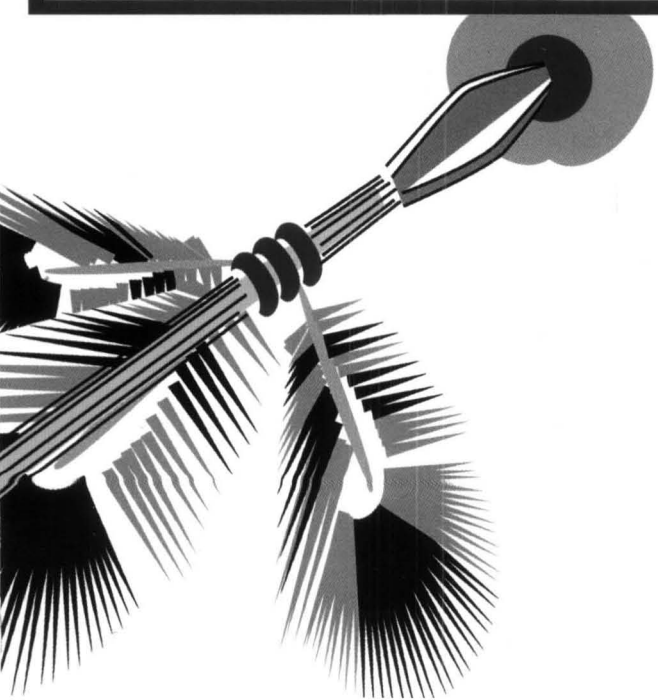
Байдачный С. С.

«СОЛОН»

Silverlight 4: Создание насыщенных Web-приложений

- Создание Web-приложений нового поколения
- Взаимодействие с объектной моделью браузера
- Управление шаблонами и стилями
- Использование графики и анимации
- Создание собственных элементов управления
- Взаимодействие с данными
- Использование медиа-элементов и Deep Zoom
- Отладка и тестирование Silverlight-приложений
- Интеграция с SharePoint 2010



Библиотека Профессионала

УДК 621.396.218
ББК 32.884.1
Б18

Байдачный С. С.
Б18 Silverlight 4: Создание насыщенных Web-приложений — М.: СОЛОН-ПРЕСС, 2010. — 288 с.: ил. — (Серия «Библиотека профессионала»).

ISBN 978-5-91359-079-4

Silverlight 4 — новая технология от Microsoft, предназначенная для разработки насыщенных Web-приложений, или приложений с «богатым» интерфейсом. Основные характеристики Silverlight-приложений — это интенсивное использование графики, анимации, работа с медиа-файлами, а также эффективное взаимодействие с данными и серверными компонентами. При этом разработчик имеет возможность не только использовать управляемые языки программирования (C#, VB.NET) для разработки Silverlight-приложений, но и получить доступ к большинству преимуществ, доступных в .NET Framework. Если взять во внимание, что процесс разработки Silverlight-приложений тесно интегрирован в Visual Studio, то можно утверждать, что использование Silverlight не вызовет затруднений у существующих .NET разработчиков.

Данная книга может быть полезна для всех, кто решил изучить Silverlight 4 и уже имеет общие познания в разработке приложений на платформе .NET.

УДК 621.396.218
ББК 32.884.1

По вопросам приобретения обращаться:
Тел: (495) 254-44-10, (499) 795-73-26

Сайт издательства СОЛОН-ПРЕСС: **www.solon-press.ru**
E-mail: **kniga@coba.ru**

ISBN 978-5-91359-079-4

От автора

В ноябре 2009 года мне довелось побывать на Professional Developers Conference (PDC) 09, где, среди прочих анонсов, была представлена четвертая версия технологии Silverlight. Параллельно с анонсом Silverlight 4 были опубликованы новые данные, показывающие доступность предыдущих версий Silverlight на компьютерах пользователей — этот показатель составил 45%.

Сегодня, в начале февраля 2010 года, заканчивая работу над этой книгой, я обратился на сайт http://riastats.com, чтобы получить последние данные по доступности Silverlight. Этот показатель составил более 50%. Таким образом, чуть более чем за два года (Silverlight 1 появился в сентябре 2007 года) Silverlight завоевал более половины компьютеров по всему миру.

Если говорить о том, что меня привлекает в Silverlight, то можно выделить следующие преимущества:

- интеграция Silverlight в существующую платформу. Разработка Silverlight-приложений возможна на управляемом языке программирования (C# или VB.NET), при этом Silverlight поддерживает основные компоненты, доступные в .NET Framework и позволяет использовать те же подходы, что при разработке Windows- или ASP.NET-приложений. Это очень важный факт, так как в отличие от PHP- или Java-разработчика, разработчик на платформе .NET способен создавать приложения любого типа (разве что не драйвера);
- возможность создания корпоративных приложений. Несмотря на небольшой размер встраиваемого компонента Silverlight (чуть более 4 Мб), сюда входит достаточно большой набор элементов управления, позволяющий создавать интерфейсы любой сложности. Помимо базовых элементов, в Silverlight доступен дополнительный набор в пакете SDK и Silverlight Toolkit, куда входят такие элементы как Grid, элементы для построения графиков и диаграмм, механизмы навигации внутри приложения. И, наконец, Silverlight поддерживает работу приложений вне браузера, где есть возможность не только установить приложение на компьютер пользователя, но и предоставить ему дополнительные права, позволяющие взаимодействовать с COM API. Все это позволяет приблизить Silverlight-приложения по функциональности к Windows-приложениям;
- поддержка Silverlight в средствах разработки. Если Вы разрабатываете Silverlight-приложения, то можете использовать среду разработки Visual Studio 2010, которая используется для разработки всех типов приложений на платформе .NET. В Visual Studio 2010 присутствуют не только шаблоны соответствующих проектов, но и визуальный редактор, позволяющий создавать интерфейсы с использованием Drag & Drop. В свою очередь, если в Вашем приложении необходима помощь дизайнера, то тут можно

использовать новое средство разработки интерфейсов Microsoft Expression Blend. Этот продукт работает с проектами в формате Visual Studio и ориентирован на визуальное создание интерфейсов. Фактически, это первый продукт, который позволил построить «мост» между дизайнерами и разработчиками.

Поскольку я являюсь разработчиком, то в этой книге речь пойдет в основном о создании приложений средствами Visual Studio, опуская возможности редактора Blend. Изучение Silverlight 4 мы начнем с новых возможностей этой технологии, после чего перейдем к архитектуре и работе с элементами управления. Несмотря на то, что большая часть книги посвящена корпоративным приложениям, мы затронем возможности, связанные с графикой и анимацией. Закончим мы наш обзор вопросами отладки, тестирования и повышения производительности Silverlight-приложений. Поэтому, смею полагать, что данная книга содержит всю необходимую информацию для разработчика и позволит приступить к созданию даже самых сложных приложений.

Глава 1

ВВЕДЕНИЕ В SILVERLIGHT 4

Наверняка, если Вы уже сталкивались с Silverlight, то интересуетесь новыми возможностями, которые появились в четвертой версии. Поэтому, первая глава посвящена обзору Silverlight 4. Тут описаны практически все новые возможности технологии, с которыми мы будем встречаться по ходу книги.

Если Вы еще не сталкивались с Silverlight, то переходите сразу ко второй главе.

Поддержка Drag&Drop

Обзор новых возможностей начнем с простого, но приятного нововведения в Silverlight 4 — поддержки функциональности Drag&Drop. Естественно, речь идет не о том, как перемещать элементы внутри самого Silverlight-приложения (это легко сделать и в первой версии), а как передать приложению внешний объект, находящийся на машине пользователя.

В качестве примера использования Drag&Drop в вебе можно рассмотреть сценарий, когда с помощью Silverlight-приложения пользователь выполняет загрузку файла на сервер: добавляет фотографию в свой профиль; загружает файл в свое хранилище на Skydrive и т. д. Все эти сценарии можно реализовать с помощью дополнительного диалогового окна, но Drag&Drop позволит добавить некоторую «изюминку» вашему интерфейсу.

Чтобы продемонстрировать использование Drag&Drop, разработаем приложение, в окно которого можно перетянуть несколько изображений (в формате, поддерживаемом Silverlight) и отобразить их.

Для реализации Drag&Drop механизма в Silverlight 4 присутствует очень полезное свойство **AllowDrop**, которое доступно у любого из наследников **UIElement**, то есть у любого из визуальных элементов, включая элементы компоновки и графические примитивы. Это означает, что можно реализовать механизм «перетаскивания» внешнего объекта на любой из визуальных элементов.

Устанавливая свойство **AllowDrop** в значение True, разработчик инициирует работу четырех событий, которые можно использовать для реализации Drag&Drop функциональности: **Drop**, **DragEnter**, **DragLeave**, **DragOver**. Основным событием является **Drop**, именно оно генерируется при завершении операции перетаскивания объекта. Остальные три события генерируются при перемещении курсора мыши, захватившего объект, но до завершения операции

перетаскивания, то есть эти события являются вспомогательными и могут быть использованы для дополнительной визуализации процесса перетаскивания в интерфейсе (например, выделение цветом областей, доступных для освобождения объекта).

Реализуем простой интерфейс, содержащий кнопку и панель типа **Canvas**. Объект на основе **Canvas** подходит для наших целей лучше всего, так как позволяет привязать отображаемое изображение к координатам курсора мыши.

```
<UserControl x:Class="DragAndDrop_Chapter0.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid x:Name="LayoutRoot" Background="White">
        <Grid.RowDefinitions>
            <RowDefinition Height="50"></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Button x:Name="printButton" Content="Print" Width="100">
        </Button>
        <Border Grid.Row="1" x:Name="photoBorder"
            BorderBrush="Blue" BorderThickness="4">
            <Canvas x:Name="photoPanel" Grid.Row="1"
                    Background="Gray" AllowDrop="True"
                    Drop="photoPanel_Drop"
                    DragEnter="photoPanel_DragEnter"
                    DragLeave="photoPanel_DragLeave">
            </Canvas>
        </Border>
    </Grid>
</UserControl>
```

В приведенном выше коде панель photoPanel будет использоваться для отображения изображений, перетянутых с компьютера пользователя. Тут же мы определили свойство **AllowDrop**, установленное в True, и указали обработчики событий. Кнопка printButton нам понадобится в следующем разделе.

Ниже показана реализация описанных выше обработчиков событий **Drop**, **DragEnter**, **DragLeave**.

```
private void photoPanel_Drop(object sender, DragEventArgs e)
{
    photoBorder.BorderBrush = new SolidColorBrush(Colors.Blue);

    IDataObject data = e.Data;
    FileInfo[] files = (FileInfo [])data.GetData
        (DataFormats.FileDrop);

    int i=0;
    foreach (FileInfo f in files)
    {
        Stream s=f.OpenRead();
        BitmapImage bitmap=new BitmapImage();
        bitmap.SetSource(s);

        Image img = new Image();
```

```
            img.Source = bitmap;

            img.Width = 100;
            img.Height = 80;

            Point p = e.GetPosition(photoPanel);
            img.Margin = new Thickness(p.X+10*i, p.Y+8*i, 0, 0);
            i++;

            photoPanel.Children.Add(img);
        }
    }

    private void photoPanel_DragEnter(object sender, DragEventArgs e)
    {
        photoBorder.BorderBrush = new SolidColorBrush(Colors.Red);
    }

    private void photoPanel_DragLeave(object sender, DragEventArgs e)
    {
        photoBorder.BorderBrush = new SolidColorBrush(Colors.Blue);
    }
```

Последние два события мы обрабатываем лишь с целью выделить область, доступную для размещения перетаскиваемого объекта (тут нужно помнить, что при генерации события **Drop**, событие **DragLeave** уже не генерируется, поэтому нужно снять выделение). А вот основную работу делает обработчик события **Drop**: получение информации о передаваемых файлах, открытие

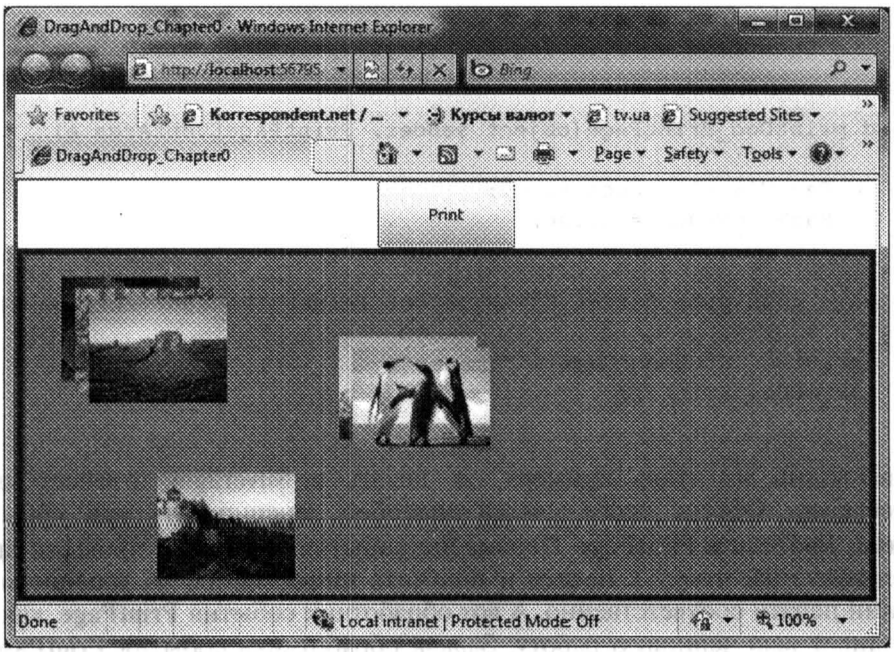


Рис. 1.1. Результат работы приложения

потока, загрузка и отображение изображения с помощью Pixel API (одно из новшеств Silverlight 3).

