ПРИЛОЖЕНИЕ Г

Веб-элементы управления, мастер-страницы и темы ASP.NET

Основное внимание в предыдущем приложении было сосредоточено на общей структуре страницы Web Forms и роли класса `Page`. В текущем приложении мы погрузимся в детали *веб-элементов управления*, которые составляют пользовательский интерфейс страницы. После исследования общей природы элемента управления Web Forms вы узнаете, как использовать многие элементы пользовательского интерфейса, включая элементы управления проверкой достоверности и разнообразные приемы привязки данных.

Значительная порция приложения будет посвящена обсуждению роли *мастер-страниц*, которые обеспечивают упрощенный способ установления общего скелета пользовательского интерфейса, повторяющегося на страницах веб-сайта. С темой мастер-страниц тесно связано применение элементов управления навигацией по сайту (и файла `.sitemap`), которые позволяют определять навигационную структуру многостраничного сайта посредством файла XML серверной стороны.

В завершение вы ознакомитесь с ролью тем Web Forms. Концептуально темы служат той же цели, что и каскадные таблицы стилей (CSS); однако темы Web Forms применяются на веб-сервере (в противоположность клиентскому браузеру) и по этой причине имеют доступ к ресурсам серверной стороны.

Природа веб-элементов управления

Главным преимуществом инфраструктуры Web Forms является возможность собирать пользовательский интерфейс из страниц с использованием типов, определенных в пространстве имен `System.Web.UI.WebControls`. Как было показано, такие элементы управления (называемые *серверными элементами управления, веб-элементами управления* или *элементами управления Web Forms*) исключительно полезны в том, что автоматически генерируют необходимую разметку HTML для запрашивающего браузера и открывают доступ к набору событий, которые могут быть обработаны на веб-сервере. Более того, поскольку каждый элемент управления Web Forms имеет соответствующий класс в пространстве имен `System.Web.UI.WebControls`, им можно манипулировать в объектно-ориентированной манере.

При конфигурировании свойств веб-элемента управления в окне Properties (Свойства) среды Visual Studio изменения фиксируются в открывающем дескрипторе данного эле-

мента внутри файла .aspx как последовательность пар "имя-значение". Таким образом, если добавить новый элемент TextBox в визуальном конструкторе и изменить его свойства ID, BorderStyle, BorderWidth, BackColor и Text, то открывающий дескриптор <asp:TextBox> будет соответствующим образом модифицирован (тем не менее, обратите внимание, что значение Text становится внутренним текстом в области TextBox):

```
<asp:TextBox ID="txtNameTextBox" runat="server" BackColor="#C0FFC0"
     BorderStyle="Dotted" BorderWidth="3px">Enter Your Name</asp:TextBox>
```

Учитывая, что объявление веб-элемента управления в итоге становится переменной-членом типа из пространства имен System.Web.UI.WebControls (посредством цикла динамической компиляции, упомянутого в приложении B), с членами такого типа можно взаимодействовать внутри блока <script> серверной стороны или более распространенным способом — через файл отделенного кода страницы. Следовательно, если добавить в файл .aspx новый элемент управления Button, то можно написать обработчик события Click серверной стороны, в котором будет изменяться цвет фона элемента TextBox:

```
partial class _Default : System.Web.UI.Page
{
  protected void btnChangeTextBoxColor_Click(object sender, EventArgs e)
  {
    // Изменить цвет фона объекта TextBox в коде.
    this.txtNameTextBox.BackColor = System.Drawing.Color.DarkBlue;
  }
}
```

Все элементы управления Web Forms в конечном счете являются производными от общего базового класса System.Web.UI.WenControls.WebControl, который в свою очередь унаследован от System.Web.UI.Control (а тот — от System.Object). Классы Control и WebControl определяют несколько свойств, общих для всех элементов управления серверной стороны. Прежде чем изучать унаследованную функциональность, давайте формализуем понятие обработки события серверной стороны.

Обработка события серверной стороны

При текущем состоянии World Wide Web обойтись без понимания фундаментальной природы взаимодействия браузеров и веб-серверов невозможно. Всякий раз, когда эти две сущности взаимодействуют, происходит лишенный состояния цикл "запрос/ответ" HTTP. Хотя серверные элементы управления Web Forms выполняют немалую работу, изолируя вас от низкоуровневых деталей протокола HTTP, всегда помните о том, что восприятие World Wide Web как управляемой событиями сущности — всего лишь маскировка, которую обеспечивает платформа .NET. Здесь нет ничего общего с управляемой событиями моделью, которая поддерживается основанной на Windows инфраструктурой для построения графических пользовательских интерфейсов, такой как WPF.

Например, несмотря на то, что в пространстве имен System.Windows.Controls из WPF и в пространстве имен System.Web.UI.WebControls из Web Forms определены классы с совпадающими простыми именами (Button, TextBox, Label и т.д.), они не предлагают одинаковые наборы свойств, методов или событий. Скажем, обработать событие MouseMove серверной стороны, когда пользователь перемещает курсор над элементом управления Button из Web Forms, не получится.

Суть в том, что заданный элемент управления Web Forms будет открывать доступ к ограниченному набору событий, которые в итоге приводят к обратной отправке веб-серверу. Любая необходимая обработка событий клиентской стороны потребует написания фрагментов сценарного кода JavaScript/VBScript *клиентской стороны*, которые

будут обрабатываться механизмом сценариев запрашивающего браузера. Поскольку Web Forms является главным образом технологией серверной стороны, тема написания сценариев клиентской стороны здесь не рассматривается.

На заметку! Обработка события для заданного веб-элемента управления с применением Visual Studio может делаться в такой же манере, как для элемента управления графического пользовательского интерфейса Windows. Понадобится только выбрать виджет на поверхности визуального конструктора и щелкнуть на значке с изображением молнии в окне Properties.

Свойство AutoPostBack

Полезно также упомянуть, что многие элементы управления Web Forms поддерживают свойство по имени `AutoPostBack` (в особенности элементы управления `CheckBox`, `RadioButton` и `TextBox`, а также любые элементы, производные от абстрактного класса `ListControl`). По умолчанию свойство `AutoPostBack` установлено в `false`, что отключает немедленную обратную отправку серверу (даже при наличии обработчика события в файле отделенного кода). В большинстве случаев именно это поведение и требуется, поскольку таким элементам пользовательского интерфейса, как флажки, обычно не нужна функциональность обратной отправки. Другими словами, выполнять обратную отправку немедленно после отметки или снятия отметки с флажка нежелательно, т.к. объект страницы может получить состояние виджета внутри более естественного обработчика события `Click` для `Button`.

Однако если необходимо заставить любой из этих виджетов немедленно выполнять обратную отправку обработчику события серверной стороны, то следует установить свойство `AutoPostBack` в `true`. Такой прием удобен, когда состояние одного виджета должно автоматически заполнять значение внутри другого виджета на той же самой странице. В целях иллюстрации предположим, что есть веб-страница, которая содержит элемент `TextBox` (по имени `txtAutoPostback`) и элемент `ListBox` (с именем `lstTextBoxData`). Ниже показана соответствующая разметка:

```
<form id="form1" runat="server">
  <asp:TextBox ID="txtAutoPostback" runat="server"></asp:TextBox>
  <br/>
  <asp:ListBox ID="lstTextBoxData" runat="server"></asp:ListBox>
</form>
```

Если обработано событие `TextChanged` элемента `TextBox`, тогда в обработчике события серверной стороны можно было бы попытаться заполнить элемент управления `ListBox` текущим значением `TextBox`:

```
partial class _Default : System.Web.UI.Page
{
  protected void txtAutoPostback_TextChanged(object sender, EventArgs e)
  {
    lstTextBoxData.Items.Add(txtAutoPostback.Text);
  }
}
```

После запуска приложения в том виде, как есть, при вводе текста в `TextBox` выяснится, что ничего не происходит. Более того, если ввести что-нибудь в `TextBox` и перейти к следующему элементу управления с помощью клавиши <Tab>, также ничего не происходит. Причина в том, что по умолчанию свойство `AutoPostBack` элемента `TextBox` установлено в `false`.

Тем не менее, если установить свойство `AutoPostBack` в `true`:

```
<asp:TextBox ID="txtAutoPostback" runat="server" AutoPostBack="true" ... >
</asp:TextBox>
```

то при покидании `TextBox` по нажатию <Tab> или <Enter> элемент управления `ListBox` автоматически заполнится текущим значением `TextBox`. Конечно, помимо необходимости заполнения элементов одного виджета на основе значения другого виджета изменять состояние свойства `AutoPostBack` виджета обычно не придется (даже в такой ситуации задачу можно решить внутри клиентского сценария, устраняя потребность во взаимодействии с сервером).

Базовые классы `Control` и `WebControl`

Базовый класс `System.Web.UI.Control` определяет разнообразные свойства, методы и события, которые предоставляют возможность взаимодействия с основными (обычно не имеющими отношения к графическому пользовательскому интерфейсу) аспектами веб-элемента управления. В табл. Г.1 документированы некоторые члены, представляющие интерес.

Таблица Г.1. Избранные члены `System.Web.UI.Control`

Член	Описание
`Controls`	Это свойство получает объект `ControlCollection`, который представляет дочерние элементы управления внутри текущего элемента
`DataBind()`	Этот метод привязывает источник данных к вызванному серверному элементу управления и всем его дочерним элементам
`EnableTheming`	Это свойство устанавливает, поддерживает ли элемент функциональность тем (стандартное значение равно `true`)
`HasControls()`	Этот метод определяет, содержит ли серверный элемент управления какие-то дочерние элементы
`ID`	Это свойство получает и устанавливает программный идентификатор серверного элемента управления
`Page`	Это свойство получает ссылку на экземпляр `Page`, который содержит данный серверный элемент управления
`Parent`	Это свойство получает ссылку на родительский элемент данного серверного элемента управления в иерархии элементов управления страницы
`SkinID`	Это свойство получает или устанавливает обложку для применения к элементу управления, позволяя определять внешний вид и поведение с использованием ресурсов серверной стороны
`Visible`	Это свойство получает или устанавливает значение, указывающее на то, будет ли серверный элемент управления визуализироваться как элемент пользовательского интерфейса на странице

Перечисление содержащихся элементов управления

Первым исследуемым аспектом класса `System.Web.UI.Control` будет тот факт, что все веб-элементы управления (включая `Page`) наследуют коллекцию специальных элементов управления (доступную через свойство `Controls`). Во многом подобно приложениям Windows Forms свойство `Controls` предоставляет доступ к строго типизованной коллекции объектов производных от `WebControl` типов.

Как и любая коллекция .NET, она позволяет динамически добавлять, вставлять и удалять элементы во время выполнения.

Наряду с тем, что формально возможно добавлять веб-элементы управления прямо к объекту производного от `Page` типа, проще (и надежнее) применять элемент управления `Panel`. Класс `Panel` представляет контейнер виджетов, которые могут быть или не быть видимыми конечному пользователю (в зависимости от значений их свойств `Visible` и `BorderStyle`).

Для примера создадим новый проект пустого веб-сайта по имени `DynamicCtrls` и добавим в него веб-форму. С помощью визуального конструктора страниц Visual Studio добавим элемент типа `Panel` (с именем `myPanel`), который содержит в себе виджеты `TextBox`, `Button` и `HyperLink` с произвольными именами (визуальный конструктор требует перетаскивания внутренних элементов на поверхность пользовательского интерфейса элемента управления `Panel`). За пределами `Panel` разместим виджет `Label` (по имени `lblControlInfo`), который будет содержать визуализированный вывод. Вот одна из возможных разметок HTML:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Dynamic Control Test</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
    <hr />
    <h1>Dynamic Controls</h1>
    <asp:Label ID="lblTextBoxText" runat="server"></asp:Label>
    <hr />
    </div>
    <!-- Элемент Panel содержит три элемента управления -->
    <asp:Panel ID="myPanel" runat="server" Width="200px"
        BorderColor="Black" BorderStyle="Solid" >
      <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox><br/>
      <asp:Button ID="Button1" runat="server" Text="Button"/><br/>
      <asp:HyperLink ID="HyperLink1" runat="server">HyperLink
      </asp:HyperLink>
    </asp:Panel>
    <br />
    <br />
    <asp:Label ID="lblControlInfo" runat="server"></asp:Label>
  </form>
</body>
</html>
```

При такой разметке поверхность визуального конструктора страницы будет выглядеть примерно так, как показано на рис. Г.1.

Предположим, что в обработчике события `Page_Load()` необходимо получить детальную информацию об элементах управления, содержащихся внутри `Panel`, и присвоить результат элементу управления `Label` (`lblControlInfo`). Взгляните на следующий код C#:

```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, System.EventArgs e)
  {
    ListControlsInPanel();
  }
```

```
  private void ListControlsInPanel()
  {
    var theInfo = "";
    theInfo = $"<b>Does the panel have controls? {myPanel.HasControls()}
                </b><br/>";
    // Получить все элементы управления в панели.
    foreach (Control c in myPanel.Controls)
    {
      if (!object.ReferenceEquals(c.GetType(),
           typeof(System.Web.UI.LiteralControl)))
      {
        theInfo += "**************************<br/>";
        theInfo += $"Control Name? {c} <br/>";      // Имя элемента управления
        theInfo += $"ID? {c.ID} <br>";   // Идентификатор элемента управления
        theInfo += $"Control Visible? {c.Visible} <br/>";
                  // Является ли элемент управления видимым
        theInfo += $"ViewState? {c.EnableViewState} <br/>";
                  // Включено ли состояние представления
      }
    }
    lblControlInfo.Text = theInfo;
  }
}
```

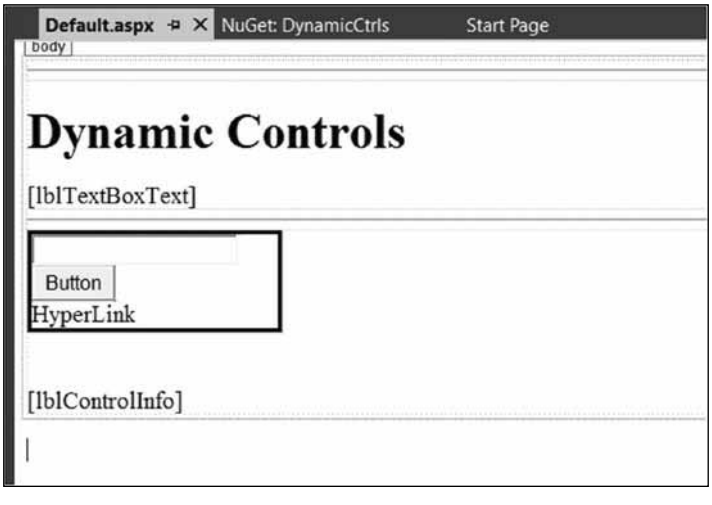

Рис. Г.1. Пользовательский интерфейс веб-страницы в проекте `DynamicCtrls`

Здесь осуществляется проход по всем объектам `WebControl`, которые обслуживает элемент управления `Panel`, с проверкой их принадлежности к типу `System.Web.UI.LiteralControl`; объекты указанного типа пропускаются. Класс `LiteralControl` используется для представления литеральных дескрипторов HTML и содержимого (`<br>`, текстовых литералов и т.д.). Если не предпринять такой проверки, тогда внутри `Panel` будет обнаружено намного больше элементов управления (для приведенного выше объявления `.aspx`). Предполагая, что элемент управления не является литеральным содержимым HTML, для него выводятся разнообразные статистические сведения. Результат показан на рис. Г.2.

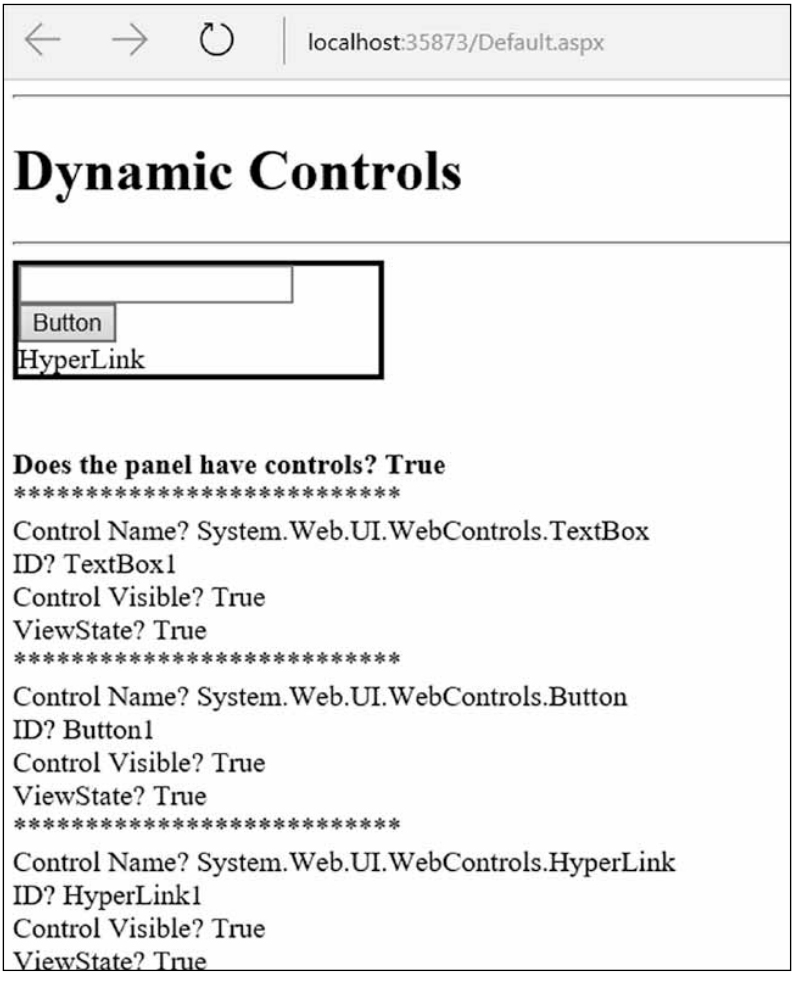

Рис. Г.2. Перечисление элементов управления во время выполнения

Динамическое добавление и удаление элементов управления

А что, если требуется изменять содержимое `Panel` во время выполнения? Давайте поместим на текущую страницу элемент `Button` (по имени `bntAddWidgets`), который будет динамически добавлять к `Panel` три новых элемента управления `TextBox`, и еще один элемент `Button` (с именем `btnRemovePanelItems`), который очистит `Panel` от всех вложенных элементов управления. Ниже представлены обработчики события `Click` для обеих кнопок:

