элементов типа Button (для кнопок OK и Cancel), одного TextBox (чтобы позволить пользователю вводить сообщение) и Label с инструкциями. На рис. 4.35 показан возможный вариант интерфейса новой формы.

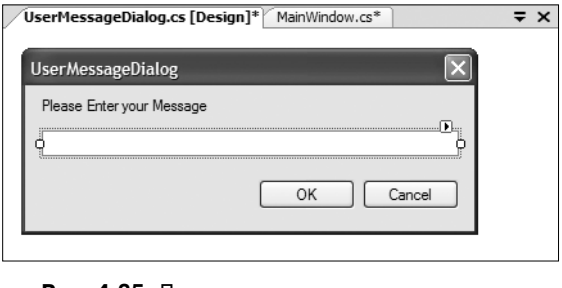


**Рис. 4.35.** Пользовательское диалоговое окно

И, наконец, представьте значение Text элемента TextBox с помощью специального свойства по имени Message:

```
public partial class UserMessageDialog : Form
{
   public UserMessageDialog()
   {
      InitializeComponent();
   }
   public string Message
   {
      set { txtUserInput.Text = value; }
      get { return txtUserInput.Text; }
   }
}
```

## Свойство DialogResult

И последнее: выбрав кнопку OK и используя окно Properties, найдите свойство DialogResult. Присвойте DialogResult.OK кнопке OK и DialogResult.Cancel кнопке Cancel. Формально можно присвоить свойству DialogResult любое значение из перечисления DialogResult:

```
public enum System.Windows.Forms.DialogResult
{
   Abort, Cancel, Ignore, No,
   None, OK, Retry, Yes
}
```

Но что именно означает присваивание значения свойству DialogResult кнопки Button? Это свойство может быть установлено для любого элемента Button (как и всей Form в целом) и позволяет родительской форме определить, какая кнопка была нажата в диалоге конечным пользователем. Для иллюстрации обновите обработчик меню Tools⇨Configure в MainForm следующим образом:

```
private void configureToolStripMenuItem_Click(object sender, EventArgs e)
{
   // Создать экземпляр UserMessageDialog.
   UserMessageDialog dlg = new UserMessageDialog();
   // Поместить текущее сообщение в TextBox.
   dlg.Message = userMessage;
   // Если пользователь щелкнул на кнопке OK, отобразить это сообщение.
   if (DialogResult.OK == dlg.ShowDialog())
   {
      userMessage = dlg.Message;
      Invalidate();
   }
   // Очистить внутренние виджеты немедленно, вместо того,
   // чтобы ждать, когда сборщик мусора уничтожит объект.
   dlg.Dispose();
}
```

Здесь отображается UserMessageDialog с помощью вызова ShowDialog(). Этот метод запустит форму в виде *модального* диалогового окна, а это, как известно, означает, что пользователь не сможет активизировать главную форму до тех пор, пока не закроет диалоговое окно. Как только он закроет его (щелкнув на кнопке OK или Cancel), форма диалогового окна исчезнет с экрана, но останется в памяти. Поэтому можно запросить у экземпляра UserMessageDialog (dlg) его значение Message, если пользователь щелкнул на кнопке OK. В этом случае отображается введенное сообщение. В противном случае ничего не делается.

**На заметку!** Для отображения немодального диалогового окно (которое позволяет пользователю переключаться между родительской формой и формой диалога) следует вызывать `Show()` вместо `ShowDialog()`.

## Наследование форм

Одним из наиболее привлекательных аспектов построения диалоговых окон в Windows Forms является *наследование форм*. Как известно, наследование — один из принципов ООП, который позволяет одному классу расширять функциональность другого класса. Обычно когда идет речь о наследовании, имеется в виду один неграфический тип (например, `SportsCar`), унаследованный от другого неграфического типа (например, `Car`). Однако в мире Windows Forms допускается наследование одной формой другой формы, с наследованием всех виджетов базового класса вместе с реализацией.

Наследование уровня формы — очень мощная техника, поскольку позволяет строить базовую форму, которая предоставляет функциональность уровня ядра для семейства связанных диалоговых окон. Если вы соберете эти формы базового уровня в сборку .NET, то другие члены вашей команды смогут расширять эти типы, используя язык .NET по своему выбору.

Для иллюстрации предположим, что необходимо создать подкласс `UserMessaheDialog`, чтобы построить новое диалоговое окно, которое также позволит пользователю специфицировать необходимость в отображении сообщения курсивом. Для этого активизируйте пункт меню Project⇨Add Windows Form, но на этот раз добавьте новую унаследованную форму (Inherited Form) по имени `ItalicUserMessageDialog.cs` (рис.4.36).