Чтобы сделать код более читабельным, мы не обрабатываем ошибки, связанные с типом файлов, но в реальной задаче это необходимо сделать обязательно (попробуйте в этом примере перетащить текстовый файл или офисный документ и Вы получите ошибку).

Чтобы продемонстрировать работу нашего приложения, выберите одно или несколько изображений на вашем компьютере и попробуйте перетащить его в окно приложения. Уменьшенное изображение отобразится по текущим координатам (рис. 1.1).

Печать из Silverlight-приложений

Одна из важных и фундаментальных возможностей, которая должна значительно расширить возможности Silverlight-приложений, это печать.

Для демонстрации возможности печати будем использовать предыдущий пример, определив обработчик события **Click** для кнопки printButton, объект типа **PrintDocument** и обработчик события **PrintPage**. Вот как будет выглядеть наш код:

```
private PrintDocument printDoc=new PrintDocument();

public MainPage()
{
    InitializeComponent();

    printDoc.PrintPage += new
        EventHandler<PrintPageEventArgs>(printDoc_PrintPage);
}

void printDoc_PrintPage(object sender, PrintPageEventArgs e)
{
    e.PageVisual = photoPanel;
    e.HasMorePages = false;
}

private void printButton_Click(object sender, RoutedEventArgs e)
{
    printDoc.DocumentName = "Images";
    printDoc.Print();
}
```

Как видно из этого примера, за печать в Silverlight отвечает класс **PrintDocument**. Объект этого класса способен генерировать три события **StartPrint**, **EndPrint** и **PrintPage**. Первые два события позволяют провести предварительную подготовку к печати и получить информацию об успешном завершении печати соответственно. А вот обработчик события **PrintPage**, как раз и выполняет всю черновую работу. Задача обработчика события **PrintPage** – получить параметры страницы (тут только длина и ширина отображаемой об-

ласти), используя параметр типа **PrintPageEventArgs**, сформировать содержимое страницы и отправить ее на печать.

Фактически, объект типа **PrintDocument** способен печатать любой элемент, порожденный от **UIElement**, то есть, для печати страницы Вы можете выбрать любой из контейнеров, расположить в нем свои элементы и передать ссылку на этот контейнер с помощью свойства **PageVisual**. После этого **UIElement** преобразуется в картинку, и происходит его печать.

В примере выше мы не формировали специальную страницу для печати, а просто использовали существующий контейнер, отображающий наши картинки.

В завершение хочу отметить, что при реализации печати необходимо помнить о важном свойстве **HasMorePages**. Именно благодаря этому свойству у программиста появляется возможность печатать не одну, а несколько страниц. Как только печать закончена, свойство нужно установить в **false**.

Обработка нажатия правой кнопки мыши

Еше с выходом Silverlight 1, многие разработчики начали жаловаться на то, что не могут реализовать собственное контекстное меню при нажатии правой кнопки мыши в своем приложении. Действительно, правая кнопка полностью принадлежала Silverlight, а пользователь мог вызвать только контекстное меню, позволяющее получить доступ к настройкам встраиваемого компонента.

Silverlight 4 позволяет полностью переопределить поведение при нажатии правой кнопки мыши. При этом программист вовсе не обязан отображать меню. Действие может полностью зависеть от самого приложения.

Чтобы переопределить работу правой кнопки мыши, достаточно выполнить два действия:

1. Отключить существующее меню. Для этого нужно определить обработчик с события **MouseRightButtonDown** и установить свойство **Handled** в true. Это сигнализирует о том, что мы берем обработку события на себя и стандартное меню отображать не нужно;

2. Определить вызов собственного меню (или любые другие действия) в обработчике события **MouseRightButtonUp**.

Расширим функционал предыдущего приложения, добавив контекстное меню к изображениям на панели. Для этого расширим метод photoPanel_Drop, добавив следующий код:

```
img.MouseRightButtonDown += new
   MouseButtonEventHandler(img_MouseRightButtonDown);
img.MouseRightButtonUp += new
   MouseButtonEventHandler(img_MouseRightButtonUp);
```

Теперь реализуем сами обработчики событий. Начнем с **MouseRightButtonDown**:

```
void img_MouseRightButtonDown(object sender, MouseButtonEventArgs e)
{
    e.Handled = true;
```

Как видно, тут нет ничего сложного. Мы просто нотифицируем Silverlight о перехвате события правой кнопки и подавляем отображение стандартного контекстного меню.

После этого реализуем **MouseRightButtonUp**. Тут есть проблема, которая состоит в том, что если Вы хотите отобразить меню, то его нужно реализовать самостоятельно, или обратиться к сторонним разработчикам за дополнительными компонентами. В списке стандартных компонент и в SDK контекстное меню пока не реализовано. Я не стал разрабатывать меню сам, а установил тестовую версию с сайта: http://www.telerik.com. Тут Вы можете найти множество элементов управления для Silverlight 4.

Замечание. Никогда не используйте контекстное меню от Telerik так, как показано ниже. Компонент достаточно «умный», чтобы отобразить меню в качестве реакции на какое-то событие. Кроме того, этот элемент можно ассоциировать с любым интерфейсным элементом в момент его создания в XAML. Но, если сделать все правильно, то обработчик события для правой кнопки мыши не понадобится ☺. Поэтому будем создавать меню именно при нажатии правой кнопки мыши и уничтожать его в «ручном» режиме.

```
private Image menuImage;
private RadContextMenu contextMenu;

void img_MouseRightButtonUp(object sender, MouseButtonEventArgs e)
{
    menuImage = (Image)sender;
    if (contextMenu != null)
    {
        photoPanel.Children.Remove(contextMenu);
    }

    contextMenu = new RadContextMenu();
    contextMenu.Items.Add("Delete");
    contextMenu.Items.Add("Close Menu");

    Point p=e.GetPosition(photoPanel);

    contextMenu.Margin = new Thickness(p.X, p.Y, 0, 0);
    photoPanel.Children.Add(contextMenu);

    contextMenu.ItemClick += new
  Telerik.Windows.RadRoutedEventHandler(menu_ItemClick);
}

void menu_ItemClick(object sender,
  Telerik.Windows.RadRoutedEventArgs e)
{
    RadRoutedEventArgs args = e as RadRoutedEventArgs;
    RadMenuItem menuItem = args.OriginalSource as RadMenuItem;
    string tag = Convert.ToString(menuItem.Header);
    switch (tag)
    {
        case "Delete":
            photoPanel.Children.Remove(menuImage);
            photoPanel.Children.Remove(contextMenu);
```

```
                break;
            case "Close Menu":
                photoPanel.Children.Remove(contextMenu);
                break;
        }

        contextMenu = null;
    }
```

В коде выше я реализовал механизм создания нового контекстного меню, а также механизм обработки события, связанного с выбором пункта меню Частично продублировал работу, уже реализованную в компоненте.

Ниже показан снимок экрана во время работы приложения.

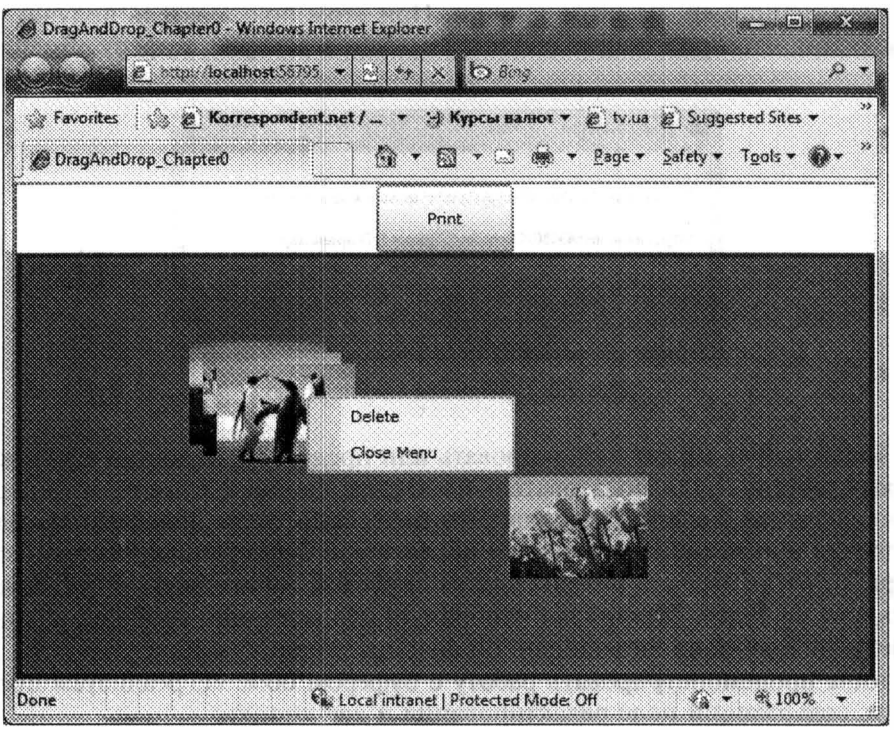


Рис. 1.2. Результат работы приложения

Работа с буфером обмена

Следующая возможность позволяет получить доступ к буферу обмена Пока речь идет только о работе с текстом, но в будущем (я пока использую бета-версию) эта функциональность будет расширена, и пользователь сможет вставлять и копировать изображения.

Механизм работы с буфером реализован в двух вариантах:

- копирование и вставка текста с помощью горячих клавиш (Ctrl+C Ctrl+V) при работе с такими элементами как **TextBox**, **RichTextBox** и т. д. Как реализован этот механизм, и как воссоздать его в своем эле-

менте управления — пока остается загадкой. Ведь Ctrl+C, Ctrl+V работают тут без каких-либо разрешений от пользователя, то есть работают всегда, в отличии от второго механизма;

- доступ к буферу обмена с помощью статического класса **Clipboard**, содержащего три статических метода:
- **ContainsText** — проверяет наличие текста в буфере обмена;
- **GetText** — позволяет получить текст из буфера обмена;
- **SetText** — сохраняет текст в буфере обмена.

Рассмотрим второй механизм более детально, так как только он предоставляет возможности для разработчика.

Итак, первое, с чем Вы встретитесь при использовании класса **Clipboard**, — это доступность его методов только в ответ на событие инициированное пользователем, например, нажатие кнопки. При этом пользователь получит сообщение о том, что приложение пытается получить доступ к буферу обмена:

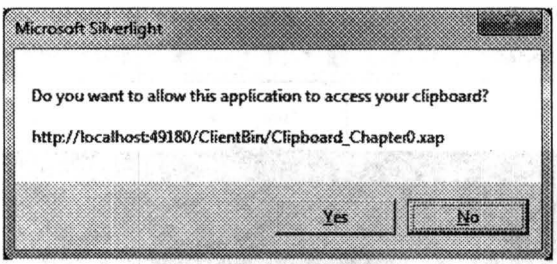


Рис. 1.3. Сообщение пользователю

Пользователь может заблокировать или предоставить доступ приложения к буферу обмена. Если пользователь блокирует доступ к буферу, то генерируется исключение **SecurityException**. Что интересно, если пользователь блокирует доступ при первой попытке, то исключение будет генерироваться всякий раз при попытке вызвать методы класса **Clipboard**, но сообщение пользователь видеть уже не будет. Механизма отобразить пользователю сообщение еще раз я не нашел. Еще одна загадка данной функциональности.