```
protected void btnClearPanel_Click(object sender, System.EventArgs e)
{
  // Очистить содержимое панели, затем заново перечислить элементы.
  myPanel.Controls.Clear();
  ListControlsInPanel();
}
```

```
protected void btnAddWidgets_Click(object sender, System.EventArgs e)
{
  for (int i = 0; i < 3; i++)
  {
    // Присвоить идентификатор, чтобы позже можно было получить
    // текстовое значение с использованием входных данных формы.
    TextBox t = new TextBox {ID = $"newTextBox{i}"};
    myPanel.Controls.Add(t);
    ListControlsInPanel();
  }
}
```

Обратите внимание, что каждому элементу управления `TextBox` назначается уникальный идентификатор (`newTextBox0`, `newTextBox1` и т. д.). Запустив приложение, можно добавлять новые элементы к элементу управления `Panel` и полностью очищать содержимое `Panel`.

Взаимодействие с динамически созданными элементами управления

Получать значения из динамически сгенерированных элементов управления `TextBox` можно разными способами. Добавим к пользовательскому интерфейсу еще один элемент управления `Button` (по имени `btnGetTextData`) и последний элемент `Label` с именем `lblTextBoxData`, после чего обработаем событие `Click` для `Button`.

Существует несколько способов доступа к данным в динамически сгенерированных элементах `TextBox`. Один из подходов предусматривает проход в цикле по всем элементам, которые содержатся во входных данных формы HTML (доступных через `HttpRequest.Form`), и накопление текстовой информации в локальной строке `System.String`. После прохождения всей коллекции результирующую строку необходимо присвоить свойству `Text` нового элемента `Label`:

```
protected void btnGetTextData_Click(object sender, System.EventArgs e)
{
  string textBoxValues = "";
  for (int i = 0; i < Request.Form.Count; i++)
  {
    textBoxValues += $"<li>{ Request.Form[i]}</li><br/>";
  }
  lblTextBoxData.Text = textBoxValues;
}
```

Запустив приложение, вы увидите, что содержимое каждого текстового поля можно просматривать в виде довольно длинной (нечитабельной) строки, которая содержит *состояние представления* каждого элемента управления на странице. Роль состояния представления обсуждается в приложении Д.

Для получения более ясного вывода текстовые данные можно разнести по уникально именованным элементам (`newTextBox0`, `newTextBox1` и `newTextBox2`). Взгляните на следующую модификацию:

```
protected void btnGetTextData_Click(object sender, System.EventArgs e)
{
  // Получить текстовые поля по их именам.
  string lableData = $"<li>{Request.Form.Get("newTextBox0")}</li><br/>";
  lableData += $"<li>{Request.Form.Get("newTextBox1")}</li><br/>";
  lableData += $"<li>{Request.Form.Get("newTextBox2")}</li><br/>";
  lblTextBoxData.Text = lableData;
}
```

Применяя любой из подходов, вы заметите, что как только запрос обработан, текстовые поля исчезают. И снова причина кроется в не поддерживающей состояние природе HTML. Чтобы сохранять динамически созданные элементы управления `TextBox` между обратными отправками, нужно использовать приемы программирования с состоянием Web Forms (см. приложение Д).

Исходный код. Веб-сайт `DynamicCtrls` доступен в подкаталоге `Appendix_D`.

Функциональность базового класса `WebControl`

Как уже известно, класс `Control` обладает несколькими линиями поведения, которые не связаны с графическим пользовательским интерфейсом (коллекция элементов управления, поддержка автоматической обратной отправки и т.д.). С другой стороны, базовый класс `WebControl` предоставляет графический полиморфный интерфейс всем веб-виджетам (табл. Г.2).

Таблица Г.2. Избранные свойства базового класса `WebControl`

Свойство	Описание
`BackColor`	Получает или устанавливает цвет фона веб-элемента управления
`BorderColor`	Получает или устанавливает цвет контура веб-элемента управления
`BorderStyle`	Получает или устанавливает стиль контура веб-элемента управления
`BorderWidth`	Получает или устанавливает ширину контура веб-элемента управления
`Enabled`	Получает или устанавливает значение, которое указывает, доступен ли веб-элемент управления
`CssClass`	Позволяет назначить веб-элементу управления класс, определенный в каскадной таблице стилей
`Font`	Получает информацию о шрифте веб-элемента управления
`ForeColor`	Получает или устанавливает цвет переднего плана (обычно цвет текста) веб-элемента управления
`Height`, `Width`	Получает или устанавливает высоту и ширину веб-элемента управления
`TabIndex`	Получает или устанавливает индекс обхода по клавише <Tab> веб-элемента управления
`ToolTip`	Получает или устанавливает всплывающую подсказку веб-элемента управления, появляющуюся при наведении на него курсора мыши

Почти все свойства в табл. Г.2 самоочевидны, поэтому вместо того, чтобы демонстрировать их применение по отдельности, давайте посмотрим на множество элементов управления Web Forms в действии.

Основные категории элементов управления Web Forms

Библиотека элементов управления Web Forms может быть разбита на ряд обширных категорий, которые видны в панели инструментов Visual Studio (при условии, что в визуальном конструкторе открыта страница `.aspx`), как показано на рис. Г.3.

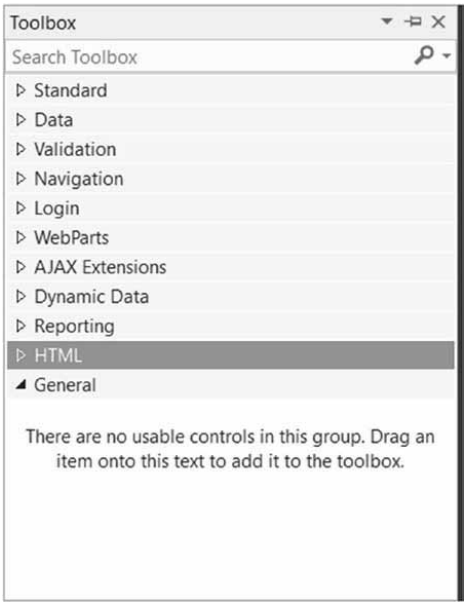

Рис. Г.3. Категории элементов управления Web Forms

В области Standard (Стандартные) панели инструментов находятся самые часто используемые элементы управления, включая `Button`, `Label`, `TextBox` и `ListBox`. Помимо этих простых элементов пользовательского интерфейса в области Standard также доступны и более экзотические веб-элементы управления, такие как `Calendar`, `Wizard` и `AdRotator` (рис. Г.4).

Область Data (Данные) содержит набор элементов управления, применяемых для операций привязки данных, в том числе элемент управления Web Forms под названием `Chart`, позволяющий визуализировать график (круговую диаграмму, гистограмму и т.д.), который обычно является результатом операции привязки данных (рис. Г.5).

Элементы управления проверкой достоверности Web Forms (находящиеся в области Validation (Проверка достоверности) панели инструментов) очень интересны тем, что их можно конфигурировать для выпуска блоков кода JavaScript клиентской стороны, которые будут проверять допустимость данных в полях ввода. Если происходит ошибка проверки достоверности, то пользователь увидит сообщение об ошибке и не сможет выполнить обратную отправку на сервер до тех пор, пока не устранит ошибку.

В области Navigation (Навигация) панели инструментов располагается небольшой набор элементов управления (`Menu`, `SiteMapPath` и `TreeView`), которые обычно работают в сочетании с файлом `.sitemap`. Как уже кратко упоминалось ранее в приложении, элементы управления навигацией позволяют описывать структуру многостраничного сайта, используя дескрипторы XML.

По-настоящему экзотическим набором элементов управления Web Forms можно назвать элементы в области Login (Вход), показанные на рис. Г.6.

Элементы управления из области Login могут радикально упростить внедрение в веб-приложения базовых средств безопасности (восстановление пароля, экраны входа и т.п.). В действительности они настолько мощные, что даже будут динамически создавать выделенную базу данных для хранения регистрационных данных (в папке `App_Data` веб-сайта), если специфическая база данных для целей безопасности отсутствует.

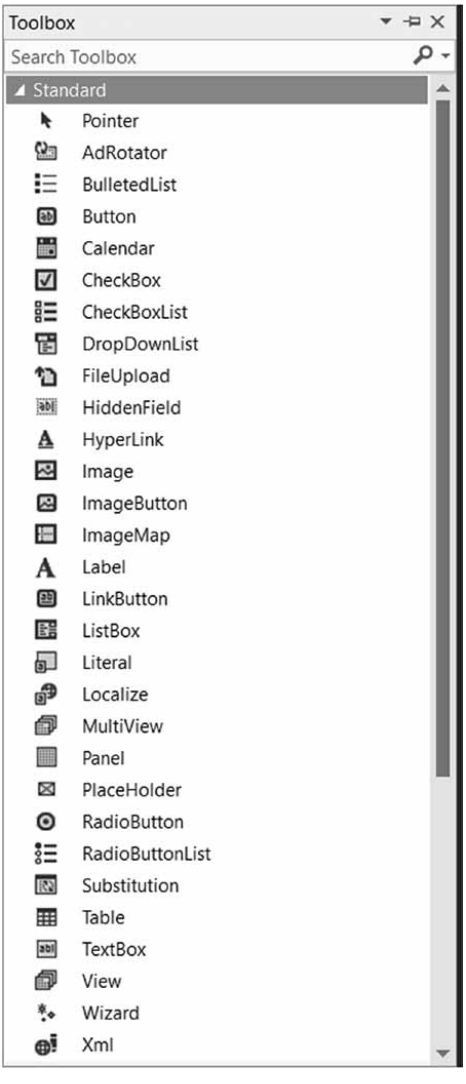

Рис. Г.4. Стандартные элементы управления Web Forms

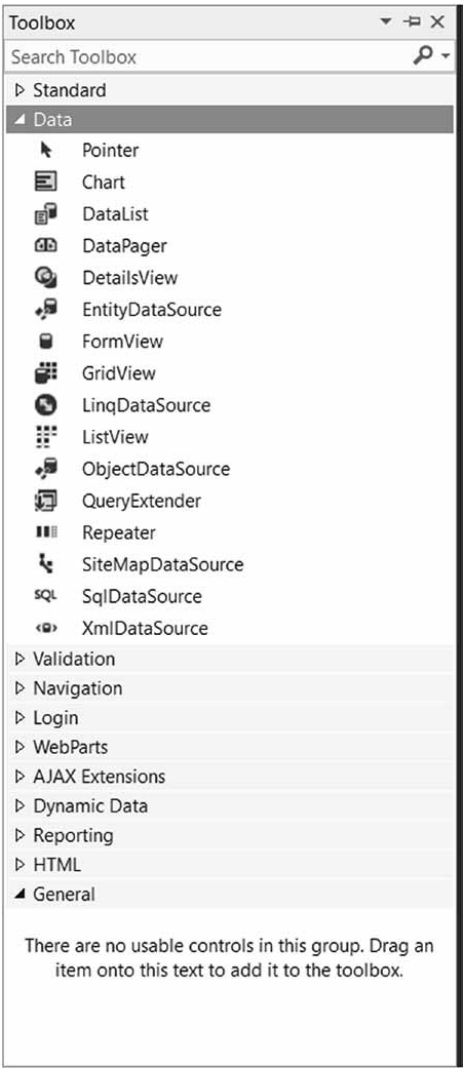

Рис. Г.5. Элементы управления Web Forms, ориентированные на данные

На заметку! Остальные категории веб-элементов управления, отображаемые в панели инструментов Visual Studio (WebParts (Веб-части), AJAX Extensions (Расширения AJAX) и Dynamic Data (Динамические данные)), предназначены для решения более специализированных задач программирования и здесь не рассматриваются.

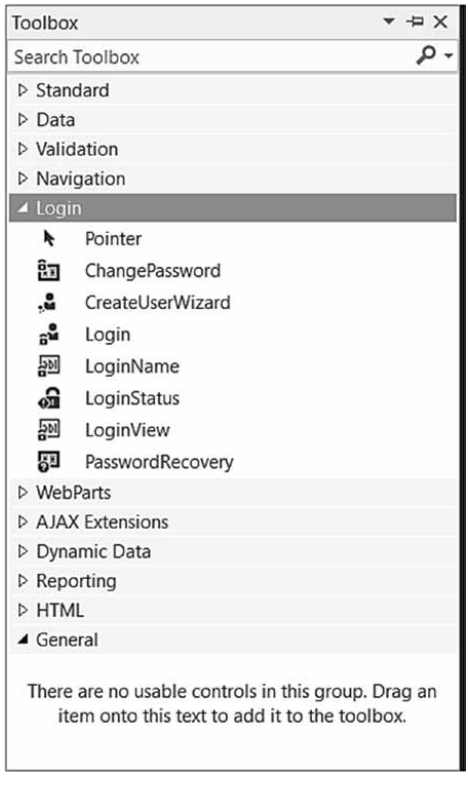

Рис. Г.6. Элементы управления Web Forms, связанные с безопасностью

Несколько слов о пространстве имен `System.Web.UI.HtmlControls`

По правде говоря, с инфраструктурой Web Forms поставляются два отдельных инструментальных набора веб-элементов управления. В дополнение к элементам управления Web Forms (внутри пространства имен `System.Web.UI.WebControls`) библиотеки базовых классов также предоставляют элементы управления HTML в пространстве имен `System.Web.UI.HtmlControls`.

Элементы управления HTML — это коллекция типов, которые позволяют применять традиционные элементы управления HTML на странице Web Forms. Однако в отличие от простых дескрипторов HTML элементы управления HTML являются объектно-ориентированными сущностями, которые могут быть сконфигурированы для запуска на сервере и соответственно поддерживать обработку событий серверной стороны. В отличие от элементов управления Web Forms элементы управления HTML довольно просты по своей природе и предлагают лишь небольшую функциональность сверх той, что обеспечивают стандартные дескрипторы HTML (`HtmlButton`, `HtmlInputControl`, `HtmlTable` и т.д.).

Элементы управления HTML могут быть полезны, если команда четко разделена на тех, кто занимается построением пользовательских интерфейсов HTML, и разработчиков .NET. Специалисты по HTML могут использовать свой веб-редактор, имея дело со знакомыми дескрипторами разметки, и передавать готовые файлы HTML команде разработки. Затем разработчики .NET могут конфигурировать элементы управления HTML

для выполнения в качестве серверных элементов управления (посредством контекстного меню, открываемого по щелчку правой кнопкой мыши на элементе управления HTML в Visual Studio). Такой подход позволит разработчикам обрабатывать события серверной стороны и программно взаимодействовать с виджетами HTML.

Элементы управления HTML предоставляют открытый интерфейс, который имитирует стандартные атрибуты HTML. Например, для получения информации из области ввода применяется свойство `Value` вместо свойства `Text`, принятого у веб-элементов управления. Поскольку элементы управления HTML не настолько многофункциональны, как элементы управления Web Forms, больше они упоминаться не будут.

Документация по веб-элементам управления

В оставшихся материалах у вас еще будет шанс поработать с элементами управления Web Forms; тем не менее, вы определенно должны выделить время на ознакомление с описанием пространства имен `System.Web.UI.WebControls` в документации .NET Framework 4.7 SDK. Там вы найдете объяснения и примеры кода для каждого члена указанного пространства имен (рис. Г.7).

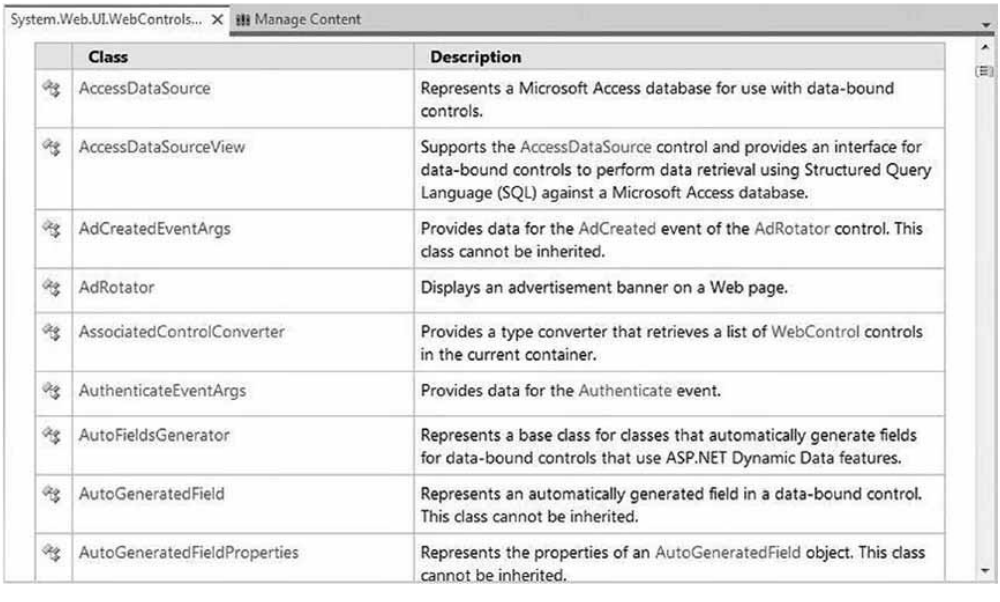

Рис. Г.7. Все элементы управления Web Forms описаны в документации .NET Framework 4.7 SDK

Построение веб-сайта Web Forms для работы с автомобилями

Учитывая, что многие "простые" элементы управления выглядят и ведут себя близко к своим аналогам из графического пользовательского интерфейса Windows, детали базовых виджетов (`Button`, `Label`, `TextBox` и т.д.) подробно рассматриваться не будут. Взамен давайте построим веб-сайт, который будет иллюстрировать работу с несколькими более экзотическими элементами управления, а также с моделью мастер-страниц Web Forms и аспектами механизма привязки данных.

В частности, в примере будут продемонстрированы следующие приемы:

- работа с мастер-страницами;
- работа с навигацией посредством карты сайта;
- работа с элементом управления `GridView`;
- работа с элементом управления `Wizard`.

Для начала создадим проект пустого веб-сайта по имени `AppNetCarsSite`. Новый полный проект веб-сайта ASP.NET пока не создается, т.к. в нем предусмотрено несколько начальных файлов, которые еще не рассматривались. В текущем проекте все необходимое будет добавляться вручную.

Работа с мастер-страницами Web Forms

Многие веб-сайты обеспечивают согласованный внешний вид и поведение, которые распространяются на целый набор страниц (общая система навигации с помощью меню, общее содержимое заголовков и нижних колонтитулов, логотип компании и т.п.). Мастер-страница является всего лишь страницей Web Forms, которая имеет файловое расширение `.master`. Сами по себе мастер-страницы не являются просматриваемыми в браузере клиентской стороны (на самом деле исполняющая среда ASP.NET не обслуживает веб-содержимое такого рода). Взамен мастер-страницы определяют общую компоновку пользовательского интерфейса, разделяемую всеми страницами (либо их подмножеством) на сайте.

Кроме того, страница `.master` будет определять разнообразные области-заполнители содержимого, которые образуют раздел пользовательского интерфейса, куда могут подключаться другие файлы `.aspx`. Вы увидите, что файлы `.aspx`, которые включают свое содержимое в мастер-страницу, выглядят и ведут себя немного иначе, чем те файлы `.aspx`, с которыми вы сталкивались до сих пор. В частности, такая разновидность файла `.aspx` называется *страницей содержимого*. Страницы содержимого представляют собой файлы `.aspx`, в которых не определен HTML-элемент `<form>` (это относится к работе мастер-страницы).

Однако с точки зрения конечного пользователя запрос производится к заданному файлу `.aspx`. На веб-сервере соответствующий файл `.master` и любые связанные страницы содержимого `.aspx` смешиваются вместе в единственное объявление страницы HTML.

Для демонстрации использования мастер-страниц и страниц содержимого вставим в разрабатываемый веб-сайт новую мастер-страницу посредством пункта меню Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент); результирующее диалоговое окно приведено на рис. Г.8.

Начальная разметка файла `MasterPage.master` выглядит следующим образом:

```
<%@ Master Language="C#" AutoEventWireup="true"
    CodeFile="MasterPage.master.cs" Inherits="MasterPage" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
  <asp:ContentPlaceHolder id="head" runat="server">
  </asp:ContentPlaceHolder>
</head>
<body>
  <form id="form1" runat="server">
    <div>
```

```
      <asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server">
      </asp:ContentPlaceHolder>
    </div>
  </form>
</body>
</html>
```

Первый интересный момент связан с новой директивой `<%@ Master %>`, которая по большей части поддерживает те же самые атрибуты, что и директива `<%@ Page %>`, описанная в приложении В. Подобно типам `Page` мастер-страница является производной от специфичного базового класса, в данном случае `MasterPage`. Открыв соответствующий файл кода, вы обнаружите такое определение класса:

```
public partial class MasterPage : System.Web.UI.MasterPage
{
  protected void Page_Load(object sender, EventArgs e)
  {
  }
}
```

Другой интересный момент в разметке мастер-страницы касается определения `<asp:ContentPlaceHolder>`. В эту область мастер-страницы могут подключаться виджеты пользовательского интерфейса из связанного файла содержимого `.aspx`, а не содержимое, которое определено самой мастер-страницей.

Если вы намерены подключить файл `.aspx` к указанному разделу, то область внутри дескрипторов `<asp:ContentPlaceHolder>` и `</asp:ContentPlaceHolder>` обычно оставляется пустой. Тем не менее, этот раздел можно заполнить разнообразными элементами управления Web Forms, которые функционируют как стандартный пользовательский интерфейс для применения в случае, если заданный файл `.aspx` на сайте не предоставит специфическое содержимое. В рассматриваемом примере предположим, что каждая страница `.aspx` сайта действительно будет предоставлять специальное содержимое, и потому элементы `<asp:ContentPlaceHolder>` будут пустыми.

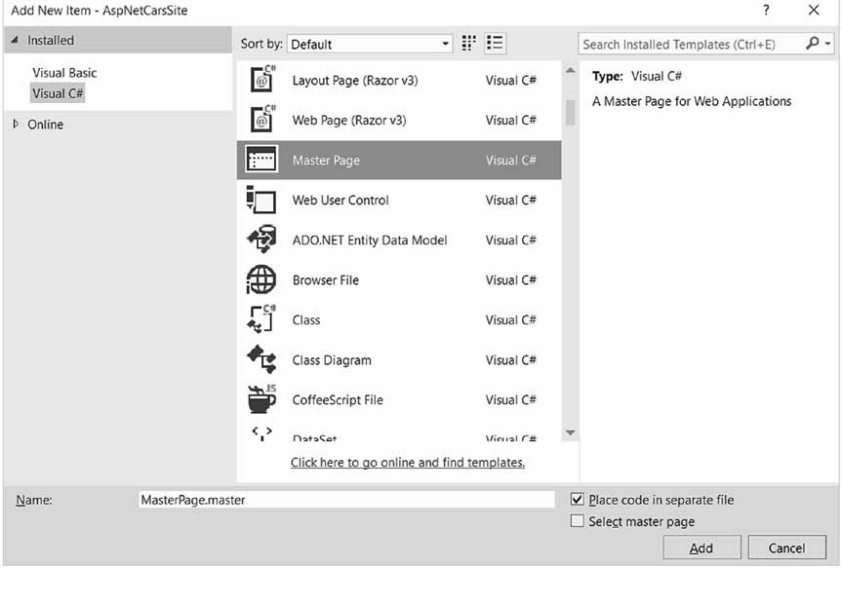

Рис. Г.8. Вставка нового файла `.master`

На заметку! Внутри страницы `.master` можно определять произвольное количество заполнителей содержимого. Кроме того, страница `.master` может иметь вложенные страницы `.master`.

Общий пользовательский интерфейс файла `.master` можно строить с помощью тех же визуальных конструкторов Visual Studio, которые используются для создания файлов `.aspx`. Добавим к сайту описательный элемент `Label` (служащий общим приветственным сообщением), элемент управления `AdRotator` (который отображает случайно выбранное одно из двух изображений) и элемент управления `TreeView` (позволяющий пользователю выполнять навигацию на другие области сайта). Вот как выглядит возможная разметка мастер-страницы:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title> </title>
  <asp:ContentPlaceHolder id="head" runat="server">
  </asp:ContentPlaceHolder>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <hr />
      <asp:Label ID="Label1" runat="server" Font-Size="XX-Large"
        Text="Welcome to the ASP.NET Cars Super Site!"></asp:Label>
      <asp:AdRotator ID="myAdRotator" runat="server"/>
       <br />
      <br />
      <asp:TreeView ID="navigationTree" runat="server">
      </asp:TreeView>
      <hr />
    </div>
    <div>
      <asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server">
      </asp:ContentPlaceHolder>
    </div>
  </form>
</body>
</html>
```

На рис. Г.9 показано представление текущей мастер-страницы во время проектирования (обратите внимание, что область отображения элемента управления `AdRotator` на данный момент пуста).

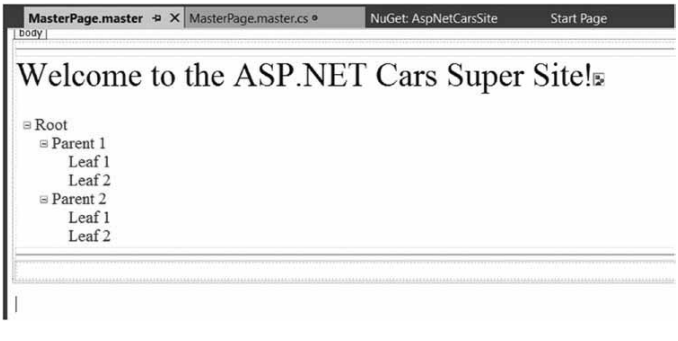

Рис. Г.9. Разделяемый пользовательский интерфейс в файле `.master`

Можете улучшить внешний вид элемента управления `TreeView` с применением встроенного редактора элемента управления и выбора ссылки Auto Format (Автоформат). Кроме того, можете настроить отображение остальных элементов управления в окне Properties. После получения удовлетворительных результатов переходите к следующему разделу.

Конфигурирование логики навигации по сайту с помощью элемента управления `TreeView`

Инфраструктура Web Forms поставляется с несколькими веб-элементами управления, которые позволяют поддерживать навигацию по сайту: `SiteMapPath`, `TreeView` и `Menu`. Как и можно было предположить, упомянутые веб-виджеты можно конфигурировать многими способами. Например, каждый из этих элементов управления может динамически генерировать свои узлы через внешний файл XML (или файл `.sitemap`, основанный на XML), программно в коде или посредством разметки с использованием визуальных конструкторов среды Visual Studio.

Создаваемая система навигации будет динамически наполняться с применением файла `.sitemap`. Преимущество такого подхода в том, что можно определить общую структуру веб-сайта во внешнем файле и затем привязывать его к элементу управления `TreeView` (или `Menu`) на лету. Таким образом, если навигационная структура веб-сайта изменится, то достаточно будет просто модифицировать файл `.sitemap` и перегрузить страницу. Вставим в проект новый файл `Web.sitemap`, выбрав пункт меню Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент), в результате чего откроется диалоговое окно, показанное на рис. Г.10.

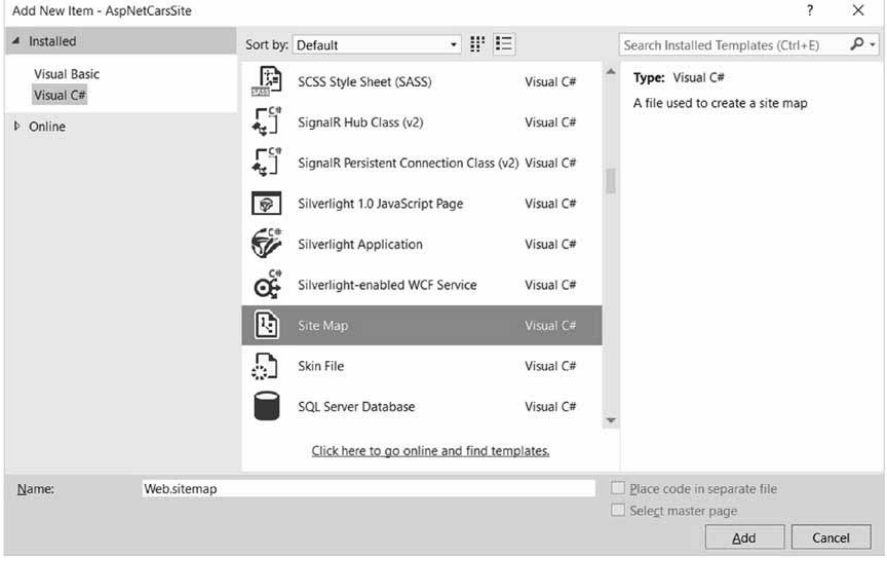

Рис. Г.10. Вставка нового файла `Web.sitemap`

В начальном содержимом файла `Web.sitemap` определен элемент самого верхнего уровня с двумя подузлами:

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode url="" title="" description="">
```

```
    <siteMapNode url="" title="" description="" />
    <siteMapNode url="" title="" description="" />
  </siteMapNode>
</siteMap>
```

После привязки такой структуры к элементу управления `Menu` должен отобразиться элемент верхнего уровня с двумя подэлементами. Следовательно, для создания подэлементов понадобится просто определить новые элементы `<siteMapNode>` внутри существующего элемента `<siteMapNode>`. В любом случае цель связана с определением общей структуры веб-сайта в файле `Web.sitemap` с использованием разнообразных элементов `<siteMapNode>`. Каждый такой элемент может определять заголовок и атрибут URL. Атрибут URL представляет файл `.aspx`, к которому будет выполнен переход, когда пользователь щелкнет на заданном элементе (или на узле `TreeView`). Создаваемая карта сайта будет содержать три узла (расположенные ниже узла верхнего уровня):

- Home (Домой): `Default.aspx`
- Build a Car (Собрать автомобиль): `BuildCar.aspx`
- View Inventory (Просмотреть склад): `Inventory.aspx`

Вскоре мы добавим в проект три указанных новых страницы Web Forms. А пока нужно просто сконфигурировать файл карты сайта.

Система навигации имеет единственный элемент верхнего уровня Welcome (Добро пожаловать) с тремя подэлементами. Модифицируем файл `Web.sitemap` следующим образом (каждое значение `url` должно быть уникальным, иначе возникнет ошибка времени выполнения):

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode url="" title="Welcome!" description="">
    <siteMapNode url="~/Default.aspx" title="Home"
      description="The Home Page" />
    <siteMapNode url="~/BuildCar.aspx" title="Build a car"
      description="Create your dream car" />
    <siteMapNode url="~/Inventory.aspx" title="View Inventory"
      description="See what is in stock" />
    </siteMapNode>
</siteMap>
```

На заметку! Префикс `~/` перед каждой страницей в атрибуте `url` обозначает корень веб-сайта.

Вопреки тому, что вы могли подумать, файл `Web.sitemap` не ассоциируется непосредственно с элементом управления `Menu` или `TreeView` с применением какого-то свойства. Взамен файл `.master` или `.aspx`, содержащий виджет пользовательского интерфейса, который отобразит файл `Web.sitemap`, должен содержать компонент `SiteMapDataSource`. Этот компонент будет автоматически загружать файл `Web.sitemap` в свою объектную модель, когда страница запрашивается. Затем объекты `Menu` и `TreeView` установят свои свойства `DataSourceID` так, чтобы указывать на данный экземпляр `SiteMapDataSource`.

Чтобы добавить новый компонент `SiteMapDataSource` в файл `.master` и автоматически установить свойство `DataSourceID`, можно задействовать визуальный конструктор Visual Studio. Активизируем встроенный редактор элемента управления `TreeView` (щелкнув на небольшой стрелке в правом верхнем углу `TreeView`), раскроем список Choose Data Source (Выберите источник данных) и выберем пункт <New Data Source> (<Новый источник данных>), как показано на рис. Г.11.

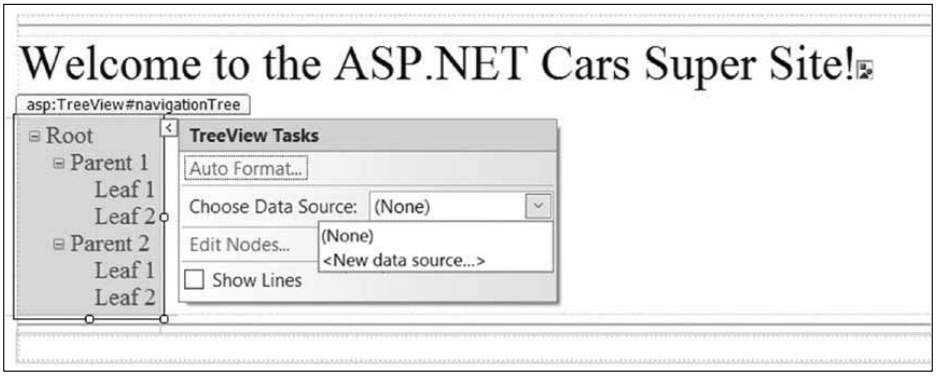

Рис. Г.11. Добавление нового компонента `SiteMapDataSource`

В открывшемся диалоговом окне выберем значок SiteMap (Карта сайта). В результате будет установлено свойство `DataSourceID` элемента управления `Menu` или `TreeView`, а к странице добавится компонент `SiteMapDataSource`. Вот и все, что потребовалось сделать, чтобы сконфигурировать элемент управления `TreeView` для перехода на дополнительные страницы внутри сайта. Если необходимо выполнить дополнительную обработку, когда пользователь выбивает пункт меню, то придется обработать событие `SelectedNodeChanged` элемента управления `TreeView`. В рассматриваемом примере в этом нет нужды, но имейте в виду, что выяснить, какой пункт меню был выбран, можно с использованием входных аргументов события.