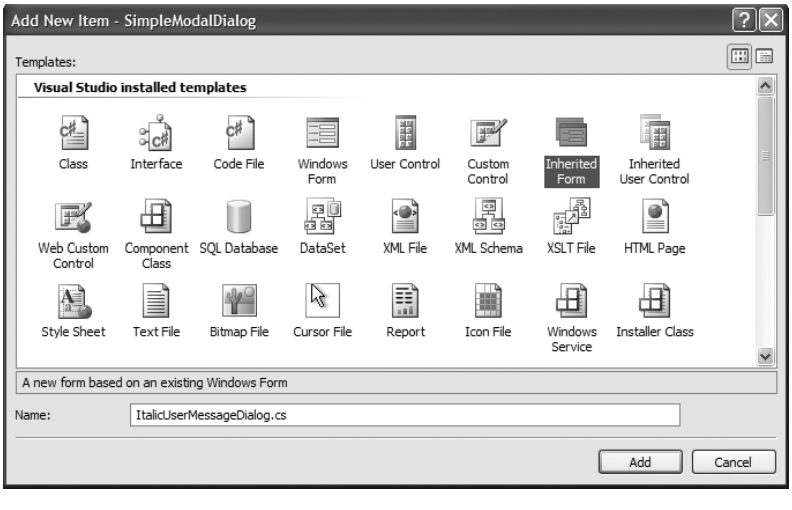


**Рис. 4.36.** Добавление в проект унаследованной формы

После щелчка на кнопке Add (Добавить) вы увидите *утилиту выбора унаследованной сущности*, которая позволяет выбрать `Form` из текущего проекта либо из существующей сборки через кнопку Browse. Для данного примера выберите существующий тип `UserMessageDialog`. Вы увидите, что новый тип `Form` расширяет имеющийся тип диалога вместо наследования непосредственно от класса `Form`. Эту производную форму можно расширить любым способом по своему усмотрению. В целях тестирования просто добавьте новый элемент управления `CheckBox` (`checkBoxItalic`), который будет управлять свойством по имени `Italic`:

```
public partial class ItalicUserMessageDialog :
    SimpleModalDialog.UserMessageDialog
{
    public ItalicUserMessageDialog()
    {
        InitializeComponent();
    }
    public bool Italic
    {
        set { checkBoxItalic.Checked = value; }
        get { return checkBoxItalic.Checked; }
    }
}
```

Теперь, поскольку было выполнено наследование от базового типа `UserMessageDialog`, модифицируйте `MainForm`, добавив новое свойство `Italic`. Просто добавьте новую булевскую переменную-член, которая будет использоваться для построения объекта `Font` с курсивным начертанием, и обновите обработчик события `Click` меню Tools⇨Configure, чтобы использовался `ItalicUserMessageDialog`. Ниже показаны необходимые изменения кода:

```
public partial class MainWindow : Form
{
    private string userMessage = "Default Message";
    private bool textIsItalic = false;
...
    private void configureToolStripMenuItem_Click(object sender, EventArgs e)
    {
        ItalicUserMessageDialog dlg = new ItalicUserMessageDialog();
        dlg.Message = userMessage;
        dlg.Italic = textIsItalic;
        // Если пользователь щелкнул на кнопке OK, отобразить сообщение.
        if (DialogResult.OK == dlg.ShowDialog())
        {
            userMessage = dlg.Message;
            textIsItalic = dlg.Italic;
            Invalidate();
        }
        // Очистить внутренние виджеты немедленно, вместо того,
        // чтобы ждать, когда сборщик мусора уничтожит объект.
        dlg.Dispose();
    }
    private void MainWindow_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        Font f = null;
        if(textIsItalic)
            f = new Font("Times New Roman", 24, FontStyle.Italic);
        else
            f = new Font("Times New Roman", 24);
        g.DrawString(userMessage, f, Brushes.DarkBlue,
            50, 50);
    }
}
```

**Исходный код.** Приложение `SimpleModalDialog` включено в подкаталог `Bonus Chapter 4`.

# Динамическое позиционирование элементов управления Windows Forms

В завершение этой главы давайте рассмотрим несколько приемов, которые можно использовать для контроля компоновки виджетов в Form. В основном, при построении типа Form предполагается, что элементы управления визуализируются по абсолютным позициям, в том смысле, что если вы поместили Button в дизайнере Form на 10 пикселей ниже и на 10 пикселей правее левого верхнего угла формы, то рассчитываете, что эта кнопка там и останется на все время ее существования.