Чтобы продемонстрировать работу с буфером обмена, реализуем простой пример, содержащий поле редактирования и две кнопки:

MainPage.xaml

```
<UserControl x:Class="Clipboard_Chapter0.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
>
    <StackPanel x:Name="LayoutRoot" Background="White">
        <TextBox Height="92" Name="textBox1" Width="339"
         AcceptsReturn="True"
         HorizontalScrollBarVisibility="Auto" />
        <Button Content="Copy" Height="23"
         Name="copyButton" Width="75"
         Click="copy_Click" />
```

```
        <Button Content="Past" Height="23"
         Name="pasteButton" Width="75" Click="past_Click" />
    </StackPanel>
</UserControl>
```

MainPage.xaml.cs

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Security;

namespace Clipboard_Chapter0
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();
        }

        private void past_Click(object sender, RoutedEventArgs e)
        {
            try
            {
                if (Clipboard.ContainsText())
                    textBox1.Text = Clipboard.GetText();
            }
            catch (SecurityException ex)
            {
                textBox1.Text = "You didn't grant permissions!";
            }
        }

        private void copy_Click(object sender, RoutedEventArgs e)
        {
            try
            {
                Clipboard.SetText(textBox1.Text);
            }
            catch (SecurityException ex)
            {
                textBox1.Text = "You didn't grant permissions!";
            }
        }
    }
}
```

Если Вы запустите этот пример, то на экране появится следующее окно позволяющее работать с текстом, копируя и вставляя содержимое из буфера обмена (рис. 1.4).

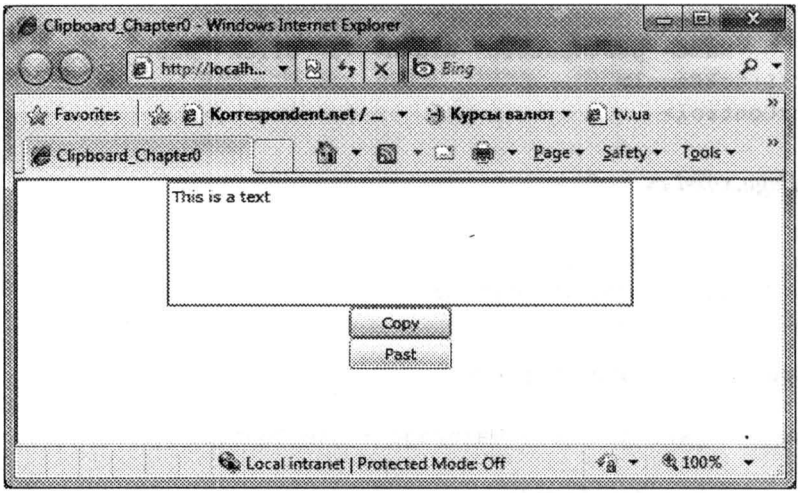


Рис. 1.4. Результат работы приложения

Элементы управления WebBrowser и HtmlBrush

Когда я услышал об очередной возможности Silverlight — отображение HTML, то, конечно же, сильно обрадовался. Ведь это позволило бы разработчикам использовать уже наработанные XML, вместе с XSLT-преобразованием, для отображения отформатированного содержимого (говорят, что слово «контент» — это уже русское слово, поэтому дальше буду использовать его). Однако, все оказалось немного более печально, чем в моих ожиданиях.

Возможность отображения HTML действительно появилась, но работает только для **приложений вне браузера** (Out Of Browser). И это очень странно. Ведь если речь идет об Internet-приложениях, то вряд ли конечный пользователь будет что-то устанавливать себе на машину, не поработав предварительно с приложением в окне браузера. Корпоративная сеть — другое дело, но тут можно использовать и технологию WPF (а то и Win Forms), которая к тому же и быстрее. А учитывая способ инсталляции приложений вне браузера (ведь их нельзя отправить по электронной почте или указать ссылку на инсталляцию) разработчики получают дополнительную головную боль при разработке «двух» версий одного приложения (в браузере и вне браузера). Фактически речь будет идти об одной версии, но с различным поведением, в зависимости от способа запуска.

Из сказанного выше, я делаю вывод, что с помощью этой функциональности Silverlight 4 пытаются позиционировать, как технологию, обладающую рядом преимуществ при создании приложений, работающих в корпоративной сети. Ниже мы рассмотрим еще несколько аналогичных возможностей, которые ставят Silverlight 4 на одну ступень с WPF, но только для приложений, работающих вне браузера.

Как бы то ни было, но если Вы разрабатываете приложение, работающее вне браузера, то можете использовать элемент управления **WebBrowser**, распо-

ложенный в стандартной сборке **System.Windows.dll**. Удивительно, но этот элемент входит в стандартный набор и не требует дополнительных сборок.

При создании элемента **WebBrowser** необходимо явно установить свойства **Height** и **Width**, так как по умолчанию они установлены в 0. Также важным свойством является **Source**, позволяющее установить адрес страницы. Чаще всего это свойство используется для установки начального значения в XAML-файле, но может быть полезно и для получения текущего адреса.

Особое внимание нужно обратить на то, что страница, отображаемая в **WebBrowser**, **должна находиться в домене** Silverlight-приложения (даже после установки на компьютер пользователя ощущается боязнь DOS-атак у разработчиков этой функциональности). Для отображения HTML-контента в процессе работы приложения, можно использовать один из двух методов:

- **Navigate** — действие этого метода аналогично установке свойства **Source**, то есть элемент **WebBrowser** отобразит страницу, если она находится в домене приложения. Если Вы хотите отобразить внешний контент, то необходимо разместить ссылку на него внутри элемента **iframe**;
- **NavigateToString** — этот метод более универсален и позволяет отобразить любой HTML-контент, полученный в качестве параметра. Это.означает, что если Вы хотите отобразить внешний контент, то просто формируете строку, содержащую **iframe**, и указываете ссылку на необходимую страницу (независимо от домена).

Ниже показан пример приложения, которое позволяет осуществлять навигацию на любую страницу из Silverlight-приложения:

```
<UserControl x:Class="HTML_Chapter0.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
>
    <StackPanel x:Name="LayoutRoot" Background="White">
        <StackPanel Orientation="Horizontal"
            HorizontalAlignment="Center">
            <TextBox Name="urlText"
              Text="http://www.microsoft.com" Width="300">
            </TextBox>
            <Button Content="Navigate"
Click="Button_Click"></Button>
        </StackPanel>
        <WebBrowser Height="800" Width="1024" Name="webBrowser">
        </WebBrowser>
    </StackPanel>
</UserControl>
```

А вот и код, осуществляющий навигацию:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    webBrowser.NavigateToString(
        String.Format(
        "<iframe height=800 width=1024 src={0}></iframe>",
        urlText.Text));
```

При этом не забудьте сконфигурировать приложение как работающее вне браузера (и установить его после запуска на локальную машину).

Результат работы приложения Вы можете увидеть ниже:

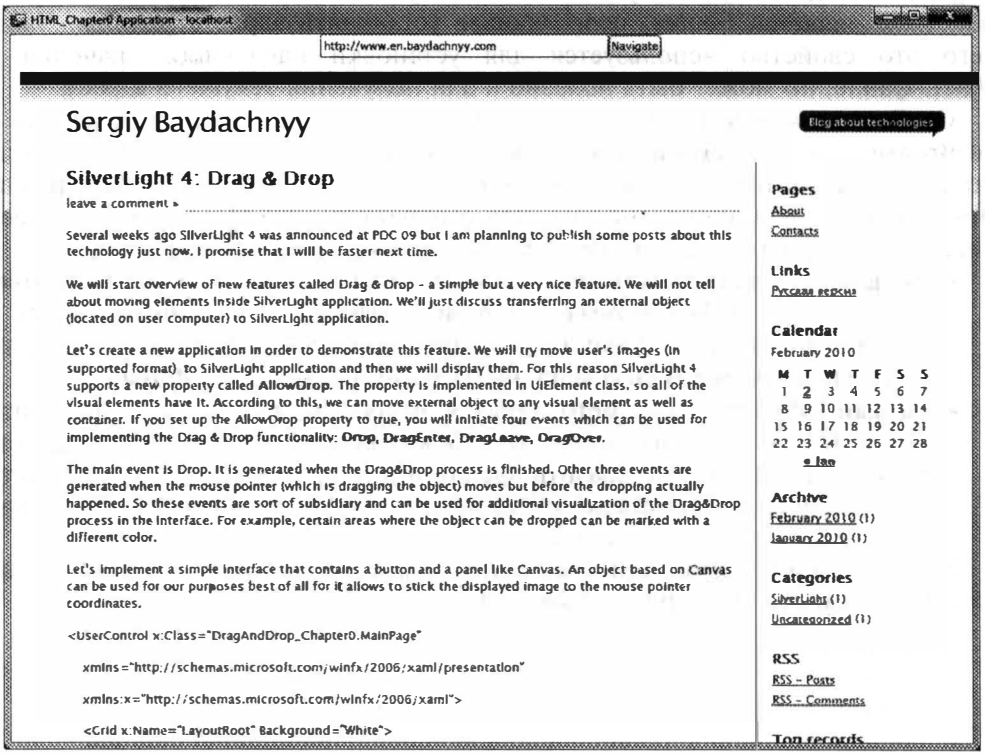


Рис. 1.5. Результат работы приложения

Использование **iframe** позволяет не только отобразить HTML-страницу, которая находится вне домена приложения, но и внедрить элемент **<object>**. Это автоматически означает, что Silverlight-приложения с легкостью могут содержать в себе Flash-приложения (например, отображать ролики с youtube).

Нужно также отметить и возможность принудительно выполнять Java-скрипты, содержащиеся в **WebBrowser** элементе. Для этого используется метод **InvokeScript**.

В завершении раздела отмечу, что HTML можно использовать в качестве заливки для любого элемента, поддерживающего работу с **Brush**. Для этих целей существует специальный класс **HtmlBrush**, который в качестве источника принимает объект **WebBrowser**.

RichTextArea элемент управления

Описывая очередную возможность Silverlight 4, почему-то вспомнил лозунг: «Перед использованием — доработать напильником». Действительно если смотреть на предыдущие разделы, то вроде бы новый функционал и есть

но чего-то все время не хватает. Есть возможность определить контекстное меню, но инфраструктуры для такого меню нет (хотя инфраструктура есть даже для подсказок), есть возможность работать с буфером обмена, но с жесткими ограничениями, есть возможность отображать HTML, но только для приложений, работающих вне браузера. Не исключением является и новый элемент управления, который сразу попал в стандартную поставку Silverlight 4, — это **RichTextArea**.

Уже пару лет разработчики пытаются добиться элемента управления, который бы позволил отображать форматированный текст. Пусть это будет DOCX, RTF, XPS или PDF, что не столь принципиально. Большинство текстовых редакторов спокойно преобразуют текст из одного формата в другой. Вместо ожидаемого элемента, разработчикам предоставили **RichTextArea**, который действительно способен отображать текст с минимальным форматированием, но в своем, только ему понятном, формате. Давайте посмотрим в каком именно.

Ниже пример элемента RichTextArea, отображающего текст с минимальный форматированием:

```
<RichTextArea HorizontalAlignment="Left"
    Name="rArea" VerticalAlignment="Top"
    Height="300" Width="400" >
    <Paragraph>
        <Bold>This is a bold text</Bold>
        <LineBreak></LineBreak>
        <Underline>This is an underline text</Underline>
        <LineBreak></LineBreak>
        <Italic>This is an italic text</Italic>
        <LineBreak></LineBreak>
        This is a button:
        <InlineUIContainer>
            <Button Content="Button"></Button>
        </InlineUIContainer>
    </Paragraph>
</RichTextArea>
```

В результате на экране отобразится следующий контент (рис. 1.6).

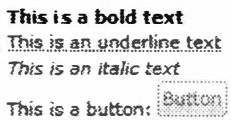


Рис. 1.6

Как видно, **RichTextArea** является контейнером для других элементов. Так, основным наполнением элемента **RichTextArea** выступает набор элементов **Paragraph**. Эти элементы описывают параграфы текстового документа. В свою очередь, параграф может включать набор из следующих элементов:

- **Run** — задает обычный текст;
- **Span** — служит для группировки других элементов;
- **Bold** — определяет жирное начертание символов;

- **LineBreak** — переход на другую строку;
- **Italic** — определяет рукописное начертание символов;
- **Underline** — текст с подчеркиванием;
- **HyperLink** — создает гиперссылку, которая становится активной только в режиме ReadOnly элемента **RichTextArea**;
- **InlineUIContainer** — позволяет вставить в документ любой из элементов, порожденных от **UIElement**.

Естественно, создавать и заполнять **RichTextArea** можно и программно. Вот небольшой пример кода:

```
Bold b = new Bold();
b.Inlines.Add("This is a bold text");

Italic i = new Italic();
i.Inlines.Add("This is an italic text");

Underline u = new Underline();
u.Inlines.Add("This is an underlined text");

Paragraph myPar = new Paragraph();
myPar.Inlines.Add(b);
myPar.Inlines.Add(new LineBreak());
myPar.Inlines.Add(i);
myPar.Inlines.Add(new LineBreak());
myPar.Inlines.Add(u);

rArea.Blocks.Add(myPar);
```

Таким образом, **RichTextArea** является элементом с минимальным интерфейсом, где всю работу необходимо выполнить программисту. Конечно, Вы можете найти примеры готовых методов, позволяющих сохранить и загрузить содержимое **RichTextArea**, или реализацию удобного интерфейса пользователя (вот один из примеров: http://channel9.msdn.com/learn/courses/Silverlight4/RichTextEditor), но много нужно будет сделать самостоятельно.

Управление окном приложения

Еще одна возможность приложений, работающих вне браузера, — это поддержка объекта типа **Window**. С помощью класса **Window** разработчик способен управлять окном во время работы приложения.

Естественно, что объект типа **Window** создается «за сценой», а разработчик может получить доступ к нему с помощью свойства **MainWindow** объекта **Application**.

Среди основных свойств класса **Window** можно выделить следующие:

- **Height** — определяет высоту окна;
- **Width** — определяет длину окна;
- **Left** — задает отступ от левой границы экрана;
- **Top** — задает отступ от верхней границы экрана;
- **IsActive** — возвращает **true**, если окно активно в данный момент и находится в фокусе. В противном случае возвращает **false**;

- **TopMost** — если это свойство установлено в **true**, то окно всегда располагается поверх других окон (всегда находится на экране);
- **WindowState** — определяет состояние окна (Normal, Minimize, Maximize).

Среди методов можно выделить лишь **Activate**, который активирует окно приложения, выводя его на передний план и передавая ему фокус.

Стоит также отметить, что устанавливать свойства объекта, порожденного от **Window**, можно лишь в ответ на действия пользователя (нажатие кнопки и др.) либо в обработчике события **Startup** объекта **Application** (либо до Startup).

Silverlight 4 поддерживает специальный раздел в конфигурационном файле, позволяющий установить начальные параметры окна. Вот как может выглядеть часть конфигурационного файла с установленными параметрами:

```
<OutOfBrowserSettings>
  <OutOfBrowserSettings.WindowSettings>
    <WindowSettings Title="My Window"
      Left="100" Top="100"
      Height="300" Width="300" />
  </OutOfBrowserSettings.WindowSettings>
</OutOfBrowserSettings>
```

Поддержка уведомлений

Следующая возможность приложений, работающих вне браузера, — это способность отображать уведомления. В данном случае, под уведомлениями понимаются всплывающие окна, не требующие взаимодействия с пользователем (хотя они способны обрабатывать события от мыши, но не от клавиатуры) и исчезающие через заданный промежуток времени. Примеры таких окон можно найти в приложении Microsoft Outlook, которое отображает всплывающие сообщения при получении нового письма.