Установка навигационных цепочек с помощью элемента `SiteMapPath`

Прежде чем переходить к элементу управления `AddRotator`, добавим в файл `.master` элемент `SiteMapPath` (расположенный на вкладке Navigation (Навигация) панели инструментов). Такой виджет будет автоматически настраивать свое содержимое на основе текущего выбора в системе меню. В итоге конечный пользователь получает полезную визуальную подсказку (формально данный аспект пользовательского интерфейса называется *навигационными цепочками* или *"хлебными крошками"*). По завершении можно заметить, что при выборе пункта меню Welcome⇨Build a Car (Добро пожаловать⇨Укомплектовать автомобиль) виджет `SiteMapPath` автоматически обновится надлежащим образом.

Конфигурирование элемента управления `AdRotator`

Роль виджета `AdRotator` из Web Forms заключается в показе случайно выбранного изображения в определенной позиции внутри браузера. В настоящий момент виджет `AdRotator` отображает пустой заполнитель. Этот элемент управления не может выполнять свою работу до тех пор, пока в его свойстве `AdvertisementFile` не будет указан исходный файл, описывающий каждое изображение. В рассматриваемом примере источником данных будет простой файл XML по имени `Ads.xml`.

Чтобы добавить файл XML к веб-сайту, выберем пункт меню Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент) и укажем вариант XML file (XML-файл). Назовем файл `Ads.xml` и предусмотрим в нем уникальный элемент `<Ad>` для каждого изображения, которое планируется отображать. Как минимум в элементе `<Ad>` должно быть указано графическое изображение для показа (`ImageUrl`), URL для навигации, если изображение выбрано (`TargetUrl`), текст, появляющийся при наведении курсора мыши (`AlternateText`), и вес показа (`Impressions`):

```
<Advertisements>
  <Ad>
    <ImageUrl>SlugBug.jpg</ImageUrl>
    <TargetUrl>http://www.Cars.com</TargetUrl>
    <AlternateText>Your new Car?</AlternateText>
    <Impressions>80</Impressions>
  </Ad>
  <Ad>
    <ImageUrl>car.gif</ImageUrl>
    <TargetUrl>http://www.CarSuperSite.com</TargetUrl>
    <AlternateText>Like this Car?</AlternateText>
    <Impressions>80</Impressions>
  </Ad>
</Advertisements>
```

Здесь указаны два файла с изображениями (`SlugBug.jpg` и `car.gif`), наличие которых придется обеспечить в корневом каталоге веб-сайта (они включены в состав загружаемого кода примеров для книги). Чтобы добавить их в текущий проект, выберем пункт меню Web Site⇨Add Existing Item (Веб-сайт⇨Добавить существующий элемент). Затем файл XML можно ассоциировать с элементом управления `AdRotator` посредством свойства `AdvertisementFile` (в окне Properties):

```
<asp:AdRotator ID="myAdRotator" runat="server"
               AdvertisementFile="~/Ads.xml"/>
```

Позже запустив приложение и выполнив обратную отправку страницы, можно увидеть случайно выбранный из двух файлов изображения.

Определение стандартной страницы содержимого

Располагая готовой мастер-страницей, можно приступить к проектированию индивидуальных страниц `.aspx`, которые будут определять содержимое пользовательского интерфейса для помещения внутрь дескриптора `<asp:ContentPlaceHolder>` мастер-страницы. Файлы `.aspx`, которые объединяются с мастер-страницей, называются *страницами содержимого* и имеют несколько ключевых отличий от нормальной автономной страницы Web Forms.

Выражаясь кратко, в файле `.master` определяется раздел `<form>` финальной страницы HTML. Следовательно, существующая область `<form>` внутри файла `.aspx` должна быть заменена областью `<asp:Content>`. В то время как можно было бы модифицировать разметку в начальном файле `.aspx` вручную, лучше вставить в проект новую страницу содержимого. Удалим имеющийся файл `Default.aspx`, щелкнем правой кнопкой мыши в любом месте на поверхности визуального конструктора файла `.master` и выберем в контекстном меню пункт Add Content Page (Добавить страницу содержимого), как показано на рис. Г.12.

В результате будет сгенерирован новый файл `.aspx` с приведенной ниже начальной разметкой:

```
<%@ Page Language="C#" MasterPageFile="~/MasterPage.master"
        AutoEventWireup="true" CodeFile="Default.aspx.cs"
        Inherits="_Default" Title="" %>

<asp:Content ID="Content1"
  ContentPlaceHolderID="head" Runat="Server">
</asp:Content>
<asp:Content ID="Content2"
  ContentPlaceHolderID="ContentPlaceHolder1" Runat="Server">
</asp:Content>
```

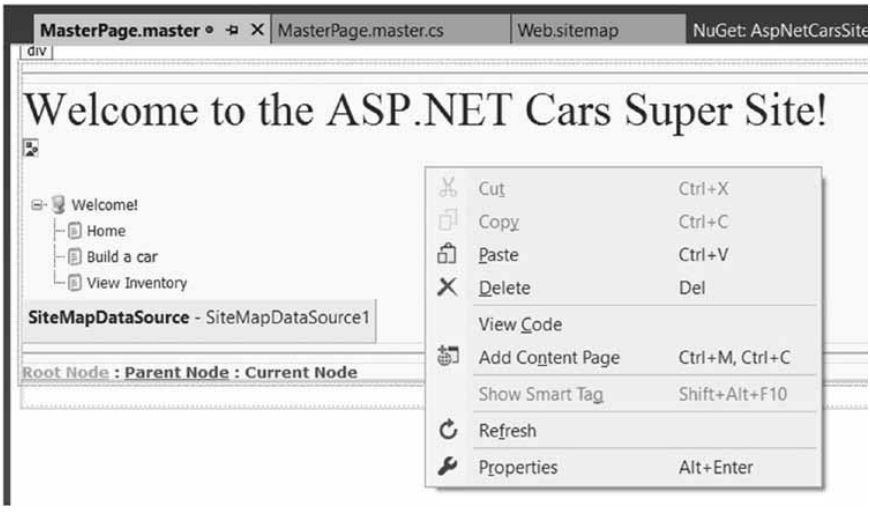

Рис. Г.12. Добавление к мастер-странице новой страницы содержимого

Прежде всего, обратите внимание на то, что директива `<%@ Page %>` дополнена новым атрибутом `MasterPageFile`, который указывает на файл `.master`. Кроме того, вместо элемента `<form>` имеется область `<asp:Content>` (пока пустая), в которой для атрибута `ContentPlaceHolderID` установлено значение, идентичное компоненту `<asp:ContentPlaceHolder>` в файле мастер-страницы.

С учетом упомянутых сопоставлений странице содержимого известно, куда подключается ее содержимое, тогда как содержимое мастер-страницы отображается в стиле только для чтения на странице содержимого. Строить сложный пользовательский интерфейс для области содержимого `Default.aspx` нет нужды. В данном примере просто добавим некоторый литеральный текст с базовыми инструкциями относительно сайта, как показано на рис. Г.13 (в правом верхнем углу страницы содержимого в визуальном конструкторе есть ссылка для переключения на связанную мастер-страницу).

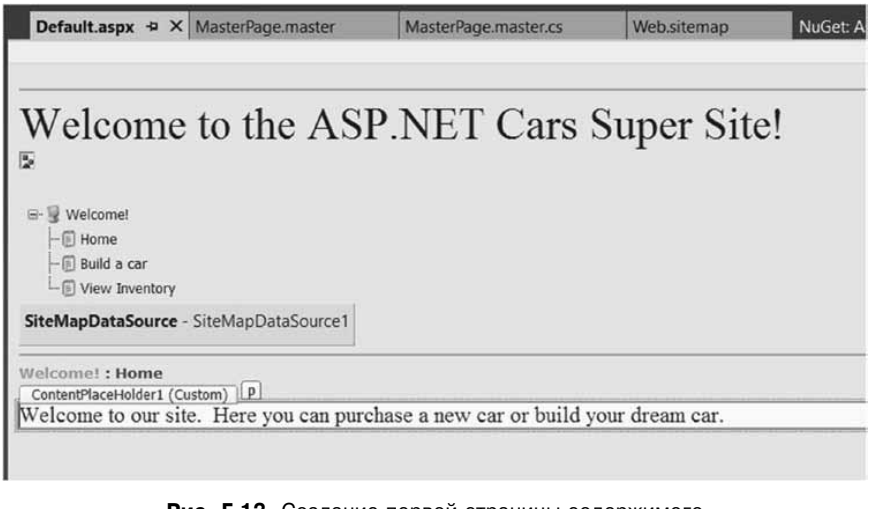

Рис. Г.13. Создание первой страницы содержимого

Теперь после запуска проекта обнаруживается, что содержимое пользовательского интерфейса из файлов `.master` и `Default.aspx` объединено в единый поток разметки HTML. Как видно на рис. Г.14, браузер (или конечный пользователь) даже не осведомлен о существовании мастер-страницы (обратите внимание, что браузер просто отображает разметку HTML из `Default.aspx`). Вдобавок, если обновить страницу (нажатием <F5>), то элемент управления `AdRotator` отобразит случайно выбранное одно из двух изображений.

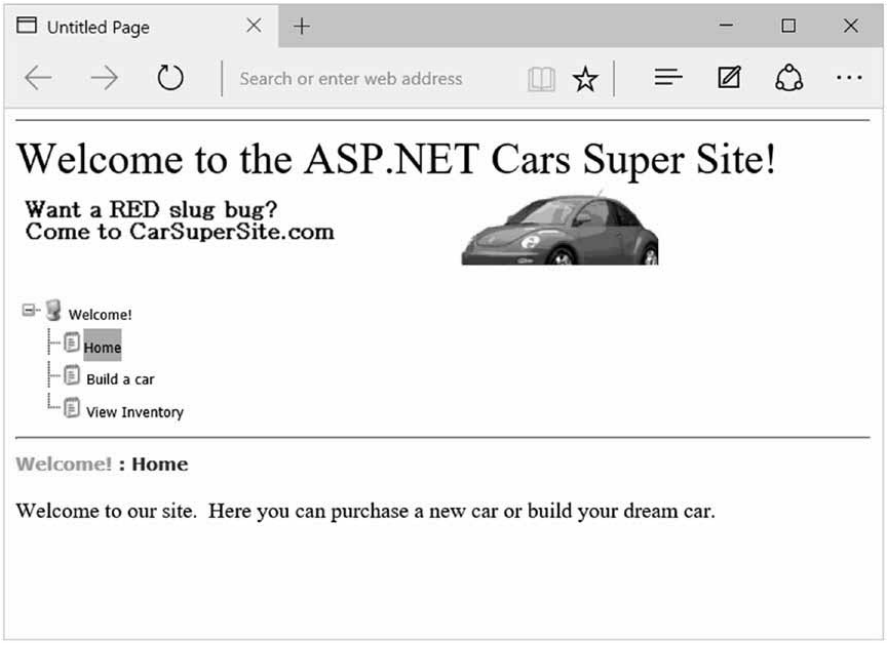

Рис. Г.14. Во время выполнения мастер-страницы и страницы содержимого визуализируются в единую форму

Проектирование страницы содержимого `Inventory.aspx`

Чтобы вставить в текущий проект страницу содержимого `Inventory.aspx`, откроем файл `.master` в IDE-среде, выберем пункт меню Website⇨Add Content Page (Вебсайт⇨Добавить страницу содержимого) и в окне Solution Explorer переименуем файл в `Inventory.aspx`. Страница содержимого `Inventory.aspx` будет отображать записи таблицы `Inventory` из базы данных `AutoLot` внутри элемента управления `GridView`. Однако в отличие от предыдущего приложения этот элемент управления `GridView` будет сконфигурирован для взаимодействия с базой данных `AutoLot` с применением встроенной поддержки привязки данных и обновленной сборки `AutoLotDAL.dll` из главы 22.

Хотя элемент управления `GridView` из Web Forms обладает способностью представлять данные строки соединения и SQL-операторы `Select`, `Insert`, `Update` и `Delete` (или в качестве альтернативы хранимые процедуры) *в разметке*, предпочтительнее использовать уровень доступа к данным (Data Access Layer — DAL). Такой подход способствует разделению обязанностей, а также изолирует хранилище данных от кода пользовательского интерфейса.

С помощью нескольких простых атрибутов и минимального кода (с учетом наличия DAL) элемент управления GridView можно сконфигурировать для автоматической выборки, обновления и удаления записей лежащего в основе хранилища данных. Это значительно сокращает объем стереотипного кода и включается через свойства SelectMethod, DeleteMethod и UpdateMethod (а также дополнительные средства для списковых элементов управления), введенные в ASP.NET 4.5.

Добавление сборки AutoLotDAL и инфраструктуры Entity Framework в проект AspNetCarsSite

Скопируем сборку AutoLotDAL из подкаталога Chapter_22 (либо из подкаталога Appendix_D в загружаемом коде примеров). Добавим ссылку на сборку AutoLotDAL, для чего щелкнем правой кнопкой мыши на узле References (Ссылки) проекта AspNetCarsSite, выберем в контекстном меню пункт Add Reference (Добавить ссылку), в открывшемся диалоговом окне Add Reference (Добавление ссылки) щелкнем на кнопке Browse (Обзор) и укажем файл AutoLotDAL.dll.

Затем добавим в веб-проект инфраструктуру Entity Framework, щелкнув правой кнопкой мыши на имени проекта и выбрав в контекстном меню пункт Manage NuGet Packages (Управление пакетами NuGet). Также понадобится добавить в файл Web.config строку подключения. Изменение будет выглядеть следующим образом (строка подключения зависит от установленной версии SQL Server Express):

```
  <connectionStrings>
    <add name="AutoLotConnection"
      connectionString="data source=.\SQLEXPRESS2014;initial
catalog=AutoLot;integrated security=True;MultipleActiveResultSets=True;
App=EntityFramework"
      providerName="System.Data.SqlClient"/>
  </connectionStrings>
```

Наполнение данными элемента управления GridView

Для демонстрации работы с элементом управления GridView в декларативной манере модифицируем страницу содержимого Inventory.aspx с определением GridView. В табл. Г.3 кратко описаны атрибуты, которые будут добавлены в последующих разделах.

Таблица Г.3. Избранные атрибуты списковых элементов управления

Атрибут	Описание
DataKeyNames	Указывает первичный ключ таблицы
ItemType	Включает строгую типизацию для списковых элементов управления
SelectMethod	Указывает метод в отделенном коде, который будет применяться для наполнения спискового элемента управления. Он вызывается при каждой визуализации элемента управления
DeleteMethod	Указывает метод, который будет использоваться для удаления записи из источника данных для таблицы
UpdateMethod	Указывает метод, который будет применяться для обновления записи, когда отредактированные данные отправляются серверу

Начнем с использования атрибутов ItemType и SelectMethod (как делалось в приложении В). Модифицируем объявление GridView (во втором элементе <asp:Content>):

```
<asp:GridView ID="GridView2" runat="server" CellPadding="4"
  AutoGenerateColumns="False"
  ItemType="AutoLotDAL.Models.Inventory" SelectMethod="GetData"
  EmptyDataText="There are no data records to display." ForeColor="#333333"
  GridLines="None">
```

Далее добавим следующие записи в разделе <Columns> (пока не обращая внимания на атрибуты `SortExpression`):

```
<Columns>
 <asp:BoundField DataField="CarID" HeaderText="CarID" ReadOnly="True"
      SortExpression="CarID" />
 <asp:BoundField DataField="Make" HeaderText="Make" SortExpression="Make" />
 <asp:BoundField DataField="Color" HeaderText="Color" SortExpression="Color" />
 <asp:BoundField DataField="PetName" HeaderText="PetName" SortExpression="PetName"/>
</Columns>
```

Удостоверимся в наличии закрывающего дескриптора `GridView` после закрывающего дескриптора `Columns`:

```
</asp:GridView>
```

Откроем файл `Inventory.aspx.cs` и добавим метод `GetData()`, который не принимает параметры и возвращает `IEnumerable<Inventory>`. Из-за того, что сборка DAL уже создана, код будет тривиален. Добавим операторы `using` для пространств имен `AutoLotDAL.Models` и `AutoLotDAL.Repos`, после чего реализуем метод `GetData()`:

```
public IEnumerable<Inventory> GetData() => new InventoryRepo().GetAll();
```

Теперь приложение можно запустить. Щелкнем на пункте меню View Inventory (Просмотреть склад) и просмотрим данные (рис. Г.15). (Внешний вид элемента управления `GridView` был изменен с помощью встроенного редактора.)