Кстати, во время создания Form, содержащей элементы управления пользовательского интерфейса, следует решить, будет ли форма допускать изменения размера. Обычно главное окно делается изменяемого размера, в то время как диалоговые окна — нет. Напомним, что возможность изменять размер Form управляется свойством FormBorderStyle, которое может быть установлено в любое значение из перечисления FormBorderStyle.

```
public enum System.Windows.Forms.FormBorderStyle
{
   None, FixedSingle, Fixed3D,
   FixedDialog, Sizable,
   FixedToolWindow, SizableToolWindow
}
```

Предположим, что для формы установлены изменяемые размеры. Это вызывает некоторые интересные вопросы относительно содержащихся в ней элементов управления. Например, если пользователь сделает форму меньше размера прямоугольника, который необходим для отображения всех элементов управления, должны ли они пропорционально изменять свой размер (и, возможно, расположение), чтобы корректно выглядеть на форме?

## Свойство Anchor

В Windows Forms свойство Anchor используется для определения относительной фиксированной позиции, в которой элемент всегда должен отображаться. Каждый тип-наследник Control имеет свойство, которое может быть установлено в любые значения из перечисления AnchorStyles, описанного в табл. 4.13.

**Таблица 4.13. Значения перечисления AnchorStyles**

| Значение | Смысл |
|---|---|
| Bottom | Нижняя граница элемента управления прикрепляется к нижней границе ее контейнера. |
| Left | Левая граница элемента управления прикрепляется к левой границе ее контейнера. |
| None | Элемент управления не прикрепляется ни к одной из границ контейнера. |
| Right | Правая граница элемента управления прикрепляется к правой границе ее контейнера. |
| Top | Верхняя граница элемента управления прикрепляется к верхней границе ее контейнера. |

Чтобы прикрепить виджет к левому верхнему углу, можно объединять стили операцией битового "ИЛИ" (т.е. AnchorStyles.Top | AnchorStyles.Left). Идея, положенная в основу свойства Anchor, заключается в том, чтобы указать, какие грани элемента управления прикреплены к каким границам его контейнера. Например, если установить для Button следующее значение Anchor:

```
// Прикрепить виджет в правой границе.
myButton.Anchor = AnchorStyles.Right;
```

будет гарантироваться, что при изменении размера Form элемент Button сохранит позицию относительно правой грани Form.

## Свойство Dock

Другой аспект программирования Windows Forms касается установки поведения *стыковки* (docking) элементов управления. При желании можно установить свойство Dock, которое укажет, с какой границей (или границами) Form данный элемент должен быть стыкован. Значение, которое присваивается Dock, соблюдается независимо от текущих размеров Form. В табл. 4.14 описаны возможные варианты.

**Таблица 4.14. Значения DockStyle**

| Значение | Смысл |
|---|---|
| Bottom | Нижняя грань элемента управления стыкована с нижней гранью включающего элемента управления. |
| Fill | Все грани элемента управления стыкованы со всеми гранями включающего элемента управления и размеры установлены соответственно. |
| Left | Левая грань элемента управления стыкована с левой гранью включающего элемента управления. |
| None | Элемент управления не стыкован. |
| Right | Правая грань элемента управления стыкована с правой гранью включающего элемента управления. |
| Top | Верхняя грань элемента управления стыкована с верхней гранью включающего элемента управления. |

Таким образом, например, если необходимо, чтобы определенный виджет всегда был стыкован с левой гранью формы, следует написать так:

```
// Этот элемент всегда располагается в левой части Form,
// независимо от текущего размера Form.
myButton.Dock = DockStyle.Left;
```

Чтобы помочь понять влияние установки свойств Anchor и Dock, загружаемый код содержит проект под названием AnchoringControls. Построив и запустив это приложение, можно будет с помощью системы меню Form устанавливать различные значения AnchorStyles и DockStyle и наблюдать за поведением элемента Button (рис. 4.37).

Не забудьте изменить размер Form после изменения свойства Anchor, чтобы понаблюдать за поведением Button.

**Исходный код.** Приложение AnchoringControls включено в подкаталог Bonus Chapter 4.

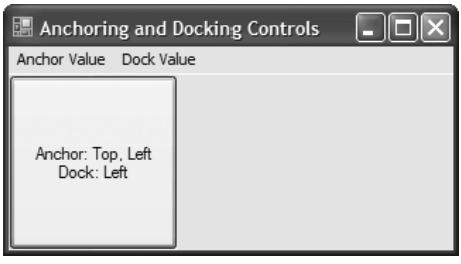


**Рис. 4.37.** Приложение `AnchoringControls`

## Табличная и потоковая компоновка

.NET 2.0 предлагает дополнительный способ управления компоновкой виджетов `Form` с использованием одного из двух диспетчеров компоновки. Типы `TableLayoutPanel` и `FlowLayoutPanel` могут быть прикреплены к клиентской области `Form` для размещения внутренних элементов управления. Например, предположим, что вы поместили виджет `FlowLayoutPanel` в дизайнер форм и сконфигурировали его для стыковки полностью внутри родительской `Form` (рис. 4.38).