Чтобы отобразить уведомление, достаточно создать объект типа **NotificationWindow** и установить свойство **Content** у созданного объекта. Свойство **Content** принимает любой объект, порожденный от **FrameworkElement**, то есть любой визуальный элемент или элемент компоновки. После установки свойств **Content**, **Width**, **Height**, достаточно вызвать метод **Show**, принимающий время показа окна в миллисекундах.

Рассмотрим пример приложения, работающего вне браузера и отображающего примитивное окно в момент запуска. Интерфейс приложения реализуем следующим образом:

```
<UserControl x:Class="Notify_Chapter0.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid x:Name="LayoutRoot" Background="White">
        <StackPanel HorizontalAlignment="Center">
            <TextBlock Text="Notification Demo"
               TextAlignment="Center" FontSize="18">
```

```
            </TextBlock>
            <Button Name="btn" Visibility="Collapsed"
               Content="Инсталировать приложение" FontSize="18"
               Click="Button_Click"></Button>
            <TextBlock Name="txt" Visibility="Collapsed"
               TextWrapping="Wrap" TextAlignment="Center"
               Text="Приложение должно быть запущено вне браузера"
               FontSize="18">
            </TextBlock>
        </StackPanel>
    </Grid>
</UserControl>
```

Во время запуска приложения необходимо проверить, действительно ли приложение запущено вне браузера, и только затем приступать к созданию окна. Вот как это можно реализовать:

```
public MainPage()
{
    InitializeComponent();

    if (App.Current.IsRunningOutOfBrowser)
    {
        NotificationWindow notify = new NotificationWindow();

        StackPanel panel = new StackPanel();
        panel.Background = new SolidColorBrush(Colors.Gray);
        panel.Width = 250;
        panel.Height = 50;

        TextBlock header = new TextBlock();
        header.Text = "New message";
        header.FontWeight = FontWeights.Bold;

        TextBlock message = new TextBlock();
        message.Text = "This is a new message";

        panel.Children.Add(header);
        panel.Children.Add(message);

        notify.Content = panel;
        notify.Width = panel.Width;
        notify.Height = panel.Height;
        notify.Show(10000);
    }
    else
    {
        if (App.Current.InstallState == InstallState.Installed)
        {
            txt.Visibility = Visibility.Visible;
        }
        else
        {
            btn.Visibility = Visibility.Visible;
```

```
            }
        }
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        App.Current.Install();
    }
```

В результате после запуска приложения на экране можно увидеть следующую картину:

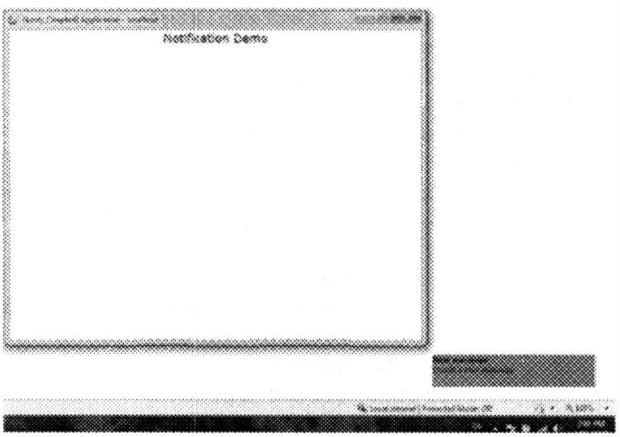


Рис. 1.7. Результат работы приложения

Как видно, уведомление отображается в системной области с небольшим отступом от нее (по 44 единицы).

Данная функциональность может быть полезна при обработке обновлений, коммуникациях, асинхронной обработке данных и др.

Поддержка микрофона и камеры

Еще одна потрясающая возможность Silverlight 4, — это поддержка камеры и микрофона. Чтобы показать эти возможности в действии, рассмотрим небольшой пример. Для этого создадим новый Silverlight-проект и реализуем следующий интерфейс:

```
<UserControl x:Class="WebCam_Chapter0.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <StackPanel x:Name="LayoutRoot" Background="White"
        Loaded="LayoutRoot_Loaded">
        <Rectangle Width="400" Height="300" Fill="Gray"
            Name="videoContainer">
        </Rectangle>
```

```
        <StackPanel Orientation="Horizontal"
HorizontalAlignment="Center">
            <StackPanel>
                <TextBlock Text="Video Devices"
Margin="5"></TextBlock>
                <ListBox Height="30" Name="videoBox">
                    <ListBox.ItemTemplate>
                        <DataTemplate>
                            <TextBlock
                             Text="{Binding FriendlyName}">
                            </TextBlock>
                        </DataTemplate>
                    </ListBox.ItemTemplate>
                </ListBox>
            </StackPanel>
            <StackPanel>
                <TextBlock Text="Audio Devices"
Margin="5"></TextBlock>
                <ListBox Height="30" Name="audioBox">
                    <ListBox.ItemTemplate>
                        <DataTemplate>
                            <TextBlock
                             Text="{Binding FriendlyName}">
                            </TextBlock>
                        </DataTemplate>
                    </ListBox.ItemTemplate>
                </ListBox>
            </StackPanel>
        </StackPanel>
        <StackPanel Orientation="Horizontal"
HorizontalAlignment="Center">
            <Button Content="Start" Width="100" Height="30"
                Click="Start_Click" Margin="5"></Button>
            <Button Content="Stop" Width="100" Height="30"
                Click="Stop_Click" Margin="5"></Button>
            <Button Content="Capture" Width="100" Height="3
                Margin="5" Click="Capture_Click"></Button>
        </StackPanel>
        <ScrollViewer Width="500" Height="200"
            HorizontalScrollBarVisibility="Visible"
            VerticalScrollBarVisibility="Hidden">
            <ItemsControl Name="imageContainer">
                <ItemsControl.ItemTemplate>
                    <DataTemplate>
                        <Image Source="{Binding}"
                         Stretch="UniformToFill"
                         Height="150" Margin="5">
                        </Image>
                    </DataTemplate>
                </ItemsControl.ItemTemplate>
                <ItemsControl.ItemsPanel>
                    <ItemsPanelTemplate>
                        <StackPanel Orientation="Horizontal
```

```
                        VerticalAlignment="Center"
                        HorizontalAlignment="Center">
                      </StackPanel>
                  </ItemsPanelTemplate>
              </ItemsControl.ItemsPanel>
          </ItemsControl>
      </ScrollViewer>
  </StackPanel>
</UserControl>
```

Предложенный интерфейс будет выглядеть так, как на рисунке ниже:

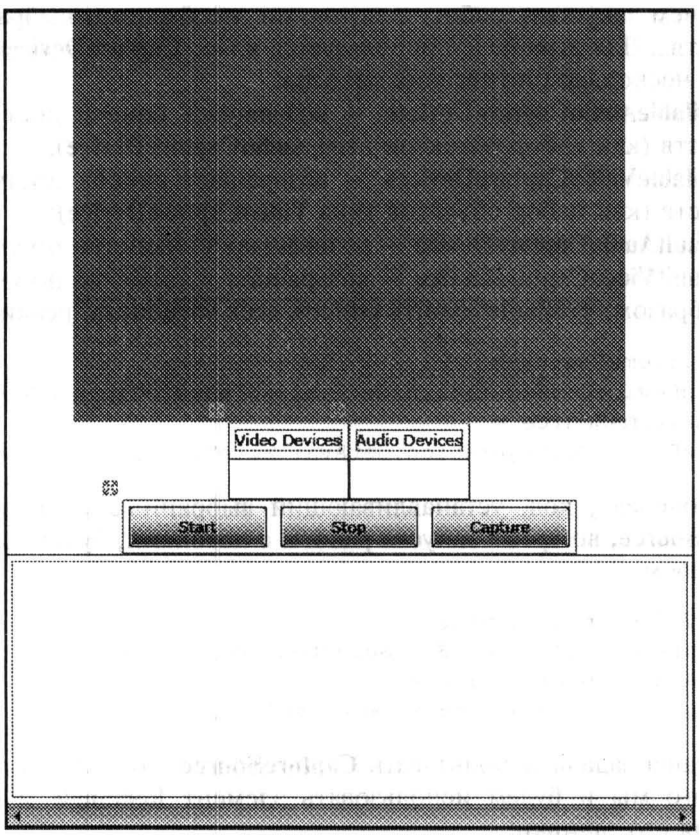


Рис. 1.8. Внешний вид интерфейса

Элемент **Rectangle** мы будем использовать для отображения видео, два элемента **ListBox** — для выдачи списка доступных устройств, а **ScrollView** элемент — для отображения изображений, захваченных во время отображения видео.

Итак, получить аудио и видео потоки можно с помощью класса **CaptureSource**. Структура этого класса достаточно простая. Так, тут можно выделить всего два основных метода и три основных свойства:

- **Start** — инициирует доступ объекта типа **CaptureSource** к аудио и видео потокам:

- **Stop** — закрывает доступ к видео и аудио потокам;
- **VideoCaptureDevice** — задает устройство для получения видео потока;
- **AudioCaptureDevice** — задает устройство для получения аудио потока;
- **State** — определяет состояние объекта типа **CaptureSource**: **Started** — запущен и получает потоки; **Stopped** — остановлен; **Failed** — проблемы с подключением.

Таким образом, чтобы приступить к работе с видео и аудио, необходимо создать объект типа **CaptureSource**:

```
private CaptureSource _source = new CaptureSource();
```

Прежде чем запустить работу с потоками, необходимо выбрать и установить устройства. Для этих целей используется класс **CaptureDeviceConfiguration**, содержащий несколько статических методов:

- **GetAvailableAudioCaptureDevices** — возвращает список доступных аудио устройств (как набор объектов типа **AudioCaptureDevice**);
- **GetAvailableVideoCaptureDevices** — возвращает список доступных видео устройств (как набор объектов типа **VideoCaptureDevice**);
- **GetDefaultAudioCaptureDevice** — возвращает устройство по умолчанию;
- **GetDefaultVideoCaptureDevice** — возвращает устройство по умолчанию.

Таким образом, чтобы получить список всех устройств, реализуем код:

```
audioBox.ItemsSource =
  CaptureDeviceConfiguration.GetAvailableAudioCaptureDevices();
videoBox.ItemsSource =
  CaptureDeviceConfiguration.GetAvailableVideoCaptureDevices();
```

В свою очередь, код, устанавливающий выбранные устройства объекту типа **CaptureSource**, во время запуска работы с потоками, будет выглядеть следующим образом:

```
_source.VideoCaptureDevice =
(VideoCaptureDevice)videoBox.SelectedItem;
_source.AudioCaptureDevice =
(AudioCaptureDevice)audioBox.SelectedItem;
```

Теперь наша задача использовать **CaptureSource** для вывода видео на экран. Для этого мы и будем использовать элемент **Rectangle** и объект типа **VideoBrush** для его заливки:

```
VideoBrush vBrush = new VideoBrush();
vBrush.SetSource(_source);

videoContainer.Fill = vBrush;
```

Создав объект типа **CaptureSource** и установив все необходимые параметры, можно приступать к вызову метода **Start**. Но, прежде чем сделать это, необходимо проверить, имеет ли приложение права на доступ к устройствам, а если не имеет, то права необходимо запросить у пользователя. Проверить наличие прав и запросить доступ можно с помощью уже знакомого класса **CaptureDeviceConfiguration**. Для этого используются свойство **AllowedDeviceAccess**

и метод **RequestDeviceAccess** соответственно. Таким образом, код, выполняющий запуск объекта типа **CaptureSource**, должен выглядеть следующим образом:

```
if (CaptureDeviceConfiguration.AllowedDeviceAccess
   || CaptureDeviceConfiguration.RequestDeviceAccess())
{
    _source.Start();
}
```

Выше были перечислены основные действия, которые необходимо проделать для отображения видео в Silverlight-приложении. Естественно, к данному коду необходимо добавить различные проверки и реализовать механизм захвата изображения. Детально я это расписывать не буду, а просто приведу готовый код:

```
private CaptureSource _source = new CaptureSource();
private ObservableCollection<WriteableBitmap> _images =
    new ObservableCollection<WriteableBitmap>();

public MainPage()
{
    InitializeComponent();
}

private void LayoutRoot_Loaded(object sender, RoutedEventArgs e)
{
    audioBox.ItemsSource =
      CaptureDeviceConfiguration.GetAvailableAudioCaptureDevices();
    videoBox.ItemsSource =
      CaptureDeviceConfiguration.GetAvailableVideoCaptureDevices();

    if ((videoBox.Items.Count > 0) && (audioBox.Items.Count > 0))
    {
        videoBox.SelectedIndex = 0;
        audioBox.SelectedIndex = 0;
    }

    imageContainer.ItemsSource = _images;
}

private void Stop_Click(object sender, RoutedEventArgs e)
{
    _source.Stop();
}

private void Start_Click(object sender, RoutedEventArgs e)
{
    if ((videoBox.Items.Count==0)||(audioBox.Items.Count==0))
    {
        MessageBox.Show("Audio or Video device is not available");
        return;
    }

    _source.VideoCaptureDevice =
(VideoCaptureDevice)videoBox.SelectedItem;
```

```
    _source.AudioCaptureDevice =
(AudioCaptureDevice)audioBox.SelectedItem;

    VideoBrush vBrush = new VideoBrush();
    vBrush.SetSource(_source);

    videoContainer.Fill = vBrush;

    if (CaptureDeviceConfiguration.AllowedDeviceAccess
       || CaptureDeviceConfiguration.RequestDeviceAccess())
    {
        _source.Start();
    }
}

private void Capture_Click(object sender, RoutedEventArgs e
{
    if (_source.State == CaptureState.Started)
    {
        _source.AsyncCaptureImage(CaptureImage<WriteableBit
    }
}

private void CaptureImage<WriteableBitmap>(
    System.Windows.Media.Imaging.WriteableBitmap t)
{
    _images.Add(t);
}
```