Рис. Г.15. Отображение данных из таблицы `Inventory`

Включение редактирования на месте

Следующей задачей будет включение для элемента управления GridView поддержки действий на месте. Мы начнем применять атрибуты DataKeyNames, DeleteMethod и UpdateMethod. Изменим объявление GridView, как показано ниже:

```
<asp:GridView ID="GridView2" runat="server" CellPadding="4"
  AutoGenerateColumns="False"
  DataKeyNames="CarID, Timestamp"
  ItemType="AutoLotDAL.Models.Inventory"
  SelectMethod="GetData"
  DeleteMethod="Delete"
  UpdateMethod="Update"
  EmptyDataText="There are no data records to display."
  ForeColor="#333333" GridLines="None">
```

Поле CarId является первичным ключом, так что ему имеет смысл находиться в атрибуте DataKeyNames. Поле типа Timestamp добавляется в DataKeyNames как ключ данных, поэтому оно будет передаваться методам Update() и Delete(). Добавим в раздел <Columns> следующую запись CommandField. В итоге к каждой строке будут добавлены ссылки Edit (Редактировать) и Delete (Удалить). Вот как выглядит модифицированная разметка:

```
<Columns>
 <asp:CommandField ShowDeleteButton="True" ShowEditButton="True" />
 <asp:BoundField DataField="CarID" HeaderText="CarID" ReadOnly="True"
    SortExpression="CarID" />
 <asp:BoundField DataField="Make" HeaderText="Make" SortExpression="Make" />
 <asp:BoundField DataField="Color" HeaderText="Color" SortExpression="Color" />
 <asp:BoundField DataField="PetName" HeaderText="PetName" SortExpression="PetName"/>
</Columns>
```

Откроем файл Inventory.aspx.cs и добавим в него методы Delete() и Update(). Метод Delete() возвращает void, а принимает в качестве параметров carId типа int и timeStamp типа byte[]. Оба значения передаются методу, поскольку они указаны внутри атрибута DataKeyNames в разметке.

```
public void Delete(int carId, byte[] timeStamp)
{
  new InventoryRepo().Delete(carId, timeStamp);
}
```

Метод Update() возвращает void и использует привязку модели, поэтому он может принимать параметр типа Inventory. Привязка модели представляет собой средство ASP.NET MVC, которое было перенесено в инфраструктуру ASP.NET Web Forms 4.5. Оно берет все пары "имя-значение" формы, строки запроса и других источников и пробует воссоздать объект указанного типа с применением рефлексии. Различают *явную привязку модели* и *неявную привязку модели*. В каждом случае механизм привязки модели пытается присвоить значения из пар "имя-значение" (отправленной формы) соответствующим свойствам объекта желаемого типа. Если присвоить значение одному или нескольким свойствам не удалось (из-за проблем с преобразованием типов данных или ошибок проверки достоверности), то механизм привязки модели устанавливает свойство ModelState.IsValid в false. Если же всем соответствующим свойствам были успешно присвоены значения, тогда ModelState.IsValid устанавливается в true.

Для явной привязки модели необходимо вызвать метод TryUpdateModel(), передав ему экземпляр интересующего типа. В случае отказа привязки модели метод

`TryUpdateModel()` возвращает `false`. Например, метод `Update()` можно было бы реализовать так:

```
public async void Update(int carID)
{
  var inv = new Inventory() {CarID = carID};
  if (TryUpdateModel(inv))
  {
    await new InventoryRepo().SaveAsync(inventory);
  }
}
```

Для неявной привязки модели нужно передать методу объект желаемого типа в качестве параметра. Поступить так понадобится с методом `Update()`. В теле метода сначала проверим состояние модели на предмет допустимости (более подробно о проверке достоверности речь пойдет позже в приложении) и затем вызовем метод `SaveAsync()` на экземпляре `InventoryRepo`. Поскольку при вызове `SaveAsync()` используется `await`, добавим модификатор `async` к методу `Update()`:

```
public async void Update(Inventory inventory)
{
  if (ModelState.IsValid)
  {
    await new InventoryRepo().SaveAsync(inventory);
  }
}
```

После добавления модификатора `async` к методу `Update()` возникнет ошибка, когда метод `Update()` будет вызван, потому что объект `Page` не помечен как `async`. К счастью, решение сводится просто к добавлению атрибута `Async="true"` к директиве `Page`:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/MasterPage.master"
    AutoEventWireup="true" CodeFile="Inventory.aspx.cs"
    Inherits="InventoryPage"
    Async="true" %>
```

Запустив приложение, можно заметить ссылки Edit (Редактировать) и Delete (Удалить). Щелчок на ссылке Edit включает редактирование на месте (рис. Г.16). В режиме редактирования ссылки изменяются на Update (Обновить) и Cancel (Отменить).

Включение сортировки и разбиения на страницы

Элемент управления `GridView` можно легко сконфигурировать для выполнения сортировки (через гиперссылки с именами столбцов) и разбиения на страницы (посредством числовых гиперссылок или гиперссылок “следующая/предыдущая”). Чтобы сделать это, модифицируем разметку для элемента управления `GridView`, добавив атрибуты `AllowPaging`, `PageSize` и `AllowSorting`:

```
<asp:GridView ID="carsGrid" runat="server"
  AllowPaging="True" PageSize="2"
  AllowSorting="True" AutoGenerateColumns="False" CellPadding="4"
  DataKeyNames="CarID" ItemType="AutoLotDAL.Models.Inventory"
  SelectMethod="GetData" DeleteMethod="Delete" UpdateMethod="Update"
  EmptyDataText="There are no data records to display." ForeColor="#333333"
  GridLines="None">
```

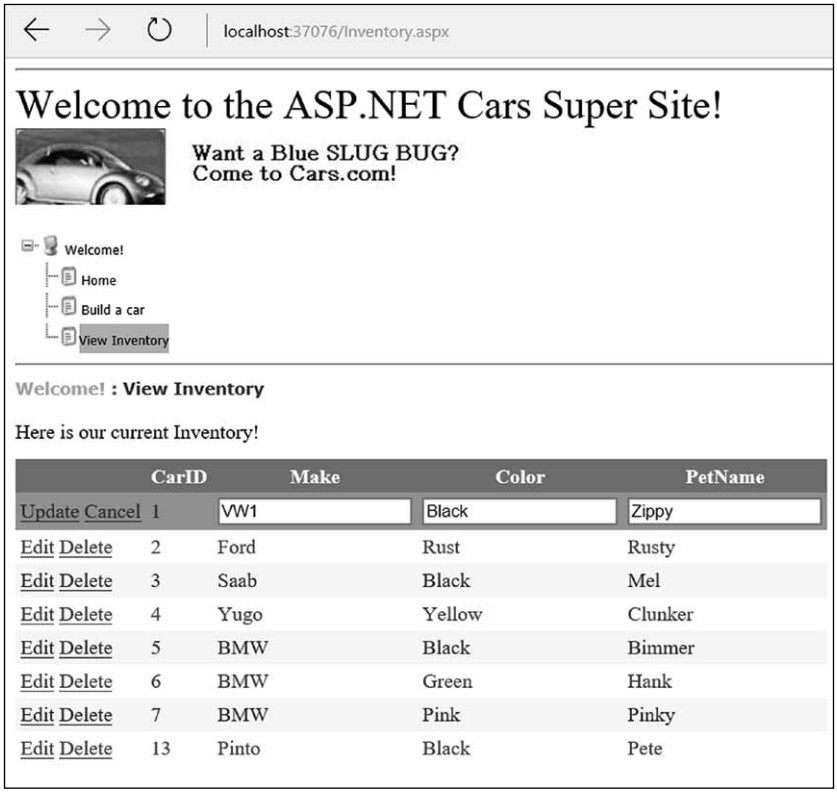

Рис. Г.16. Редактирование на месте и удаление

Если запустить приложение прямо сейчас, тогда будет получено следующее сообщение об ошибке:

```
When the DataBoundControl has paging enabled, either the SelectMethod
should return an IQueryable<ItemType> or should have all these mandatory
parameters: int startRowIndex, int maximumRows, out int totalRowCount
```

Когда в элементе DataBoundControl включено разбиение на страницы, указанный в SelectMethod метод либо должен возвращать IQueryable<ItemType>, либо должен иметь все обязательные параметры: int startRowIndex, int maximumRows, out int totalRowCount

Ошибка легко устраняется путем добавления вызова метода `AsQueryable()` после вызова `GetAll()` на экземпляре `InventoryRepo` и изменения сигнатуры метода для возвращения `IQueryable<Inventory>`:

```
public IQueryable<Inventory> GetData() =>
    new InventoryRepo().GetAll().AsQueryable();
```

На заметку! Несмотря на то что добавление вызова `AsQueryable()` решает проблему в данном примере, было бы лучше открыть доступ к версии `IQueryable` метода `GetAll()` в самом хранилище.

После запуска приложения появится возможность сортировки данных за счет щелчка на именах столбцов и прокрутки данных с помощью страничных ссылок (при условии, что в таблице `Inventory` присутствует достаточное количество записей), как показано на рис. Г.17.

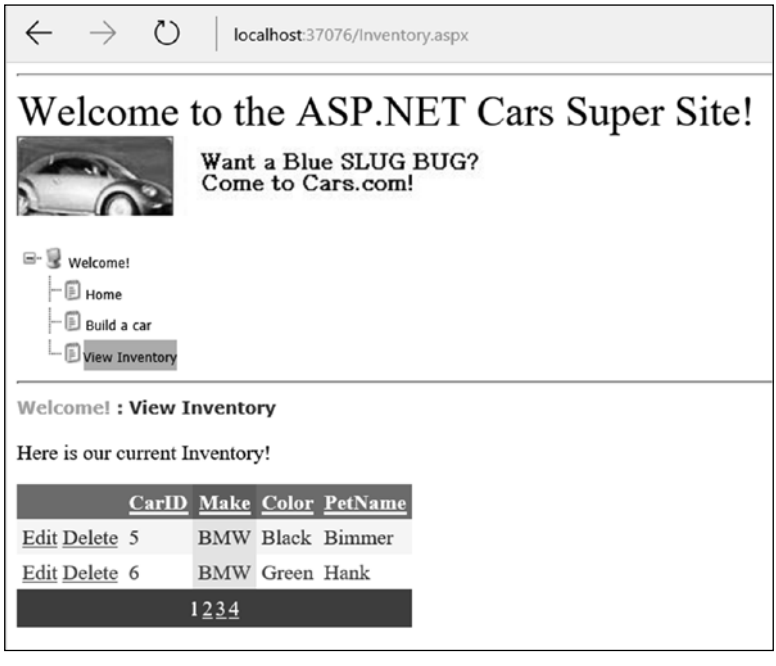

Рис. Г.17. Включение сортировки и разбиения на страницы

Включение фильтрации

Следующей задачей будет добавление к элементу управления `GridView` возможности фильтрации. И снова благодаря функциональным возможностям, добавленным в ASP.NET Web Forms 4.5, все довольно просто. Начнем с добавления элемента управления `DropDownList`, который будет привязан к отдельному списку значений `Make` в базе данных `AutoLot`. Атрибуты `DataTextField` (то, что отображается) и `DataValueField` (значение в раскрывающемся списке, основанное на выбранном элементе) установлены в `Make`. В атрибуте `SelectMethod` должен быть указан метод по имени `GetMakes()`. Основной момент в том, что в элементе управления *обязана* присутствовать настройка `runat="server"`. Кроме того, обратите внимание на дескриптор `<asp:ListItem>`. Он добавляет вариант выбора (All) (Все), если в списке ничего не выбрано. Ниже приведена разметка:

```
<asp:DropDownList ID="cboMake" SelectMethod="GetMakes"
  AppendDataBoundItems="true" AutoPostBack="true"
  DataTextField="Make" DataValueField="Make" runat="server">
  <asp:ListItem Value="" Text="(All)" />
</asp:DropDownList>
```

Теперь откроем файл `Inventory.aspx.cs` и создадим метод `GetMakes()`. В методе `GetMakes()` должен возвращаться список новых анонимных объектов, которые содержат разные значения `Make` из базы данных. Код выглядит так:

```
public IEnumerable GetMakes() =>
  new InventoryRepo().GetAll().Select(x => new {x.Make}).Distinct();
```

Метод `GetData()` также должен быть обновлен для фильтрации данных в случае передачи значения `Make`. Параметр помечается атрибутом `[Control("cboMake")]`, в котором указано имя элемента управления. Указывать имя элемента управления необязательно, если оно совпадает с именем параметра, но из-за того, что в рассматриваемом примере они отличаются, имя элемента управления необходимо предоставить. Параметр будет получать значение из элемента управления при обратной отправке формы (отсюда и требование наличия `runat="server"`) и принимать значение пустой строки, если ничего не выбрано. Вот как это делается:

```
public IQueryable<Inventory> GetData([Control("cboMake")]string make="")
{
  return string.IsNullOrEmpty(make) ?
    new InventoryRepo().GetAll().AsQueryable() :
    new InventoryRepo().GetAll().Where(x => x.Make == make).AsQueryable();
}
```

Запустив приложение, можно выбирать марку автомобиля и фильтровать результирующий набор на основе выбранного значения (рис. Г.18).

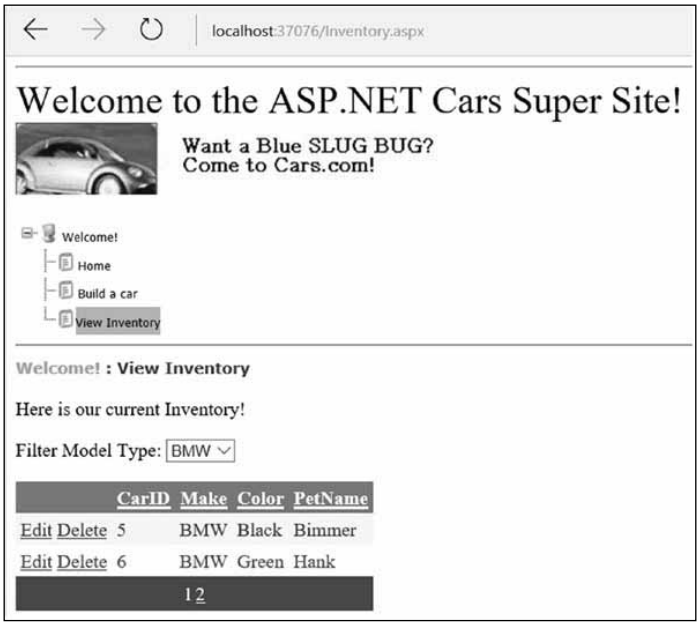

Рис. Г.18. Фильтрация данных на основе значения, выбранного в элементе управления

Проектирование страницы содержимого `BuildCar.aspx`

В рассматриваемом примере осталось еще спроектировать страницу содержимого `BuildCar.aspx`. При открытом для редактирования файле `.master` вставим новую страницу содержимого в текущий проект (с помощью пункта меню Website⇨Add Content Page; это альтернатива щелчку правой кнопкой мыши на мастер-странице проекта и выбору подходящего пункта в контекстном меню). В окне Solution Explorer переименуем новый файл в `BuildCar.aspx`.

На новой странице будет применяться элемент управления `Wizard` из Web Forms, который предоставляет простой способ для проведения конечного пользователя через последовательность связанных шагов. Здесь такие шаги будут эмулировать действие по комплектации автомобиля для покупки.

Поместим в область содержимого элементы управления `Label` и `Wizard`. Затем активизируем встроенный редактор для `Wizard` и щелкнем на ссылке Add/Remove WizardSteps (Добавить/удалить шаги мастера). Добавим четыре шага, как показано на рис. Г.19.

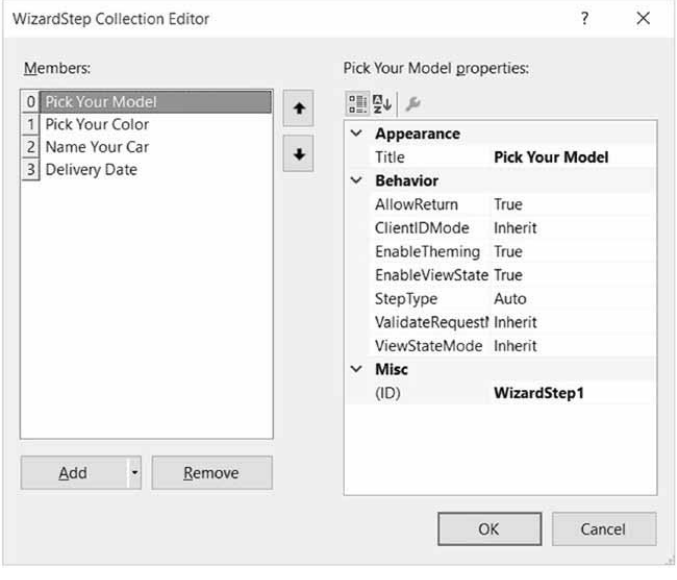

Рис. Г.19. Конфигурирование элемента управления `Wizard`

После определения шагов элемент управления `Wizard` предложит пустую область содержимого, куда можно перетаскивать элементы управления для шага мастера, выбранного в текущий момент. В рассматриваемом примере модифицируем каждый шаг, добавив следующие элементы пользовательского интерфейса (не забыв о назначении каждому элементу подходящих идентификаторов в окне Properties):

- Pick Your Model (Выбор модели) — элемент управления `TextBox`;
- Pick Your Color (Выбор цвета) — элемент управления `ListBox`;
- Name Your Car (Название автомобиля) — элемент управления `TextBox`;
- Delivery Date (Дата доставки) — элемент управления `Calendar`.

Элемент управления `ListBox` — единственный элемент пользовательского интерфейса внутри `Wizard`, который требует выполнения дополнительных шагов. Выберем его в визуальном конструкторе (сначала выбрав ссылку Pick Your Color) и заполним виджет набором цветов, используя свойство `Items` в окне Properties. В области определения `Wizard` появится разметка следующего вида:

```
<asp:ListBox ID="ListBoxColors" runat="server" Width="237px">
  <asp:ListItem>Purple</asp:ListItem>
  <asp:ListItem>Green</asp:ListItem>
  <asp:ListItem>Red</asp:ListItem>
  <asp:ListItem>Yellow</asp:ListItem>
  <asp:ListItem>Pea Soup Green</asp:ListItem>
```

```
    <asp:ListItem>Black</asp:ListItem>
    <asp:ListItem>Lime Green</asp:ListItem>
  </asp:ListBox>
```

После определения всех шагов можно обработать событие `FinishButtonClick` для автоматически сгенерированной кнопки Finish (Готово). Тем не менее, кнопка Finish не отображается до тех пор, пока не будет выбран финальный шаг мастера. Выбрав последний шаг, просто дважды щелкнем на кнопке Finish, чтобы создать обработчик события. Внутри обработчика события серверной стороны извлечем выбор из каждого элемента пользовательского интерфейса и построим строку описания, которую понадобится присвоить свойству `Text` дополнительного элемента `Label` по имени `lblOrder`:

```
public partial class BuildCarPage : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
  }
  protected void carWizard_FinishButtonClick(object sender,
    WizardNavigationEventArgs e)
  {
    // Получить каждое значение.
    string order = $"{txtCarPetName.Text},
      your { ListBoxColors.SelectedValue } { txtCarModel.Text }
      will arrive on { carCalendar.SelectedDate.ToShortDateString()}";
    // Присвоить результирующую строку метке.
    lblOrder.Text = order;
  }
}
```

Веб-сайт `AspNetCarsSite` готов. На рис. Г.20 показан элемент управления `Wizard` в действии.

Рис. Г.20. Элемент управления `Wizard` в действии

Итак, краткий обзор разнообразных элементов управления Web Forms, мастер-страниц, страниц содержимого и навигации с помощью карты сайта завершен. Далее мы займемся исследованием функциональности элементов управления проверкой достоверности Web Forms. Чтобы изолировать тематику настоящего приложения, для демонстрации приемов проверки достоверности мы построим новый веб-сайт. Однако элементы управления проверкой достоверности можно добавить и в текущий проект.

Исходный код. Веб-сайт `AspNetCarsSite` доступен в подкаталоге `Appendix_D`.

Роль элементов управления проверкой достоверности

В следующий набор элементов управления Web Forms, который мы рассмотрим, входят *элементы управления проверкой достоверности*. В отличие от других элементов управления Web Forms они не выпускают разметку HTML в целях визуализации, а применяются для выпуска кода JavaScript клиентской стороны, проверяющего достоверность данных формы. Как было показано в начале приложения, проверка достоверности данных формы клиентской стороны удобна тем, что позволяет на месте проверять данные на предмет соответствия различным ограничениям, прежде чем отправлять их обратно веб-серверу, в итоге сокращая количество затратных процедур обмена с сервером. В табл. Г.4 приведена краткая сводка по элементам управления проверкой достоверности Web Forms.

Таблица Г.4. Элементы управления проверкой достоверности Web Forms

Элемент управления	Описание
`CompareValidator`	Проверяет значение в элементе ввода на равенство заданному значению другого элемента ввода или фиксированной константе
`CustomValidator`	Позволяет строить специальную функцию проверки достоверности, которая проверяет заданный элемент управления
`RangeValidator`	Определяет, находится ли данное значение внутри заранее определенного диапазона
`RegularExpressionValidator`	Проверяет значение в ассоциированном элементе ввода на соответствие шаблону регулярного выражения
`RequiredFieldValidator`	Проверяет заданный элемент ввода на наличие значения (т.е. что он не пуст)
`ValidationSummary`	Отображает итог по всем ошибкам проверки достоверности страницы в формате простого списка, маркированного списка или одиночного абзаца. Ошибки могут отображаться встроенным способом и/или во всплывающем окне сообщения

Все элементы управления проверкой достоверности (кроме `ValidationSummary`) в конечном итоге являются производными от общего базового класса по имени `System.Web.UI.WebControls.BaseValidator` и потому обладают набором общих функциональных возможностей. Основные члены класса `BaseValidator` перечислены в табл. Г.5.

Таблица Г.5. Общие члены элементов управления проверкой достоверности Web Forms

Член	Описание
`ControlToValidate`	Получает или устанавливает элемент управления, подлежащий проверке достоверности
`Display`	Получает или устанавливает поведение сообщений об ошибках в элементе управления проверкой достоверности
`EnableClientScript`	Получает или устанавливает значение, которое указывает, включена ли проверка достоверности клиентской стороны
`ErrorMessage`	Получает или устанавливает текст для сообщения об ошибке
`ForeColor`	Получает или устанавливает цвет сообщения, отображаемого при неудачной проверке достоверности

Чтобы проиллюстрировать работу с элементами управления проверкой достоверности, создадим новый проект пустого веб-сайта по имени `ValidatorCtrls` и вставим в него новую веб-форму по имени `Default.aspx`. Для начала поместим на страницу четыре (именованных) элемента управления `TextBox` (с четырьмя соответствующими описательными элементами `Label`). Затем поместим на страницу рядом с каждым полем элементы управления `RequiredFieldValidator`, `RangeValidator`, `RegularExpressionValidator` и `CompareValidator`. Наконец, добавим элементы `Button` и `Label`. На рис. Г.21 показан возможный вид компоновки.

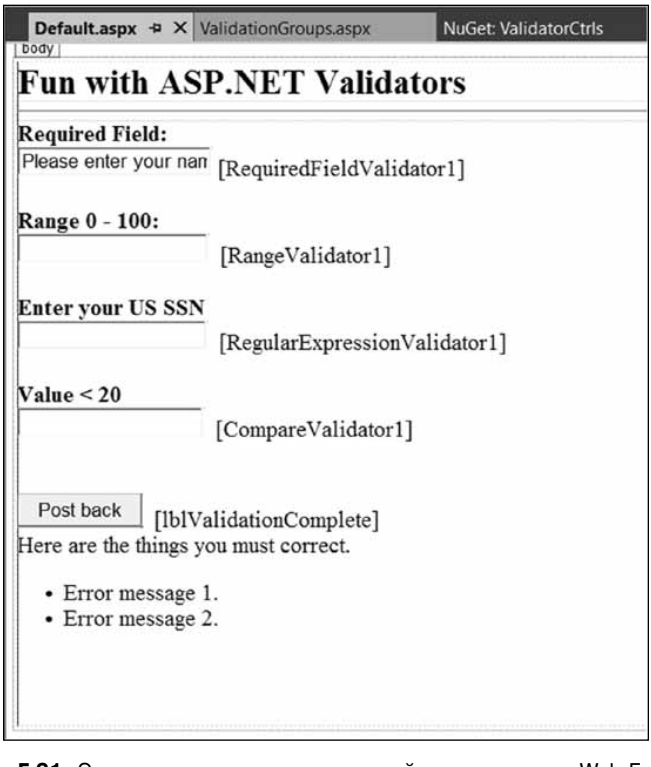

Рис. Г.21. Элементы управления проверкой достоверности Web Forms обеспечат корректность данных формы перед ее обратной отправкой

Располагая начальным пользовательским интерфейсом для проведения экспериментов, давайте пройдемся по процессу конфигурирования каждого элемента управления проверкой достоверности и посмотрим на конечный результат всех действий. Но сначала понадобится модифицировать текущий файл `Web.config`, разрешив обработку клиентской стороны для элементов управления проверкой достоверности.

Включение поддержки проверки достоверности с помощью кода JavaScript клиентской стороны

Начиная с версии ASP.NET 4.5, в Microsoft ввели новую настройку для управления способом реагирования элементов управления проверкой достоверности во время выполнения. Открыв файл `Web.config` после создания веб-приложения ASP.NET, в нем можно обнаружить следующую настройку:

```
<appSettings>
 <add key="ValidationSettings:UnobtrusiveValidationMode" value="WebForms" />
</appSettings>
```

Когда эта настройка присутствует в файле `Web.config`, веб-сайт будет проводить проверку достоверности с использованием разнообразных атрибутов данных HTML 5.0, а не отправлять обратно порции кода JavaScript клиентской стороны для обработки веб-браузером. Поскольку HTML 5.0 в книге подробно не рассматривается, после создания проекта приложения Web Forms (вместо веб-сайта) показанную выше строку необходимо закомментировать (или удалить), чтобы текущий пример проверки достоверности работал корректно.

Элемент управления `RequiredFieldValidator`

Конфигурировать элемент управления `RequiredFieldValidator` легко: нужно просто надлежащим образом установить свойства `ErrorMessage` и `ControlToValidate` в окне Properties среды Visual Studio. Ниже показана результирующая разметка, которая обеспечивает то, что текстовое поле `txtRequiredField` не будет пустым:

```
<asp:RequiredFieldValidator ID="RequiredFieldValidator1"
  runat="server" ControlToValidate="txtRequiredField"
  ErrorMessage="Oops! Need to enter data.">
</asp:RequiredFieldValidator>
```

Класс `RequiredFieldValidator` поддерживает свойство `InitialValue`. Его можно применять для проверки того факта, что пользователь ввел в связанном элементе `TextBox` какое-то значение, отличающееся от начального. Например, когда пользователь впервые получает страницу, элемент управления `TextBox` можно сконфигурировать так, чтобы он содержал значение “Please enter your name” (“Введите свое имя”). Если свойство `InitialValue` элемента `RequiredFieldValidator` не установлено, то исполняющая среда будет предполагать, что строка “Please enter your name” является допустимым значением. Таким образом, чтобы обязательное поле `TextBox` было допустимым только в случае, когда в нем введено значение, отличающееся от строки “Please enter your name”, сконфигурируем виджет, как показано ниже:

```
<asp:RequiredFieldValidator ID="RequiredFieldValidator1"
  runat="server" ControlToValidate="txtRequiredField"
  ErrorMessage="Oops! Need to enter data."
  InitialValue="Please enter your name">
</asp:RequiredFieldValidator>
```

Элемент управления RegularExpressionValidator

Элемент управления RegularExpressionValidator можно использовать, когда к символам, введенным в заданном поле, необходимо применить шаблон. Например, чтобы обеспечить наличие в элементе TextBox корректного номера карточки социального страхования США, виджет можно было бы определить следующим образом:

```
<asp:RegularExpressionValidator ID="RegularExpressionValidator1"
  runat="server" ControlToValidate="txtRegExp"
  ErrorMessage="Please enter a valid US SSN."
  ValidationExpression="\d{3}-\d{2}-\d{4}">
</asp:RegularExpressionValidator>
```

Обратите внимание на то, как в RegularExpressionValidator определено свойство ValidationExpression. Если вы ранее не работали с регулярными выражениями, тогда в рассматриваемом примере достаточно знать лишь то, что они используются для проверки соответствия строки определенному шаблону. Здесь выражение "\d{3}-\d{2}-\d{4}" получает стандартный номер карточки социального страхования США в форме *xxx-xx-xxxx* (где *x* — десятичная цифра).

Указанное конкретное регулярное выражение вполне очевидно; тем не менее, предположим, что требуется проверить правильность телефонного номера в Японии. Нужное выражение становится намного сложнее: "(0\d{1,4}-|\(0\d{1,4}\)?)?\d{1,4}-\d{4}". Хорошая новость в том, что при выборе свойства ValidationExpression в окне Properties по щелчку на кнопке с троеточием доступен заранее определенный список распространенных регулярных выражений.

На заметку! За программное манипулирование регулярными выражениями в .NET отвечают два пространства имен — System.Text.RegularExpressions и System.Web.RegularExpressions.

Элемент управления RangeValidator

В дополнение к свойствам MinimumValue и MaximumValue класс RangeValidator имеет свойство по имени Type. С учетом того, что мы заинтересованы в проверке пользовательского ввода на вхождение в диапазон целых чисел, понадобится указать тип Integer (который *не* является стандартной установкой):

```
<asp:RangeValidator ID="RangeValidator1"
  runat="server" ControlToValidate="txtRange"
  ErrorMessage="Please enter value between 0 and 100."
  MaximumValue="100" MinimumValue="0" Type="Integer">
</asp:RangeValidator>
```

Класс RangeValidator также может применяться для проверки вхождения в диапазон денежных значений, дат, чисел с плавающей точкой и строковых данных (стандартная установка).

Элемент управления CompareValidator

Наконец, обратите внимание, что CompareValidator поддерживает следующее свойство Operator:

```
<asp:CompareValidator ID="CompareValidator1" runat="server"
  ControlToValidate="txtComparison"
  ErrorMessage="Enter a value less than 20." Operator="LessThan"
  ValueToCompare="20" Type="Integer">
</asp:CompareValidator>
```

Поскольку целью этого элемента управления проверкой достоверности является сравнение значения в текстовом поле с другим значением, используя бинарную операцию, не должен удивлять тот факт, что свойство `Operator` способно принимать такие значения, как `LessThan`, `GreaterThan`, `Equal` и `NotEqual`. Свойство `ValueToCompare` применяется для установки значения, с которым производится сравнение. Обратите внимание, что в `Type` указано `Integer`. По умолчанию `CompareValidator` будет выполнять сравнение со строковыми значениями.

На заметку! Элемент управления `CompareValidator` также может быть сконфигурирован для сравнения со значением внутри другого элемента управления Web Forms (вместо жестко закодированного значения), используя свойство `ControlToCompare`.

Чтобы завершить код страницы, обработаем событие `Click` элемента управления `Button` и проинформируем пользователя об успешном прохождении логики проверки достоверности:

```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
  }
  protected void btnPostback_Click(object sender, EventArgs e)
  {
    lblValidationComplete.Text = "You passed validation!";
  }
}
```

Загрузим готовую страницу в браузер. Пока что не должны быть видны какие-то заметные изменения. Однако при попытке щелкнуть на кнопке Submit (Отправить) после ввода некорректных данных появится сообщение об ошибке. После ввода правильных данных сообщение об ошибке исчезнет и произойдет обратная отправка. Просмотрев разметку HTML, визуализированную браузером, можно обнаружить, что элементы управления проверкой достоверности генерируют функцию JavaScript клиентской стороны, которая задействует специфическую библиотеку функций JavaScript, автоматически загружаемую на пользовательскую машину. Как только проверка достоверности успешно прошла, данные формы отправляются обратно серверу, где исполняющая среда проведет ту же самую проверку еще раз на веб-сервере (просто чтобы удостовериться в том, что данные не были искажены по пути).

Кстати, если HTTP-запрос был отправлен браузером, который не поддерживает JavaScript клиентской стороны, то вся проверка достоверности выполняется на сервере. Таким образом, программировать элементы управления проверкой достоверности можно, не задумываясь о целевом браузере; возвращенная страница HTML переадресует обработку ошибок веб-серверу.

Создание итоговой панели проверки достоверности

Следующая тема, касающаяся проверки достоверности, которую мы рассмотрим — применение виджета `ValidationSummary`. В настоящий момент каждый элемент управления проверкой достоверности отображает свое сообщение об ошибке в месте, куда он был помещен во время проектирования. Во многих случаях именно это и требуется. Однако в сложных формах с многочисленными виджетами ввода вариант с засорением формы многочисленными надписями красного цвета может не устроить. С использованием элемента управления `ValidationSummary` можно заставить все типы проверки достоверности отображать свои сообщения об ошибках в определенном месте страницы.

Первый шаг предусматривает помещение элемента управления `ValidationSummary` в файл `.aspx`. Дополнительно можно установить его свойство `HeaderText`, а также свойство `DisplayMode`, которое по умолчанию будет отображать список всех сообщений об ошибках в виде маркированного списка.