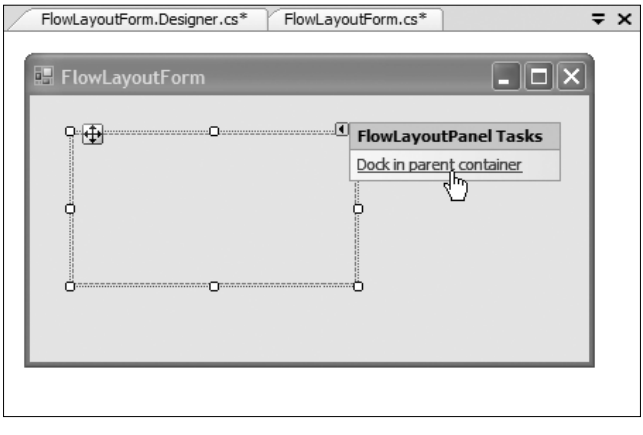


**Рис. 4.38.** Стыковка `FlowLayoutPanel` в `Form`

Теперь добавьте десять новых элементов типа `Button` внутрь `FlowLayoutPanel` с использованием дизайнера форм. Если теперь запустить приложение, легко заметить, что десять `Button` автоматически реорганизуются в манере, очень похожей на стандартный HTML.

С другой стороны, если создать `Form`, содержащую `TableLayoutPanel`, можно строить пользовательский интерфейс, разбитый на "ячейки" (рис. 4.39).

Выбрав пункт меню Edit Rows and Columns (Редактировать строки и колонки) в дизайнере форм (как показано на рис. 4.39), можно управлять общим форматом `TableLayoutPanel` на основе отдельных ячеек (рис. 4.40).

Единственный способ увидеть эффект от применения типа `TableLayoutPanel` — это попробовать самому. Заинтересованные читатели справятся с этой задачей самостоятельно.

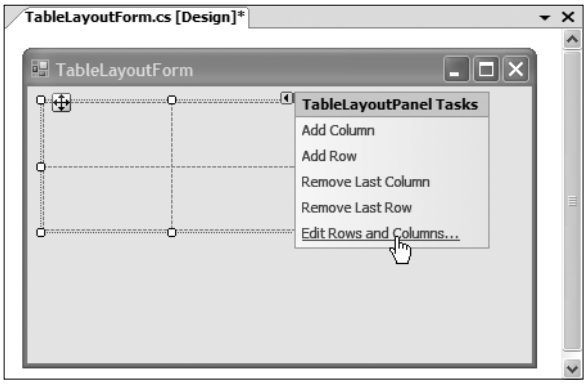


**Рис. 4.39.** Тип `TableLayoutPanel`

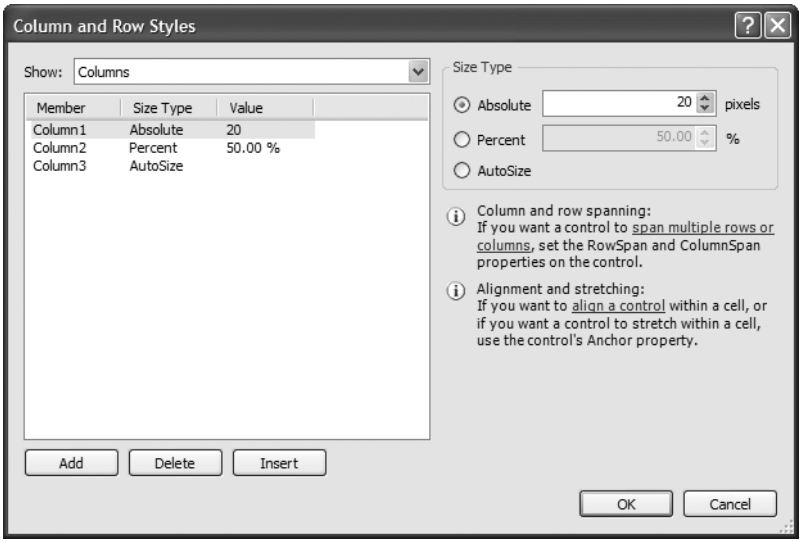


**Рис. 4.40.** Конфигурирование ячеек типа `TableLayoutPanel`

# Резюме

Эта глава завершает знакомство с пространством имен Windows Forms рассмотрением многочисленных виджетов GUI — от самых простых (вроде `Label`) до наиболее экзотических (наподобие `TreeView`). После рассмотрения многочисленных типов элементов управления мы ознакомились с конструированием специальных элементов управления, включая время проектирования.

Во второй половине главы был описан процесс построения специальных диалоговых окон и наследование новых типов `Form` от существующих. Эта глава завершилась кратким обзором поведения стыковки, связанного со свойствами `Anchor` и `Dock`, которые можно использовать для организации специфической компоновки типов GUI, а также новых диспетчеров компоновки .NET 2.0.