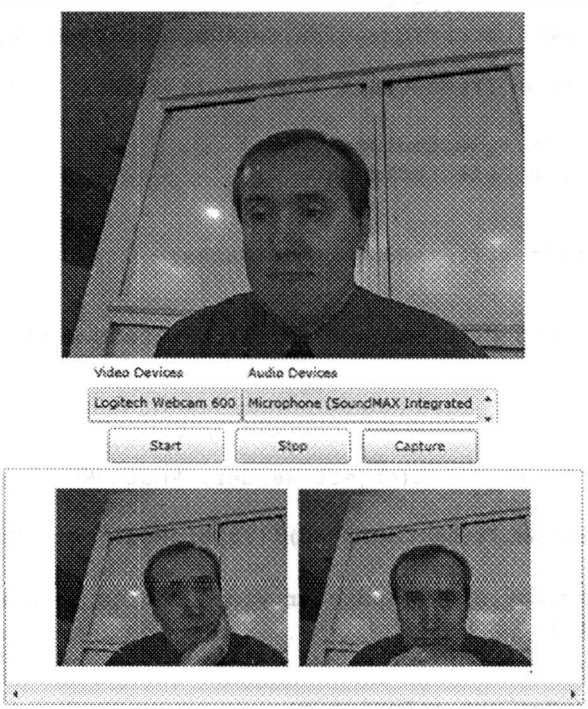


Рис. 1.9. Результат работы приложения

При запуске данного приложения должно получиться что-то такое (если Вы увидите свое лицо вместо моего, то не расстраивайтесь, так задумано) (рис. 1.9).

В завершение хочу отметить два класса: **AudioSink** и **VideoSink**, которые позволяют получить доступ к видео и аудио, как к набору байт. Оба класса абстрактные, поэтому от разработчика потребуется определить собственные классы и несколько методов.

Поддержка колесика мыши

Вспоминаю те времена, когда компьютерные мышь имели всего две кнопки, а мышь с колесиком представляла собой нечто диковинное, чему сложно было найти применение. Сейчас сложно представить мышь, которая не имеет колесика. При этом наступает сильное раздражение, когда, в редких случаях, приложение не поддерживает колесико мыши. Теперь поддержка колесика мыши есть и в Silverlight.

Реализуем простое приложение, в интерфейсе которого имеется изображение, к которому мы применим трехмерную проекцию по оси X. Вот код интерфейса приложения:

```
<UserControl x:Class="MouseWheel_Chapter0.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
>
    <Grid x:Name="LayoutRoot" Background="White">
        <Image Source="Penguins.jpg" Width="640"
            Height="480" MouseWheel="Image_MouseWheel">
            <Image.Projection>
                <PlaneProjection x:Name="proj"></PlaneProjection>
            </Image.Projection>
        </Image>
    </Grid>
</UserControl>
```

Как видно, тут мы определили обработчик события **MouseWheel**, которое генерируется при вращении колесика мыши. Код обработчика довольно простой:

```
private void Image_MouseWheel(object sender, MouseWheelEventArgs e)
{
    proj.RotationX += e.Delta/10;
}
```

Запустите приложение, щелкните на картинке и начните вращать колесико мыши — картинка будет вращаться вокруг оси X.

Нужно отметить, что все встроенные элементы, работающие с набором записей, поддерживают работу с колесиком мыши по умолчанию. Сюда относятся такие элементы как **DataGrid**, **ListBox** и др.

Элемент управления ViewBox

Чтобы продемонстрировать работу нового элемента управления **ViewBox**, достаточно рассмотреть небольшой пример:

```
<UserControl x:Class="ViewBox_Chapter0.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid x:Name="LayoutRoot" Background="White">
        <StackPanel>
            <Button Content="Hello" Width="400"
Height="100"></Button>
        </StackPanel>
    </Grid>
</UserControl>
```

Если попробовать запустить приложение выше и изменить размер окна браузера, то можно увидеть, что размер кнопки остается неизменным. Если изменить размер окна браузера, сделав его меньшим размера кнопки, то она попросту перестанет помещаться на экран. Таким образом, существует проблема изменения размеров интерфейса при изменении размеров окна браузеров. Конечно, данную проблему можно решить с помощью элемента компоновки **Grid**, но при работе со сложными интерфейсами элемент **Grid** не помогает. Поэтому появление элемента **ViewBox**, позволяющего сжимать и расширять свое содержимое, является даже запоздалым.

Замечание. Элемент ViewBox был доступен и разработчикам более ранней версии SilverLight, но не входил в стандартную поставку, а поставлялся в составе специальной библиотеки с открытым кодом, разрабатываемой сообществом.

Рассмотрим пример выше, но добавим элемент компоновки **ViewBox**:

```
<UserControl x:Class="ViewBox_Chapter0.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid x:Name="LayoutRoot" Background="White">
        <Viewbox MaxHeight="200" MaxWidth="800"
            MinHeight="50" MinWidth="200">
            <StackPanel>
                <Button Content="Hello" Width="400"
Height="100"></Button>
            </StackPanel>
        </Viewbox>
    </Grid>
</UserControl>
```

Теперь, при изменении размеров окна браузера, также изменяется и размер нашей кнопки.

Рассмотрим элемент **ViewBox** более детально. Фактически, тут можно выделить всего три основных свойства: **Child**, **Stretch**, **StretchDirection**.

Свойство **Child** содержит ссылку на тот элемент, размерами которого управляет **ViewBox**. Тут может быть только один элемент, но это может быть и контейнер (как в примере выше).

Свойство **Stretch** определяет, каким образом содержимое **ViewBox** размещается внутри выделенного пространства. Тут возможно одно из следующих значений:

- **None** — сохранять исходный размер контента;
- **Fill** — полностью заполнить элемент **ViewBox**, не сохраняя пропорции;
- **Uniform** — сохранять пропорции контента;
- **UniformToFill** — полностью заполнить элемент **ViewBox**, с сохранением пропорций контента. Естественно, что с этим параметром в большинстве случаев весь контент отобразить не удается, и он выходит за границы **ViewBox**.

Последнее свойство — **StretchDirection** — позволяет разрешить только сжатие или расширение содержимого и принимает одно из значений:

- **UpOnly** — только расширять контент, если размер ViewBox больше размера самого контента;
- **DownOnly** — только сжимать контент, если размер ViewBox меньше размера самого контента;
- **Both** — позволяет расширять и сжимать контент (установлено по умолчанию).

Таким образом, в Silverlight 4, как минимум, на один полезный элемент стало больше.

Повышение доверия

При создании SilverLight 2 и 3 в Microsoft задумывались о том, чтобы реализовать возможность размещения приложений в различных группах безопасности кода (подобие Code Access security). Данный подход не только **не был** реализован, но и принадлежность всех Silverlight-приложений к единому контексту безопасности (Web «песочнице») позиционировалось как преимущество технологии. Ведь если в данном контексте нет прав, например, на форматирование жесткого диска, то даже при наличии ошибок в приложении разработчика, возможность для атаки будет отсутствовать.

В Silverlight 4 ситуация кардинально изменилась. Теперь приложениям, устанавливаемым для работы **вне браузера**, можно назначать контекст безопасности с повышенными правами. Чтобы сделать это, достаточно немного изменить конфигурационный файл, установив атрибут **ElevatedPermissions** в значение **Required**:

```
<OutOfBrowserSettings.SecuritySettings>

   <SecuritySettings ElevatedPermissions="Required" />
</OutOfBrowserSettings.SecuritySettings>
```

При инсталляции такого приложения пользователь получает вот такое окно:

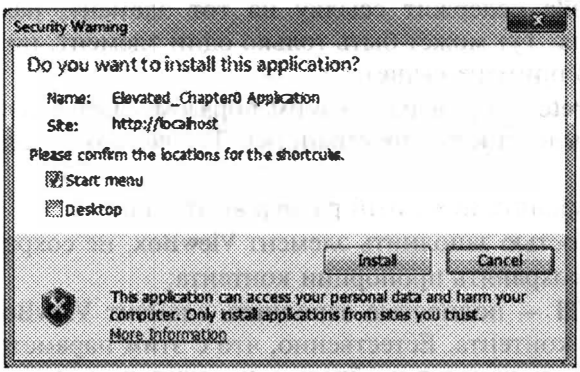


Рис. 1.10. Запрос пользователя на предоставление дополнительных полномочий

Несмотря на то, что вид окна отличается от того, который используется для инсталляции обычных приложений, работающих вне браузера, нужно помнить, что приложения разрабатываются для обычных пользователей. Я встречал громадное количество пользователей, которые смело жмут кнопку Install, не читая предупреждений (или игнорируя их). Поэтому данная возможность безопасна скорее для корпоративных пользователей, где у администратора есть возможность запретить установку всех приложений, требующих повышенный полномочий. При этом перед установкой запрета можно установить все необходимые приложения, которые после установки запрета будут работать и дальше.

Расширенные возможности работы в полноэкранном режиме

В стандартных Silverlight-приложениях нельзя перейти в полноэкранный режим без реакции на событие, инициированное пользователем. Кроме того, в полноэкранном режиме действуют ограничения, связанные с клавиатурой (невозможен ввод).

Если Ваше приложение запущенно в режиме с повышенным доверием, то все ограничения на полноэкранный режим снимаются: режим можно включать в любом месте программы, интерфейс реагирует на ввод с клавиатуры, как и обычный интерфейс. Кроме этого, теперь пользователь не сможет выйти из полноэкранного режима, нажав **Esc**. Это сделано для того, чтобы дать возможность определить собственное поведение для **Esc**. Поэтому разработчик должен помнить о том, что пользователю необходимо предоставить альтернативный способ покинуть приложение.

Отсутствие сообщений о доступе к ресурсам

При работе с буфером обмена и другими внешними ресурсами (камера, микрофон), пользователь не будет получать сообщения, позволяющие разрешить или запретить доступ. При этом, такие классы как **Clipboard**, можно использовать в любом месте кода.

Запросы между доменами

Следующая возможность, доступная для приложений с повышенным доверием, — это отправка запросов службам и приложениям, находящимся в любом домене. При этом не важно, какую конфигурацию имеет удаленным домен. Данная функциональность безвредна для пользователя, но незаменимая при организации DOS-атак ☺.

Доступ к некоторым папкам

В режиме с повышенным доверием приложение получает полный контроль к некоторым специальным папкам, ассоциированными с пользователем: **MyDocuments**, **MyVideos**, **MyPictures**, **MyMusic**. Пути к указанным папкам можно получить, используя статический класс **Environment**. При этом нужно отметить, что Silverlight-приложение работает только с папками, но не с библиотеками (Win 7).

Данная возможность хоть и ограничена, но одна из самых опасных. Я не вижу никаких ограничений, запрещающих мне записать исполняемый файл в одну из этих папок, а затем запустить его. Если при этом пользователь имеет выключенным нотификации UAC (Windows 7), то права такого приложения будут неограниченные.

Взаимодействие с COM

И, наконец, самая интересная возможность — доступ к COM API.

С одной стороны это означает, что теперь разработчик может получить доступ к COM-модели таких продуктов как Word и Excel. Это позволит создавать документы «на лету» и заполнять их данными, выбранными пользователем из внешних хранилищ.

С другой стороны, пользователь обычно имеет достаточно большой спектр COM-объектов, установленных по умолчанию. Например, **WScript**, позволяющий запускать приложения из любой директории на вашей машине (даже те, которые Вы можете скопировать, используя предыдущую возможность ☺):

```
dynamic cmd = ComAutomationFactory.CreateObject("WScript.Shell");
cmd.Run(string.Format(writerApp, @ArgumentFileBox.Text),1, true);
```

Таким образом, приложения с повышенным доверием имеют достаточно много прав, чтобы навредить пользователю. Будет ли от подобных приложений реальный вред, покажет время.

В завершении темы хочу отметить, что объект **Application** имеет специальное свойство **HasElevatedPermissions**, которое позволяет проверить, запущено ли это приложение с повышенным доверием. Свойство имеет смысл, если Вы будете делать инсталляции сразу двух типов приложений, работающих вне браузера (с повышенным доверием и стандартное).

Неявные стили

На этот раз рассмотрим возможность, облегчающую работу при построении интерфейса приложения.

В предыдущих версиях Silverlight, чтобы иметь возможность установить новый стиль одному или нескольким элементам управления, разработчик должен был создать именованный стиль и установить имя созданного стиля для каждого из элементов управления.

То есть стиль мог выглядеть следующим образом:

```
<Style TargetType="Button" x:Key="btnStyle">
 <Setter Property="FontFamily" Value="Arial Black"></Setter>
 <Setter Property="Background" Value="Green"></Setter>
 <Setter Property="FontStyle" Value="Italic"></Setter>
 <Setter Property="Foreground" Value="Red"></Setter>
</Style>
```

А код, устанавливающий стиль для конкретного элемента, мог выглядеть так:

```
<Button Style="{StaticResource btnStyle}"
 Content="Hello" Width="100" Height="50"></Button>
```

Теперь при определении стиля от атрибута **x:Key** можно избавится. Это будет означать, что указанный стиль будет использован неявно для всех элементов заданного типа. Естественно, если в элементе прописать стиль явно, то неявный стиль для этого элемента будет игнорироваться.