```
<asp:ValidationSummary id="ValidationSummary1"
  runat="server" Width="353px"
  HeaderText="Here are the things you must correct.">
</asp:ValidationSummary>
```

Далее понадобится установить свойство `Display` в `None` для всех индивидуальных элементов управления проверкой достоверности (например, `RequiredFieldValidator`, `RangeValidator` и т.д.). Это гарантирует отсутствие дублированных сообщений об ошибках при каждой неудачной проверке достоверности (одно в итоговой панели и еще одно в месте расположения элемента управления проверкой достоверности). На рис. Г.22 приведена итоговая панель в действии.

И последнее: если взамен желательно отображать сообщения об ошибках в окне сообщений клиентской стороны, тогда следует установить свойство `ShowMessageBox` элемента управления `ValidationSummary` в `true`, а свойство `ShowSummary` — в `false`.

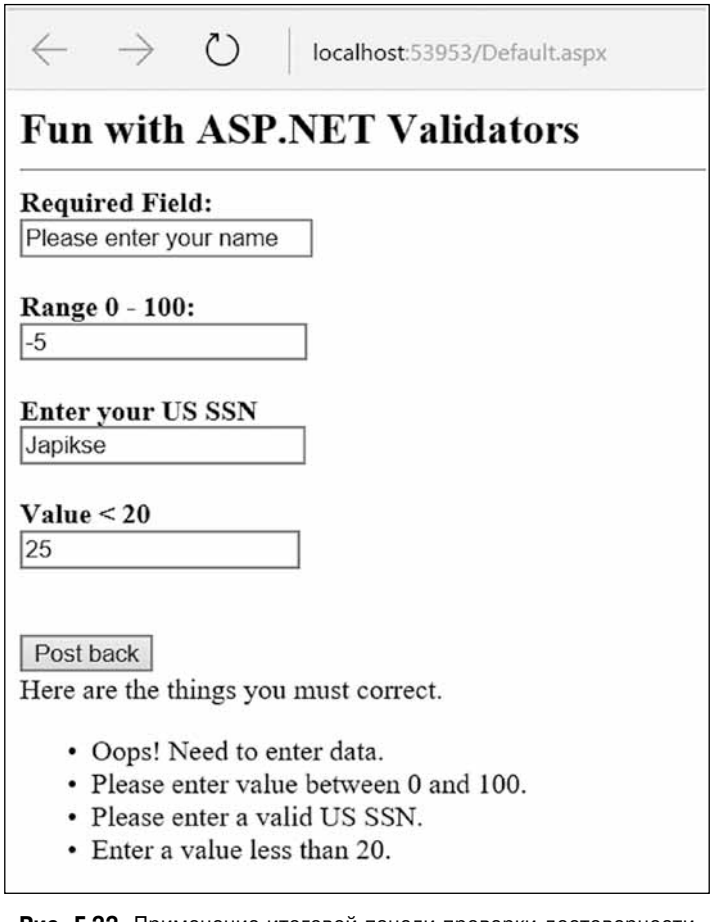

Рис. Г.22. Применение итоговой панели проверки достоверности

Определение групп проверки достоверности

Также есть возможность определять *группы*, к которым принадлежат элементы управления проверкой достоверности. Такой прием очень удобен при наличии на странице областей, работающих как единое целое. Например, в одном объекте `Panel` может существовать одна группа элементов управления, предназначенная для ввода пользователем адреса электронной почты, и а в другом объекте `Panel` — другая группа, которая отвечает за ввод информации о кредитной карте. Используя группы, можно сконфигурировать каждый набор элементов управления для независимой проверки достоверности.

Вставьте в текущий проект новую страницу по имени `ValidationGroups.aspx`, на которой определены два элемента управления `Panel`. Первый объект `Panel` будет содержать элемент `TextBox` для пользовательского ввода (проверяемого посредством `RequiredFieldValidator`), а во втором объекте `Panel` будет находиться элемент `TextBox` для ввода номера карточки социального страхования США (проверяемого с помощью `RegularExpressionValidator`). На рис. Г.23 показан возможный вид пользовательского интерфейса.

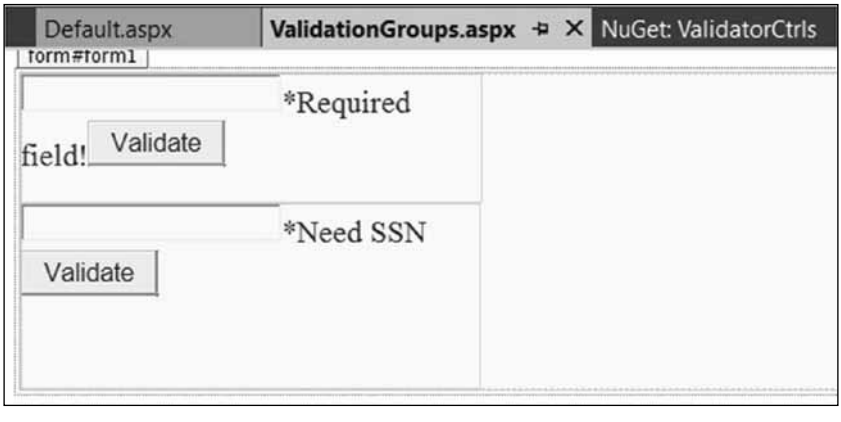

Рис. Г.23. Объекты `Panel` будут независимо конфигурировать свои области ввода

Чтобы обеспечить независимое функционирование элементов управления проверкой достоверности, мы назначим элемент управления проверкой достоверности и проверяемый им элемент управления уникально именованной группе с применением свойства `ValidationGroup`. В следующем примере разметки обратите внимание на то, что используемые обработчики события `Click` по существу являются пустыми заглушками в файле кода:

```
<form id="form1" runat="server">
  <asp:Panel ID="Panel1" runat="server" Height="83px" Width="296px">
    <asp:TextBox ID="txtRequiredData" runat="server"
                 ValidationGroup="FirstGroup">
    </asp:TextBox>
    <asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
      ErrorMessage="*Required field!" ControlToValidate="txtRequiredData"
      ValidationGroup="FirstGroup">
    </asp:RequiredFieldValidator>
    <asp:Button ID="bntValidateRequired" runat="server"
      OnClick="bntValidateRequired_Click"
      Text="Validate" ValidationGroup="FirstGroup" />
  </asp:Panel>
```

```
  <asp:Panel ID="Panel2" runat="server" Height="119px" Width="295px">
    <asp:TextBox ID="txtSSN" runat="server"
      ValidationGroup="SecondGroup">
    </asp:TextBox>
    <asp:RegularExpressionValidator ID="RegularExpressionValidator1"
        runat="server" ControlToValidate="txtSSN"
        ErrorMessage="*Need SSN" ValidationExpression="\d{3}-\d{2}-\d{4}"
        ValidationGroup="SecondGroup">
    </asp:RegularExpressionValidator> 
    <asp:Button ID="btnValidateSSN" runat="server"
        OnClick="btnValidateSSN_Click" Text="Validate"
        ValidationGroup="SecondGroup" />
  </asp:Panel>
</form>
```

Теперь щелкнем правой кнопкой мыши на поверхности визуального конструктора этой страницы и выберем в контекстном меню пункт View In Browser (Просмотреть в браузере), чтобы удостовериться, что проверка данных в элементах каждой панели происходит во взаимоисключающей манере.

Проверка достоверности с помощью аннотаций данных

В дополнение к элементам управления проверкой достоверности инфраструктура ASP.NET Web Forms поддерживает проверку достоверности с применением аннотаций данных. Вспомните из главы 22, что классы модели можно помечать атрибутами, которые определяют бизнес-требования для модели (скажем, `Required`). Используя новый элемент управления `ModelErrorMessage` и дополнительное свойство элемента управления `ValidationSummary`, можно представлять ошибки, связанные с нарушениями аннотаций данных, с помощью совсем небольшого объема кода.

Создание модели

Хотя определенно можно было бы обратиться к библиотеке `AutoLotDAL` из приложения В, ради упрощения мы создадим новый класс `Inventory`. Начнем с добавления новой папки `App_Code`, щелкнув правой кнопкой мыши на имени проекта и выбрав в контекстном меню пункт Add ASP.NET Folder⇨App_Code (Добавить папку ASP.NET⇨`App_Code`). Добавим в эту папку новый файл класса по имени `Inventory.cs` и поместим в него следующий код (не забыв об операторе `using` для пространства имен `System.ComponentModel.DataAnnotations`):

```
public class Inventory
{
  [Key,Required]
  public int CarID { get; set; }

  [Required(ErrorMessage="Make is required.")]
  [StringLength(30,ErrorMessage="Make can only be 30 charaters or less")]
  public string Make { get; set; }

  [Required, StringLength(30)]
  public string Color { get; set; }

  [StringLength(30, ErrorMessage = "Pet Name can only be 30 charaters or less")]
  public string PetName { get; set; }
}
```

Построение пользовательского интерфейса

Далее добавим новую веб-форму по имени `Annotations.aspx`. Поместим внутрь дескриптора `Form` элемент управления `asp:FormView`, который позволяет очень легко переключаться между режимами отображения, редактирования и вставки. Модифицируем атрибуты, как показано ниже (установка `DefaultMode="Insert"` приводит к загрузке элемента управления `FormView` в режиме вставки):

```
<asp:FormView runat="server" ID="fvDataBinding" DataKeyNames="CarID"
  ItemType="Inventory" DefaultMode="Insert" InsertMethod="SaveCar"
  UpdateMethod="UpdateCar" SelectMethod="GetCar">
```

Теперь создадим разметку `ItemTemplate`. Это содержимое, которое будет отображаться в режиме только для чтения. Атрибут `ItemType` строго типизирует элемент управления `FormView` и поддерживает синтаксис `<%# Item.`*`ИмяПоля`*` %>`. Добавим следующую разметку:

```
<ItemTemplate>
  <table style="width:100%">
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="make">Make:
          </asp:Label>
      </td>
      <td><asp:Label runat="server" ID="make" Text='<%# Item.Make %>' />
      </td>
    </tr>
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="color">Color:
          </asp:Label>
      </td>
      <td><asp:Label runat="server" ID="color" Text='<%#: Item.Color %>' />
      </td>
    </tr>
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="petname">Pet Name:
          </asp:Label></td>
      <td><asp:Label runat="server" ID="customerAge" Text='<%#: Item.PetName %>' />
      </td>
    </tr>
    <tr>
      <td colspan="2">
        <asp:Button ID="EditButton" runat="server" CommandName="Edit"
                    Text="Edit" /> 
      </td>
    </tr>
  </table>
</ItemTemplate>
```

Атрибут `CommandName="Edit"` указывает на то, что щелчок на кнопке будет переводить `FormView` в режим редактирования, в котором отображается разметка `EditItemTemplate`, создаваемая следующей. Существует ряд отличий между синтаксисом для шаблонов редактирования и отображения. Прежде всего, в шаблонах редактирования вместо синтаксиса `<%# Item.`*`ИмяПоля`*` %>` применяется синтаксис `<%# BindItem.`*`ИмяПоля`*` %>`. С помощью `BindItem` элемент управления настраивается на двухстороннюю привязку. Еще одно отличие касается элемента управления `ModelErrorMessage`, сопровождающего элементы управления редактированием. Элемент `ModelErrorMessage` будет отображать любые ошибки привязки модели для свойства, указанного в `ModelStateKey`.

Обратите внимание, что все зависит от строгой типизации `FormView`.

```
<EditItemTemplate>
  <table style="width:100%">
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="make">Make:
          </asp:Label>
      </td>
      <td>
        <asp:TextBox runat="server" ID="make" Text='<%# BindItem.Make %>' />
        <asp:ModelErrorMessage ModelStateKey="make" runat="server"
          ForeColor="Red" />
      </td>
    </tr>
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="color">Color:
          </asp:Label>
      </td>
      <td>
        <asp:TextBox runat="server" ID="color" Text='<%#: BindItem.Color %>' />
        <asp:ModelErrorMessage ModelStateKey="color" runat="server"
          ForeColor="Red" />
      </td>
    </tr>
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="petname">Pet Name:
          </asp:Label>
      </td>
      <td>
<asp:TextBox ID="petname" runat="server" Text='<%#: BindItem.PetName %>' />
<asp:ModelErrorMessage ModelStateKey="petname" runat="server" ForeColor="Red" />
      </td>
    </tr>
    <tr>
      <td colspan="2">
        <asp:Button runat="server" CommandName="Update" Text="Save" />
        <asp:Button runat="server" CommandName="Cancel" Text="Cancel"
            CausesValidation="false" />
      </td>
    </tr>
  </table>
</EditItemTemplate>
```

Разметка в `InsertItemTemplate` фактически является той же, что и в `EditItemTemplate`, с единственной разницей (помимо имени дескриптора) — именем команды (`CommandName`) для `Button` будет `Insert`:

```
<InsertItemTemplate>
  <table style="width:100%">
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="make">Make:
          </asp:Label>
      </td>
      <td>
<asp:TextBox runat="server" ID="make" Text='<%#: BindItem.Make %>' />
<asp:ModelErrorMessage ModelStateKey="make" runat="server" ForeColor="Red" />
      </td>
    </tr>
```

```
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="color">Color:
          </asp:Label>
      </td>
      <td>
<asp:TextBox runat="server" ID="color" Text='<%#: BindItem.Color %>' />
<asp:ModelErrorMessage ModelStateKey="color" runat="server" ForeColor="Red" />
      </td>
    </tr>
    <tr>
      <td><asp:Label runat="server" AssociatedControlID="petname">Pet Name:
          </asp:Label>
      </td>
      <td>
<asp:TextBox ID="petname" runat="server" Text='<%#: BindItem.PetName %>' />
<asp:ModelErrorMessage ModelStateKey="petname" runat="server" ForeColor="Red" />
      </td>
    </tr>
    <tr>
      <td colspan="2">
        <asp:Button runat="server" CommandName="Insert" Text="Save" />
      </td>
    </tr>
  </table>
</InsertItemTemplate>
```

Закроем дескриптор элемента управления `FormView`:

```
</asp:FormView>
```

Наконец, добавим элемент управления `ValidationSummary`. Отличие от предшествующих примеров здесь в том, что свойство `ShowModelStateErrors` устанавливается в `true`. Это инструктирует элемент управления об отображении любых ошибок привязки модели. Поместим приведенную ниже разметку после закрывающего дескриптора элемента управления `FormView`:

```
<asp:ValidationSummary runat="server" ShowModelStateErrors="true"
  ForeColor="Red" HeaderText="Please check the following errors:" />
```

Добавление кода

Мы собираемся добавить объем кода, достаточный для демонстрации проверки достоверности. В реальном приложении методы, поддерживающие элемент управления `FormView`, обращались бы к уровню DAL, а не только взаимодействовали с локальной переменной. Но ради сохранения простоты примера добавим в `Annotations.aspx.cs` представленный далее код. Нам придется добавить NuGet-пакет `Microsoft.CodeDom.Providers.DotNetCompilerPlatform`, чтобы включить функциональные средства C# 6. Обратите внимание на использование неявной привязки модели в методе `SaveCar()` и явной привязки модели в методе `UpdateCar()`.

```
private Inventory _model = null;
public void SaveCar(Inventory car)
{
  if (ModelState.IsValid)
  {
    _model = car;
    // Добавить новую запись
  }
}
```