Заключение

Итак, Silverlight 4 имеет предостаточно возможностей, чтобы заинтересовать разработчика. Некоторые из этих возможностей мы будем упоминать и дальше, а какие-то просто использовать как должное. Между тем, тенденция, которая наблюдается в Silverlight 4, — это попытка покрыть корпоративный рынок. Несомненно, корпорация Microsoft преуспела в работе с корпоративными пользователями, но Silverlight 4, в первую очередь, — Web-технология для широкого круга Internet пользователей. Поэтому, данная стратегия вызывает вопросы. Хотя, возможно, такая стратегия и есть самая верная, так как хороших приложений на том же Flash достаточно мало. Лично у меня, Flash ассоциируется только с рекламой. Не хотелось бы, чтоб такая же судьба постигла и Silverlight.

Глава 2

НАЧИНАЕМ РАБОТУ С SILVERLIGHT

Что такое Silverlight?

Прошло не так много времени с тех пор, как приложения имели текстовый интерфейс, а для работы с некоторыми программами требовались десятки часов обучения. Сегодня приложения обладают удобным и интуитивно понятным графическим интерфейсом. Разработка же таких интерфейсов давно была поставлена на поток, а любой разработчик знаком со стандартным набором элементов управления, таких как кнопка, текстовое поле и т. п. При этом стандартные элементы управления доступны как в Web-, так и в Windows-мире. Но технологии не стоят на месте, и сейчас мы находимся на том уровне, когда на смену стандартным механизмам ввода (мышь, клавиатура) приходят touch-интерфейсы. Уже сегодня есть несколько образцов мониторов, которые поддерживают реакцию на множественное касание, а устройства, реагирующие на одиночное касание, можно купить в любом компьютерном магазине. Это означает, что интерфейсы будущего будут значительно отличаться от тех, что мы видим сегодня. С одной стороны там будут присутствовать новые элементы управления, а с другой — интерфейс станет еще более интуитивно понятным (без посредников в виде мыши). Это означает, что разработчик должен иметь доступ к технологиям, которые позволили бы ему создавать интерфейсы будущего уже сейчас. Одна из таких технологий — Silverlight.

Создавая Web-приложения, программист ограничен скромными возможностями при реализации интерфейсов. Независимо от того, какая технология будет использована для создания Web-приложения, конечный пользователь будет использовать браузер, который способен отображать только HTML. Используя HTML и JavaScript, можно добиться некоторой динамики и интерактивности интерфейса, но интерфейсы Web-приложений все еще далеки от интерфейсов Windows-приложений. Кроме того, Windows-приложения продолжают активно развиваться. Так, с выходом Windows Presentation Foundation (одной из составляющих .NET Framework 3.0), программист получил возможность создавать абсолютно произвольные интерфейсы, наполненные элементами управления, вид которых ограничивается лишь фантазией программиста. Кроме того, WPF поддерживает работу с трехмерной графикой, анимацию, векторный подход при построении интерфейсов, touch-технологию и др. Все

это делает WPF мощным инструментом для построения интерфейсов, адаптированных для работы на новых устройствах.

Еще перед выходом Windows Presentation Foundation компания Microsoft заявила о разработке технологии, которая переносит возможности WPF в Web. Эта технология получила рабочее название WPF/Everywhere. И только весной 2007 года на конференции Mix 07 в Лас-Вегасе была представлена бета версия технологии WPF/Everywhere, получившая коммерческое название Silverlight (все материалы конференции Mix07 доступны на сайте http://videos.visitmix.com).

Проводя аналогию с WPF, можно сформулировать, что технология Silverlight позволяет создавать приложения в Web с «богатым» интерфейсом, то есть интерфейсом, наполненным графикой, анимацией, медиа-компонентами. Приложения, разработанные на Silverlight, выполняются в окне браузера пользователя, внутри специального подключаемого компонента (plug-in). Фактически, возможности подключаемого компонента и определяют возможности Silverlight-приложений.

> *Замечание.* Лучший способ понять и оценить возможности Silverlight — исследовать существующие примеры, с которыми можно ознакомиться на сайте http://silverlight.net/showcase.

Итак, технология Silverlight 1 появилась в сентябре 2007 года. В то время Silverlight можно было использовать лишь для ограниченного числа сценариев. В основном это было отображение видео. Логику Silverlight 1 приложений можно было реализовывать только на JavaScript, что существенно сказывалось на скорости и возможностях.

Завоевав небольшую долю рынка, Silverlight 1 не нашел большого количества приверженцев и скорее являлся небольшой демонстрацией того, что может быть в будущем. А будущее не заставило себя долго ждать. В октябре 2008 года свет увидела вторая версия технологии Silverlight. Именно эта версия позволила разрабатывать Web-приложения с поддержкой библиотек .NET Framework. При этом, разработчик смог использовать такие языки программирования как C# или VB.NET, либо целый набор динамических языков программирования (IronPython, IronRuby, Jscript).

В отличие от первой версии технологии, приложения Silverlight 2 отличались высокой производительностью, появилась возможность создавать элементы управления, а разработчик смог использовать знакомую объектную модель и подходы из .NET Framework.

Технология Silverlight 2 просто ворвалась на рынок, предоставив разработчикам все, что им было необходимо. Но, не прошло и года, как в июле 2009 пользователям и разработчикам стал доступен релиз третьей версии. Компонент Silverlight 3 позволял отображать трехмерные проекции, использовать собственные эффекты, сохранять файлы на диск и многое другое. На момент написания этой книги, Silverlight 3 был установлен более чем на 50% всех компьютеров и являлся текущей версией.

И вот, в ноябре 2009 года, Microsoft анонсировала четвертую версию уже популярной технологии. Релиз Silverlight 4 планируется на май 2010 года, но

уже сейчас разработчики могут изучать бета версию (а к моменту выхода книги и релиз-кандидат).

Для разработки Silverlight-приложений используются две технологии, это XAML (eXtensible Application Markup Language) — язык, позволяющий описывать векторный интерфейс, и один из управляемых или динамических языков программирования, позволяющий реализовать логику приложения. Таким образом, разработчик не сталкивается с проблемой изучения принципиально новых технологий, а использует накопленные ранее знания. Кроме того, для кодирования приложений разработчик может использовать существующие утилиты, начиная от текстового редактора Notepad и заканчивая редактором Visual Studio 2010, который мы и будем использовать в книге.

Говоря о технологии для создания Web-приложений, необходимо сделать акцент на независимость Silverlight 4 от браузера и платформы.

Если говорить о платформенной независимости, то следует вспомнить, что Silverlight является технологией, выполняющей приложения на машине пользователя. Фактически, пользователь должен иметь только встраиваемый компонент для запуска приложения. Специфичные для Silverlight серверные ресурсы не используются, как и отсутствует привязка к определенной серверной платформе или Web-серверу. Последнее означает, что для того, чтобы опубликовать Silverlight-приложение на Web-сервер, его достаточно скопировать. В качестве Web-сервера можно использоваться Internet Information Service (IIS), Apach или любой другой. При этом сайт, в который внедряется Silverlight-приложение, может быть реализован на технологиях, начиная от простого HTML или PHP и заканчивая ASP.NET.

Независимость от браузера достигается поддержкой встраиваемого компонента для различных типов браузеров. Так встраиваемый компонент для Silverlight 4 поддерживается такими браузерами, как Internet Explorer 6, 7 и 8 версий, FireFox 1.5 и 2.x, Safari, Google Chrome. Microsoft реализует поддержку встраиваемого компонента на таких платформах как Windows и Mac OS. В то же время Microsoft не имеет возможности реализовывать и поддерживать компонент на платформе Linux. Несмотря на это, пользователи этой операционной системы не были обделены. Так, в 2007 году, Microsoft заключила договор о сотрудничестве с компанией Novell. Результатом этого сотрудничества стал выпуск встраиваемого компонента на платформе Linux. Компонент имеет название Moonlight и поддерживает приложения, разработанные на Silverlight. Детально о технологии Moonlight можно почитать тут: http://www.mono-project.com/Moonlight.

Таким образом, Silverlight является технологией, независимой от браузера и от платформы.

Инструменты для создания Silverlight-приложений.

Прежде чем приступить к созданию первого Silverlight-приложения, рассмотрим, какие утилиты необходимо установить для работы с материалом этой книги.

Мне достаточно сложно указать прямые ссылки, так как сейчас Silverlight 4 доступен только в бета-версии, а с выходом релиза ссылки станут другими. Поэтому я просто перечислю набор необходимых продуктов, а Вы их попробуете найти сами по следующим ссылкам: http://www.microsoft.com/silverlight/resources/technical-resources/default.aspx и http://www.silverlight.net.

Основной утилитой, которая используется разработчиками для платформы .NET Framework является Visual Studio. Для создания Silverlight-приложений подходит любая редакция Visual Studio 2010 или бесплатная версия продукта для разработки Web-приложений — Visual Web Developer.

Нужно отметить, что Visual Studio 2010 обладает визуальным редактором для создания Silverlight-интерфейсов, что не было доступно в предыдущей версии. Но реализация Visual Studio задумана таким образом, чтобы работать с любой версией Silverlight, установленной на машине разработчика. Поэтому, чтобы Вы смогли создавать приложения на основе уже встроенных шаблонов, необходимо установить SDK, включающее специальную версию встраиваемого Silverlight-компонента, позволяющего отлаживать приложения. Вместе с SDK Вам будет установлен дополнительный набор библиотек, расширяющий набор существующих компонентов. Формально, вместе с Visual Studio 2010 устанавливается Silverlight SDK, но только для третьей версии, которую Вы можете использовать для поддержки старых приложений. Если говорить о четвертой версии, то ее выход намечен на месяц позже, чем выход Visual Studio, поэтому в инсталляции ее быть не должно.

Итак, Visual Studio и Silverlight SDK, составляют необходимый, но не достаточный пакет для разработки приложений. Ведь если речь идет о Silverlight-приложениях, то, как правило, подразумевают интерфейсы, отличающиеся от стандартных форм ввода. Тут не обойтись без услуг дизайнера. Но ведь дизайнер не будет запускать Visual Studio, и, тем более, заниматься созданием кода на XAML. Вместо этого дизайнер должен иметь доступ к набору утилит, позволяющих легко создавать отдельные изображения и интерфейс с помощью визуальных средств. Поэтому следующий набор утилит, который может быть полезен для создания Silverlight-приложений, предназначен в основном для дизайнеров — это Microsoft Expression Studio. К сожалению, тут нет бесплатной версии продукта, но Вы сможете скачать пробные версии с сайта http://www.microsoft.com/expression/try-it. Пакет Expression Studio включает следующие утилиты:

- **Expression Blend** — это самая полезная утилита при создании Silverlight-приложений. Фактически она позволяет работать с проектом в формате Visual Studio, но, в отличии от последнего, предназначена для создания и редактирования интерфейса визуальными средствами. Утилита устанавливает «мост» между дизайнером и разработчиком, когда дизайнер занимается созданием интерфейса, а разработчик — написанием кода. При этом они ведут работу над одним проектом в одно и то же время;
- **Expression Design** — утилита предназначена сугубо для рисования. Если дизайнеру необходимо создать какой-то сложный элемент и преобразовать его в формат XAML, то эта утилита для него;

- **Expression Encoder** — данный продукт позволяет преобразовывать видео из одного формата в другой, с возможностью удаления выбранных фрагментов. Кроме того, Expression Encoder 3 способен записывать изображение с экрана монитора. Поэтому продукт очень популярен при организации трансляций видео в сети, а также при создании учебных материалов. Если кто-то использовал ранее Windows Media Encoder, то Expression Encoder полностью его заменяет;
- **Expression Web** — утилита позволяет создавать Web-сайты и будет интересна специалистам, которые занимаются верстkой страниц.

Изучая Silverlight 4, мы также обратимся к бесплатному набору компонентов, разрабатываемому сообществом — Silverlight Toolkit, который доступен по адресу http://silverlight.codeplex.com/. Обязательно установите и его, тут есть много полезных элементов управления.

Вот и весь набор утилит, которые Вам стоит установить уже сейчас. Позже мы познакомимся с Deep Zoom технологией, что потребует дополнительное ПО, но сейчас это не существенно.

Первое приложение в Expression Blend 4

Попробуем реализовать простейшее приложение на Silverlight 4. Для этого попробуем использовать приложение Microsoft Expression Blend 4, которое было разработано специально для дизайнеров.

Попробуйте запустить Microsoft Expression Blend 4 и создать новый проект. В результате на экране появится следующее окно:

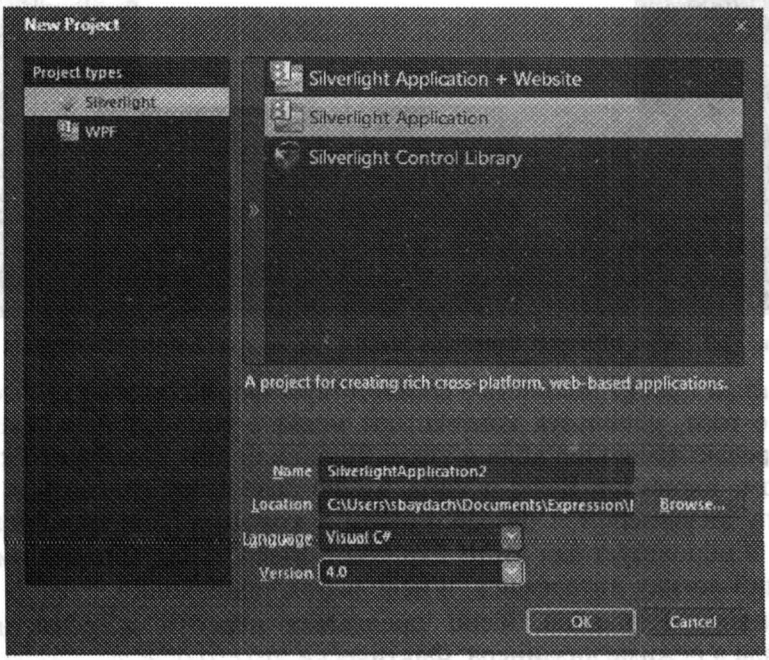


Рис. 2.1. Создание проекта в Microsoft Expression Blend 4

Как видно, Expression Blend позволяет создавать не только Silverlight-интерфейсы, но и имеет отдельный набор WPF-проектов. Это связано с тем, что подход при разработке WPF- и Silverlight-приложений полностью совпадает, а для построения интерфейса используется XAML, что позволило написать универсальный редактор.

Среди доступных Silverlight-шаблонов можно выбрать как **Silverlight 4 Application + WebSite**, так и **Silverlight Application**. Разница состоит только в том, что первый шаблон генерирует дополнительный проект, содержащий разнообразные Web-страницы, позволяющие тестировать Silverlight-приложение. Второй шаблон позволяет генерировать тестовую страницу во время компиляции приложения.

Введите имя проекта, и нажмите кнопку **OK**. В результате этих действий на экране отобразится рабочее окно:

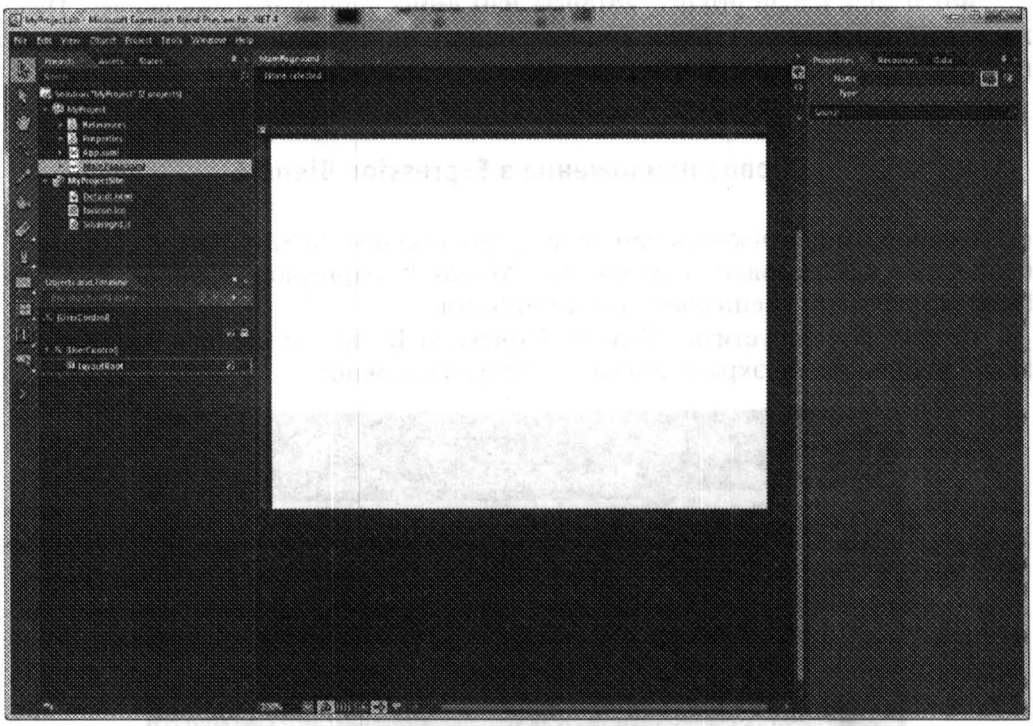


Рис. 2.2. Окно Microsoft Expression Blend 4 после создания нового проекта

Созданный проект полностью соответствует формату проекта Visual Studio 2010. Более того, используя контекстное меню в окне проекта, очень легко перейти к редактированию кода в Visual Studio или Web Developer Express. Таким образом, Expression Blend и Visual Studio позволяют наладить взаимодействие между дизайнером и разработчиком в команде. Так, дизайнер создает интерфейс и использует для этого Expression Blend, а разработчик пишет код, реализующий логику интерфейса, и использует Visual Studio.

Как видно, Expression Blend предлагает перейти к редактированию XAML-файла в режиме дизайнера. Фактически это его основная задача. Чтобы приступить к созданию интерфейса можно сразу перейти к панели инструмен-

тов, на которой расположены базовые элементы. Если Вы хотите модифицировать эту панель и получить доступ к другим элементам управления, то нажмите **Assets** (самую нижнюю кнопку на панели инструментов). Окно **Assets** также доступно в виде отдельной вкладки на одной панели с вкладкой **Projects**.

Рядом с панелью инструментов находится окно **Objects and Timeline**, отображающее дерево объектов в XAML. Это окно позволяет легко перемещаться по иерархии, даже если объект не видимый. Тут же можно реализовать анимацию любого из объектов.

Следующие три окна, размещенные в отдельных вкладках, находятся в противоположной стороне по отношению к панели инструментов. Одно из этих трех окон — **Properties**. Именно это окно позволяет задавать свойства любого из объектов в интерфейсе приложения.

Наконец, чтобы отобразить код XAML непосредственно, Вы можете воспользоваться кнопкой, находящейся в верхнем правом углу окна редактора. Хотя Expression Blend и поддерживает IntelliSense, он все же не предназначен для прямого редактирования кода.

Реализуем небольшое приложение, отображающее видео на фоне некоторой надписи. Для этого нам понадобится специальный элемент, который называется **MediaElement**. По умолчанию его нет на панели инструментов. Для того, чтобы отобразить этот элемент, нажмите кнопку **Assets** на панели инструментов и в появившемся окне выберите **Controls->All**.

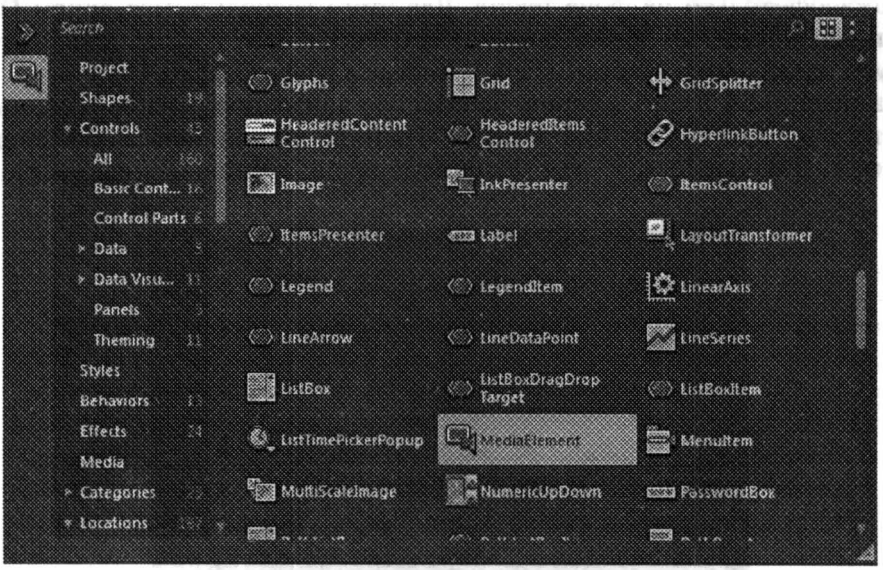


Рис. 2.3. Выбор Media-элемента

Разместите **MediaElement** в рабочей области, а в окне **Properties** на закладке **Media** установим свойство **Source**, выбрав один из видео файлов, поддерживаемых Windows. В результате **MediaElement** отобразит первый кадр из выбранного видео файла.

Разбавим наше видео небольшими эффектами. Для этого выберем на панели инструментов элемент управления **TextBlock** и разместим его поверх ви-

део элемента. Введите любую надпись (например, «Hello»), выбрав шрифт и размер таким образом, чтобы буквы было достаточно хорошо видно. Рабочая область вашего окна должна выглядеть примерно следующим образом:

Рис. 2.4. Размещение видео и текста

Теперь проведем «слияние» видео и текстовой надписи так, чтобы видео отображалось только на фоне текста. Для этого воспользуемся окном **Objects and Timeline** и выделим **MediaElement** и **TextBlock** в одну группу, после чего используем пункт меню **Objects->Path** и выберем **Make Clipping Path**.

На последнем этапе добавим небольшую анимацию в наше приложение. Для этого в окне **Objects and Timeline** нажмите кнопку **New...** для создания анимации. В появившемся окне установите время анимации (например, 3 се-

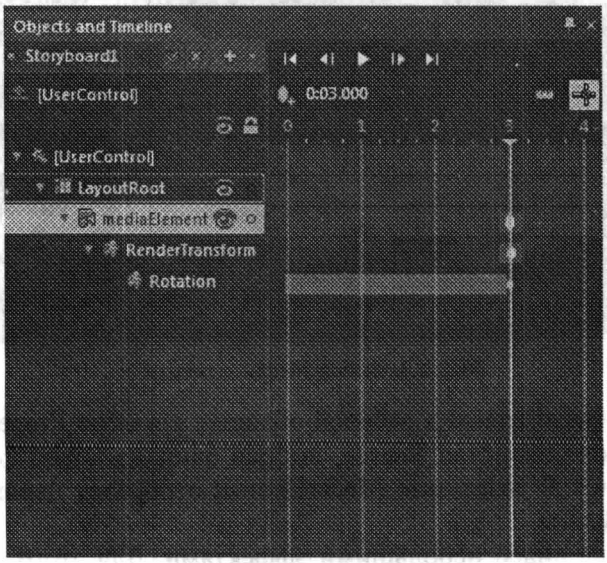


Рис. 2.5. Создание анимации

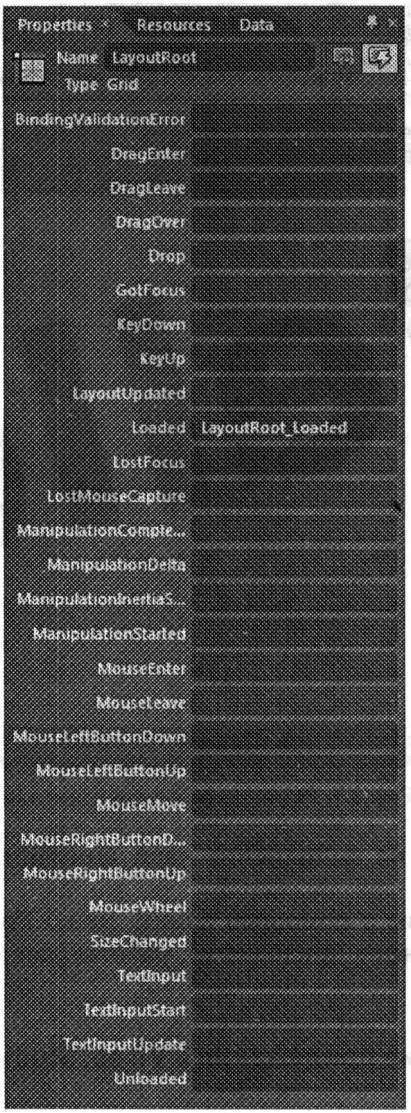


Рис. 2.6. Создание обработчика события Loaded

кунды), передвинув ползунок, и поверните **MediaElement** на 360 градусов вокруг своей оси.

С помощью кнопки **Play** можно выполнить созданную анимацию. А вот теперь придется написать немного кода. Чтобы анимация заработала, выберем в окне **Objects and Timeline** контейнер для нашего медиа-элемента (по умолчанию он имеет название LayoutRoot) и определим обработчик события **Loaded**. Для этого в окне свойств перейдем к закладке **Events** и выполним двойной щелчок на событии.

А вот и код, который нужно написать:

```
Storyboard1.Begin();
```

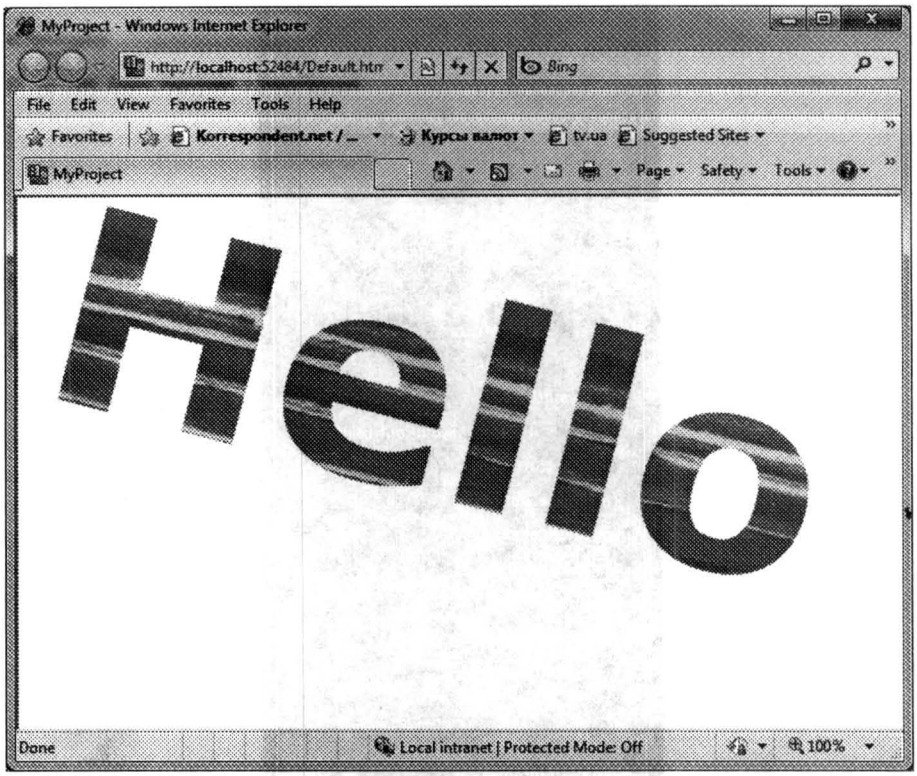


Рис. 2.7. Результат работы приложения

Этот код запускает на выполнение нашу анимацию, после загрузки контейнера.