```
public void UpdateCar(int carID)
{
  Inventory car = new Inventory();
  if (TryUpdateModel(car))
  {
    _model = car;
    // Обновить запись
  }
}
public Inventory GetCar() => _model;
```

Тестирование приложения

После запуска приложения отобразится страница, подобная показанной на рис. Г.24.

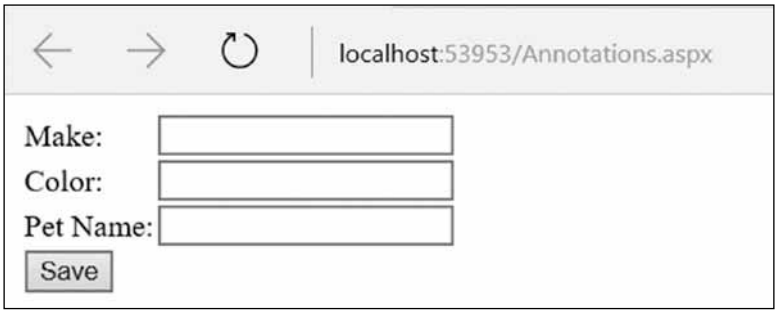

Рис. Г.24. Элемент управления `FormView` в режиме вставки

Оставим поля Make (Производитель) и Color (Цвет) пустыми, введем в поле Pet Name (Дружественное имя) строку длиннее 30 символов и щелкнем на кнопке Save (Сохранить). Должны быть получены те же самые сообщения об ошибках, что и на рис. Г.25.

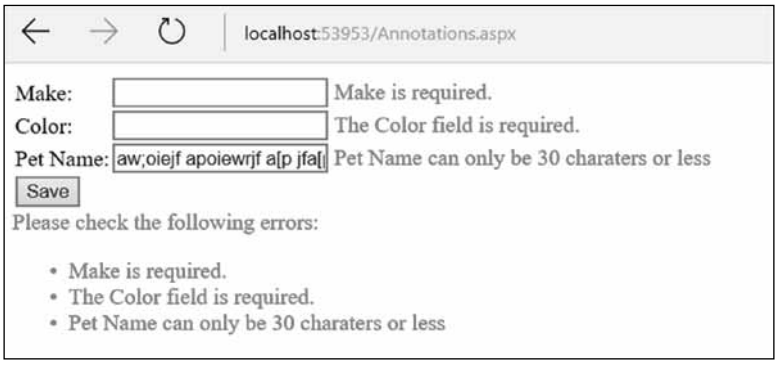

Рис. Г.25. Элемент управления `FormView` и отображение сообщений об ошибках

Как и другие элементы управления Web Forms, элементы `ModelErrorMessage` и `ValidationSummary` могут быть стилизованы намного лучше, чем было сделано здесь.

Исходный код. Веб-сайт `ValidatorCtrls` доступен в подкаталоге `Appendix_D`.

Работа с темами

К настоящему моменту мы поработали с многочисленными элементами управления Web Forms. Было показано, что каждый элемент управления открывает доступ к набору свойств (многие из которых унаследованы от `System.Web.UI.Webcontrols.WebControl`), позволяющих настраивать внешний вид и поведение пользовательского интерфейса (цвет фона, размер шрифта, стиль рамки и т.п.). Конечно, на многостраничном веб-сайте принято определять общий внешний вид и поведение виджетов различных типов. Например, все элементы `TextBox` могут быть сконфигурированы на поддержку заданного шрифта, все элементы `Button` — принятого в компании вида, а все `Calendar` — ярко-синей рамки.

Очевидно, что установка *одних и тех же* значений для свойств *каждого* виджета на *каждой* странице веб-сайта была бы очень трудоемкой (и чреватой ошибками) задачей. Даже если вы в состоянии вручную обновить свойства каждого виджета пользовательского интерфейса на каждой странице, только вообразите, насколько утомительным может стать процесс изменения цвета фона для каждого элемента `TextBox`, когда в этом возникнет необходимость. Ясно, что должен существовать более эффективный способ применения настроек пользовательского интерфейса на уровне всего сайта.

Один из подходов к упрощению установки общего внешнего вида и поведения пользовательского интерфейса предусматривает определение *таблиц стилей*. Если у вас есть опыт разработки веб-приложений, то вам известно, что таблицы стилей определяют общий набор настроек пользовательского интерфейса, которые применяются в браузере. Как и можно было ожидать, элементам управления Web Forms может быть назначен стиль за счет установки свойства `CssStyle`.

Тем не менее, инфраструктура Web Forms поставляется с дополняющей технологией для определения общего пользовательского интерфейса, которая называется *темами*. В отличие от таблиц стилей темы применяются на веб-сервере (а не в браузере) и могут использоваться программно или декларативно. Поскольку тема применяется на веб-сервере, она имеет доступ ко всем ресурсам веб-сайта серверной стороны. Более того, темы определяются путем написания той же разметки, которую можно найти в любом файле `.aspx` (вы наверняка согласитесь, что синтаксис таблиц стилей чересчур сжат).

Вспомните из приложения B, что веб-приложения ASP.NET могут определять любое количество специальных каталогов, одним из которых является `App_Themes`. Этот подкаталог может иметь подкаталоги, каждый из которых представляет одну из возможных тем веб-сайта. Например, на рис. Г.26 показан подкаталог `App_Themes`, содержащий три подкаталога, каждый из которых имеет набор файлов, образующих саму тему.

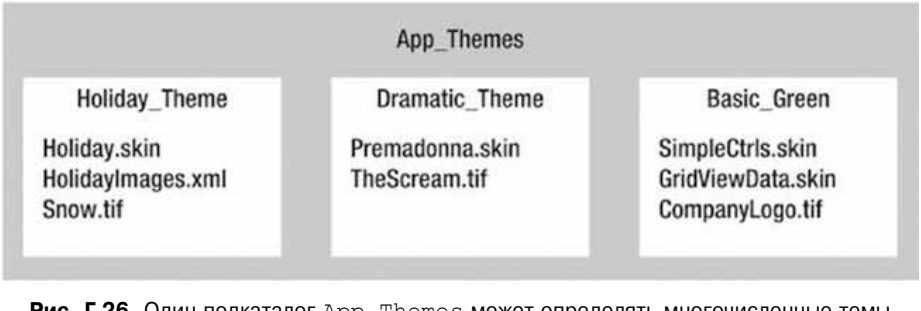

Рис. Г.26. Один подкаталог `App_Themes` может определять многочисленные темы

Файлы .skin

Каждый подкаталог темы обязательно содержит файл обложки .skin. Такие файлы определяют внешний вид и поведение различных веб-элементов управления. В целях иллюстрации создадим новый проект пустого веб-сайта по имени FunWithThemes и вставим в него новую веб-форму Default.aspx. Поместим на нее элементы управления Calendar, TextBox и Button. Конфигурировать каким-то специальным образом эти элементы управления не понадобится, а их точные имена не важны. Как будет показано, данные элементы управления будут служить целями для специальных обложек.

Добавим новый файл .skin (используя пункт меню Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент)) по имени BasicGreen.skin (рис. Г.27).

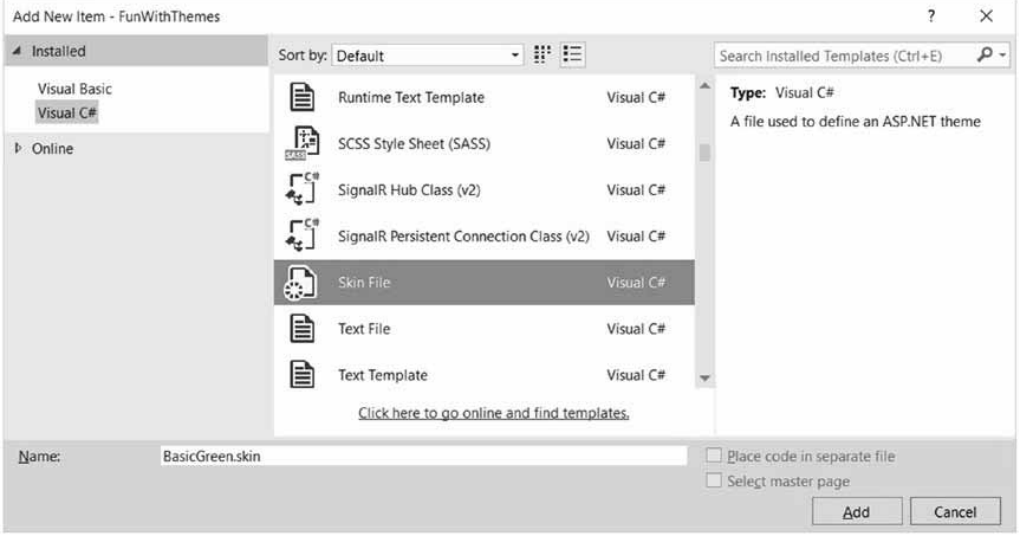

Рис. Г.27. Вставка файла .skin

Среда Visual Studio предложит подтвердить добавление этого файла в App_Themes (что как раз и требуется). Если теперь заглянуть в окно Solution Explorer, то в подкаталоге App_Themes будет виден подкаталог BasicGreen, который содержит новый файл BasicGreen.skin.

В файле .skin определяется внешность и поведение разнообразных виджетов с применением синтаксиса для объявления элементов управления Web Forms. К сожалению, в IDE-среде отсутствует поддержка визуального конструирования файлов .skin. Один из способов сокращения объема ввода заключается в добавлении к программе временного файла .aspx (скажем, temp.aspx), который может использоваться при построении пользовательского интерфейса виджетов с помощью визуального конструктора страниц Visual Studio.

Результирующую разметку можно затем скопировать в файл .skin. Однако вы *обязаны* удалить атрибут ID из каждого веб-элемента управления! Смысл вполне очевиден: мы хотим определить внешний вид и поведение не *отдельного* элемента Button (например), а *всех* элементов Button. Учитывая сказанное, вот как может выглядеть разметка внутри файла BasicGreen.skin, которая определяет стандартный внешний вид и поведение для типов Button, TextBox и Calendar:

```
<asp:Button runat="server" BackColor="#80FF80"/>
<asp:TextBox runat="server" BackColor="#80FF80"/>
<asp:Calendar runat="server" BackColor="#80FF80"/>
```

Обратите внимание, что каждый виджет по-прежнему содержит атрибут `runat="server"` (что обязательно), и ни одному из них не назначен атрибут `ID`.

Давайте определим вторую тему по имени `CrazyOrange`. В окне Solution Explorer щелкнем правой кнопкой мыши на папке `App_Themes` и добавим новую тему по имени `CrazyOrange`. В итоге внутри папки `App_Themes` сайта создается новый подкаталог.

Затем щелкнем правой кнопкой мыши на новой папке `CrazyOrange` в окне Solution Explorer и выберем в контекстном меню пункт Add New Item (Добавить новый элемент). В открывшемся диалоговом окне добавим новый файл `.skin`. Модифицируем файл `CrazyOrange.skin`, определив уникальный внешний вид и поведение для тех же самых веб-элементов управления:

```
<asp:Button runat="server" BackColor="#FF8000"/>
<asp:TextBox runat="server" BackColor="#FF8000"/>
<asp:Calendar BackColor="White" BorderColor="Black"
  BorderStyle="Solid" CellSpacing="1"
  Font-Names="Verdana" Font-Size="9pt" ForeColor="Black" Height="250px"
  NextPrevFormat="ShortMonth" Width="330px" runat="server">
  <SelectedDayStyle BackColor="#333399" ForeColor="White" />
  <OtherMonthDayStyle ForeColor="#999999" />
  <TodayDayStyle BackColor="#999999" ForeColor="White" />
  <DayStyle BackColor="#CCCCCC" />
  <NextPrevStyle Font-Bold="True" Font-Size="8pt" ForeColor="White" />
  <DayHeaderStyle Font-Bold="True" Font-Size="8pt"
    ForeColor="#333333" Height="8pt" />
  <TitleStyle BackColor="#333399" BorderStyle="Solid"
    Font-Bold="True" Font-Size="12pt"
    ForeColor="White" Height="12pt" />
</asp:Calendar>
```

В данный момент окно Solution Explorer должно выглядеть так, как показано на рис. Г.28.

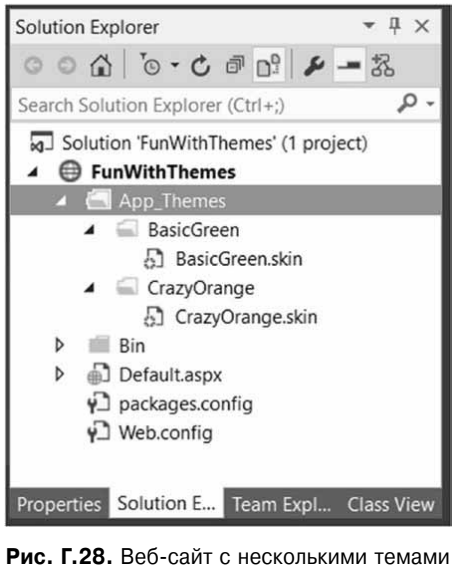

Рис. Г.28. Веб-сайт с несколькими темами

Теперь, когда сайт имеет несколько определенных тем, следующий логический шаг заключается в их применении к страницам. Как и можно было догадаться, делать это можно многими способами.

На заметку! Дизайн рассматриваемых примеров тем довольно прост (с целью экономии места на печатной странице). Вы можете усовершенствовать их по своему вкусу.

Применение тем ко всему сайту

Если необходимо обеспечить оформление каждой страницы сайта согласно той же самой теме, тогда проще всего обновить файл `Web.config`. Откроем текущий файл `Web.config` и определим элемент `<pages>` внутри области корневого элемента `<system.web>`. Добавление атрибута `theme` к элементу `<pages>` гарантирует применение одной и той же темы ко всем страницам сайта (разумеется, значением атрибута должно быть имя одного из подкаталогов внутри `App_Themes`). Вот основное изменение:

```
<configuration>
  <system.web>
    ...
    <pages controlRenderingCompatibilityVersion="4.5"
          theme="BasicGreen">
    </pages>
  </system.web>
</configuration>
```

В результате открытия страницы обнаружится, что каждый виджет имеет пользовательский интерфейс, определенный темой `BasicGreen`. Если изменить значение атрибута `theme` на `CrazyOrange` и снова открыть страницу, то пользовательский интерфейс будет соответствовать тому, который определен темой `CrazyOrange`.

Применение тем на уровне страницы

Темы также можно назначать на уровне страниц. Такой прием удобен в различных обстоятельствах. Например, в файле `Web.config` может быть определена тема для всего сайта (как описано в предыдущем разделе), но определенной странице нужно назначить другую тему. В таком случае понадобится просто обновить директиву `<%@ Page %>`. При использовании Visual Studio средство IntelliSense отобразит все доступные темы, определенные в папке `App_Themes`.

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" Theme ="CrazyOrange" %>
```

Из-за того, что данной странице назначена тема `CrazyOrange`, а в файле `Web.config` указана тема `BasicGreen`, все страницы *кроме этой* будут визуализированы с применением темы `BasicGreen`.

Свойство `SkinID`

Временами может возникнуть необходимость в определении набора возможных внешних видов и линий поведения для одиночного виджета. Например, предположим, что требуется определить два пользовательских интерфейса для типа `Button` внутри темы `CrazyOrange`. Различать варианты внешнего вида и поведения можно с использованием свойства `SkinID` элемента управления внутри файла `.skin`:

```
<asp:Button runat="server" BackColor="#FF8000"/>
<asp:Button runat="server" SkinID = "BigFontButton"
          Font-Size="30pt" BackColor="#FF8000"/>
```

Теперь при наличии страницы, которая работает с темой `CrazyOrange`, каждому элементу `Button` по умолчанию будет назначена неименованная обложка `Button`. Если необходимо, чтобы в файле `.aspx` обложку `BigFontButton` получило несколько разных кнопок, тогда следует просто указать свойство `SkinID` внутри разметки:

```
<asp:Button ID="Button2" runat="server"
          SkinID="BigFontButton" Text="Button" /><br />
```

Назначение тем программным образом

И последнее, но не менее важное: темы допускается назначать в коде. Это может пригодиться, когда конечным пользователям должен быть предоставлен способ выбора темы для текущего сеанса. Разумеется, пока еще не было показано, каким образом строить веб-приложения с поддержкой состояния, а потому выбранная тема будет утеряна между обратными отправками. На сайте производственного уровня выбранная в текущий момент тема пользователя сохраняется внутри переменной сеанса или же записывается в базу данных.

Чтобы продемонстрировать назначение темы программным образом, добавим к пользовательскому интерфейсу в файле `Default.aspx` три новых элемента управления `Button` (рис. Г.29). Затем для каждого элемента `Button` обработаем событие `Click`.

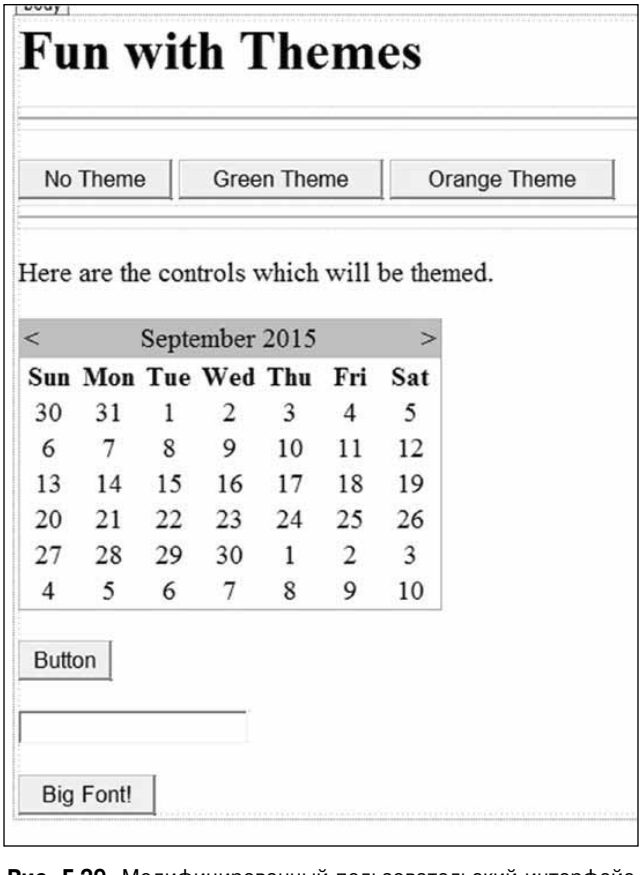

Рис. Г.29. Модифицированный пользовательский интерфейс примера работы с темами

Имейте в виду, что назначать тему в коде можно только на определенных фазах жизненного цикла страницы. Обычно это делается внутри обработчика события Page_PreInit. С учетом сказанного модифицируем файл кода, как показано ниже:

```
partial class _Default : System.Web.UI.Page
{
  protected void btnNoTheme_Click(object sender, System.EventArgs e)
  {
    // Пустая строка означает, что никакая тема не применена.
    Session["UserTheme"] = "";
    // Снова инициировать событие PreInit.
    Server.Transfer(Request.FilePath);
  }
  protected void btnGreenTheme_Click(object sender, System.EventArgs e)
  {
    Session["UserTheme"] = "BasicGreen";
    // Снова инициировать событие PreInit.
    Server.Transfer(Request.FilePath);
  }
  protected void btnOrangeTheme_Click(object sender, System.EventArgs e)
  {
    Session["UserTheme"] = "CrazyOrange";
    // Снова инициировать событие PreInit.
    Server.Transfer(Request.FilePath);
  }
  protected void Page_PreInit(object sender, System.EventArgs e)
  {
    try
    {
      Theme = Session["UserTheme"].ToString();
    }
    catch
    {
      Theme = "";
    }
  }
}
```

Обратите внимание, что выбранная тема сохраняется в переменной сеанса (детали ищите в приложении Д) по имени UserTheme, значение которой формально присваивается внутри обработчика событий Page_PreInit(). Когда пользователь щелкает на заданном элементе управления Button, программно инициируется событие PreInit путем вызова метода Server.Transfer() и запрашивания текущей страницы еще раз. Загрузив страницу в браузер, вы обнаружите, что темы можно устанавливать посредством щелчков на разных элементах Button.

Исходный код. Веб-сайт FunWithThemes доступен в подкаталоге Appendix_D.

Резюме

В настоящем приложении было показано, как использовать разнообразные элементы управления Web Forms. Сначала вы изучили роль базовых классов `Control` и `WebControl`, а затем узнали, каким образом динамически взаимодействовать с внутренней коллекцией элементов управления панели. Попутно вы ознакомились с новой моделью навигации по сайту (файлы `.sitemap` и компонент `SiteMapDataSource`), новым механизмом привязки данных и различными элементами управления проверкой достоверности.

Вторая половина приложения была посвящена обсуждению роли мастер-страниц и тем. Мастер-страницы могут быть задействованы с целью определения общей компоновки для набора страниц сайта. В файле `.master` определяется любое количество мест заполнения, куда страницы содержимого подключают свое специальное содержимое пользовательского интерфейса. Наконец, вы узнали, что механизм тем Web Forms позволяет декларативно или программно применять общий внешний вид и поведение пользовательского интерфейса к виджетам на веб-сервере.