Запустив приложение с помощью пункта меню **Project->Run Project** можно посмотреть на созданное приложение, работающее в окне браузера.

Создание приложения в Visual Studio 2010

Expression Blend 4 является мощной утилитой и позволяет сделать сколь угодно сложный дизайн интерфейса. В то же время эта утилита совершенно не подходит для написания кода, например, на C#. Ведь привычным инструментом для разработчика, является Visual Studio. Новая версия этого продукта — Visual Studio 2010 — поддерживает шаблоны для создания Silverlight-приложений.

Создадим новый проект, используя шаблон Silverlight Application. Visual Studio 2010 позволяет выбрать, какую версию SilverLight будет использовать наше приложение и нужно ли создавать Web-сайт с внедренным Silverlight-объектом, или генерировать тестовую страницу во время компиляции.

Оставим все параметры по умолчанию. В результате будет создан проект, структура которого будет рассматриваться в следующей главе. Модифицируем код страницы MainPage.xaml, как показано ниже.

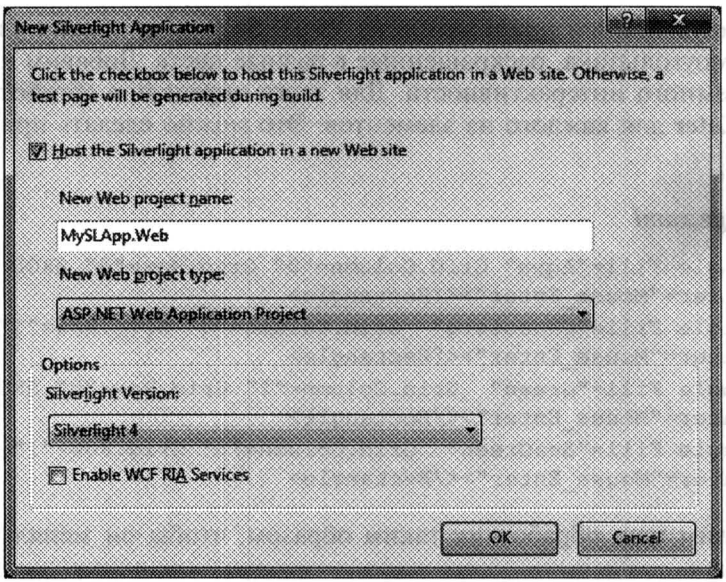


Рис. 2.8. Настройка параметров приложения

MainPage.xaml

```
<UserControl x:Class=" SilverLight1_2.Page"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot" Background="White">
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Rectangle Fill="Aqua" Grid.Column="0"
Grid.Row="0"></Rectangle>
        <Rectangle Fill="Chocolate" Grid.Column="0" @KOD =
Grid.Row="1"></Rectangle>
        <Rectangle Fill="Green" Grid.Column="1"
Grid.Row="0"></Rectangle>
        <Rectangle Fill="SeaGreen" Grid.Column="1"
Grid.Row="1"></Rectangle>
    </Grid>
</UserControl>
```

Этот код описывает один из элементов компоновки **Grid**, содержащий два столбца и две строки. В каждую из ячеек элемента **Grid** мы размещаем элемент **Rectangle**, определяющий прямоугольник заданного цвета.

Если запустить это приложение из Visual Studio, то можно увидеть лиш четыре прямоугольника, раскрашенные в разные цвета. Добавим к этому при ложению немного интерактивности. Для этого определим обработчики собы тия **MouseEnter** для каждого из элементов. Это можно сделать прямо в XAMl файле.

MainPage.xaml

```
<Rectangle Fill="Aqua" Grid.Column="0" Grid.Row="0" @KOD =
MouseEnter="Mouse_Enter"></Rectangle>
<Rectangle Fill="Chocolate"  Grid.Column="0" Grid.Row="1" @KOD =
MouseEnter="Mouse_Enter"></Rectangle>
<Rectangle Fill="Green"  Grid.Column="1" Grid.Row="0" @KOD =
MouseEnter="Mouse_Enter"></Rectangle>
<Rectangle Fill="SeaGreen"  Grid.Column="1" Grid.Row="1" @KOD =
MouseEnter="Mouse_Enter"></Rectangle>
```

Определим наш обработчик таким образом, чтобы он менял цвет у пря моугольника.

MainPage.xaml.cs

```
private void Mouse_Enter(object sender, MouseEventArgs args)
{
   Rectangle rect = (Rectangle)sender;
   Random r=new Random();
```

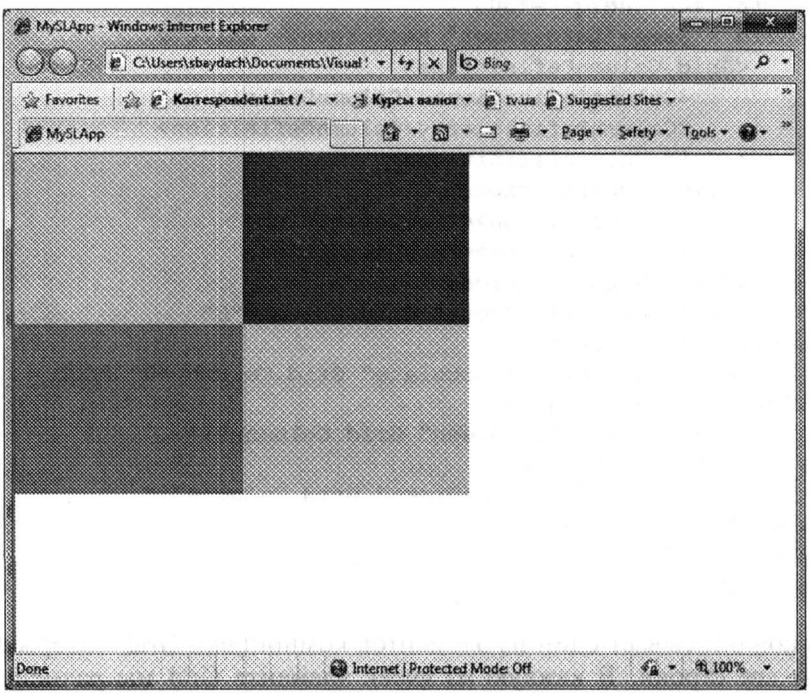


Рис. 2.9. Результат работы приложения

```
    rect.Fill = new SolidColorBrush(
        Color.FromArgb((byte)r.Next(255),
                       (byte)r.Next(255),
                       (byte)r.Next(255),
                       (byte)r.Next(255)));
}
```

Запустив приложение снова, на экране отображаются четыре прямоугольника, которые меняют цвет всякий раз, если наводить на них мышку (рис. 2.9).

Таким образом, создание проекта в Visual Studio не намного сложнее, чем в Expression Blend. В то же время Visual Studio обладает хорошим редактором кода с системой Intellisence и возможностями рефакторинга.

Обзор технологии

Теперь, когда мы имеем общие представления о Silverlight и умеем создавать простые приложения, давайте рассмотрим основные составляющие технологии, с которыми мы будем сталкиваться по ходу книги.

XAML

Как было сказано выше, для описания интерфейса Silverlight-приложений используется XAML. Все визуальные (и не только) элементы, доступные в Silverlight, имеют свое представление в виде XAML элементов. Например, следующая часть кода, описывает элемент, отображающий видеофайл:

```
<MediaElement Margin="0,0,0,0" Source="Butterfly.wmv"/>
```

При этом если есть необходимость создавать элементы динамически, то это можно делать, используя аналогичные классы, на одном из поддерживаемых языков (например C#):

```
MediaElement m = new MediaElement();
m.Margin = new Thickness(0, 0, 0, 0);
 m.Source = new Uri(".........");
 LayoutRoot.Children.Add(m);
```

Хоть XAML достаточно простой язык и базируется на XML, Вам необходимо познакомиться с основными синтаксическими особенностями этого языка. Это можно сделать в четвертой главе.

Элементы компоновки

Элемент, на базе которого строится весь интерфейс Silverlight-приложения (UserControl), может содержать лишь один элемент. Следовательно, этот единственный элемент должен быть контейнером. Таких контейнеров в

Silverlight несколько. Среди них такие элементы, как **Canvas**, **StackPanel** и **Grid**. Эти элементы не просто обеспечивают размещение нескольких элементов, но и поддерживают различные механизмы формирования компоновки элементов.

Первый элемент **Canvas** позволяет размещать элементы управления с абсолютной привязкой по позиции. **StackPanel** позволяет упорядочить элементы по горизонтали или вертикали. Последний элемент **Grid** позволяет разбить заданную область на ячейки и привязать элементы к каждой из ячеек.

Более детальную информацию об этих элементах Вы сможете найти в главе 5.

Элементы управления

Silverlight 4 предлагает достаточно большой набор элементов управления, начиная от элемента **Button**, описывающего кнопку, и заканчивая таблицами по работе с данными. Часть этих элементов поставляется в составе встраиваемого компонента Silverlight. Дополнительные элементы Вы сможете найти в SDK, а также на сайте http://codeplex.com. Описание основных элементов управления и принципы работы с ними будут описаны в пятой главе.

Графические примитивы

Естественно, что наряду с элементами управления в состав Silverlight 4 входят и графические примитивы. Сложно поверить, но именно графические примитивы являлись основными строительными блоками в Silverlight 1. В список элементов-графических примитивов входят такие элементы как **Line**, **Rectangle**, элементы позволяющие определить градиентную заливку и многие другие. Об этих элементах Вы узнаете в главе 8.

Управление видео

Именно простота внедрения видео, звука и анимации в Web-приложения делает Silverlight интересным для большинства разработчиков. С выходом Silverlight 4 у разработчика появилась возможность не просто работать с записанным видео, но и использовать видеокамеру и микрофон на стороне клиента. Однако, чтобы обеспечить высокое качество трансляции и эффективную работу сценариев, необходимо понимать серверные технологии, предназначенные для трансляции видео. Обо всех вопросах, касающихся работы с видео, мы поговорим в девятой главе.

Работа с данными

Silverlight 4 предоставляет множество механизмов по работе с данными. Сюда входит набор классов, которые позволяют работать с XML данными, связывать данные и элементы управления, поддерживать работу через LINQ. В Главе 6 мы рассмотрим эту тему.

Работа со службами

Silverlight 4 поддерживает такие механизмы работы со службами, как Windows Communication Foundation, SOAP, REST, JSON и др. Это позволяет наладить взаимодействия со службами всех типов. Работа с сервером будет обсуждаться в седьмой главе.

Работа вне браузера

Еще в Silverlight 3 появилась возможность устанавливать приложения на локальный компьютер пользователя для последующего запуска вне браузера (и в отключенном режиме). Silverlight 4 значительно расширяет эту концепцию, позволяя не просто установить приложение на машину пользователя, но и предоставить набор дополнительных привилегий. О приложениях, работающих вне браузера, речь пойдет в главе 13.

Базовые классы

Работа с Silverlight 4 удобна еще и тем, что разработчику можно использовать стандартные классы и подходы при разработке логики приложения, к которым он привык, разрабатывая другие типы приложений на платформе .NET Framework. Так, тут есть классы по взаимодействию с XML, работой с потоками, базовые функции и др. Часть базовых механизмов мы будем использовать практически в каждой главе.

Заключение

Silverlight 4 — новая технология от Microsoft, которая позволяет разрабатывать Web-приложения с «богатым» интерфейсом, наполненным графикой, видео, анимацией. Использование управляемых языков для бизнес-логики делает технологию Silverlight достаточно удобной для программистов, ранее разрабатывавших проекты на платформе .NET Framework. При этом есть возможность использовать динамические языки программирования, включая JavaScript, который является «мостом», связывающим Silverlight-модель с объектной моделью приложения в браузере (DOM). Это дает неограниченные возможности по манипуляции не только Silverlight-приложением, но и содержимым всей страницы.

Используя встраиваемый компонент, включающий все необходимые возможности для запуска приложений, Silverlight снимает ограничения с операционной системы, установленной на клиенте. Так, нет необходимости устанавливать .NET Framework или дополнительные библиотеки. Достаточно наличие встраиваемого компонента, который поддерживается на всех популярных ОС и во всех современных браузерах. Запуск и выполнение Silverlight-приложений на клиенте делает технологию независимой от серверной части, что позволяет размещать Silverlight-приложения под любым Web-сервером и внедрять на сайты, написанные на любом языке программирования, например PHP.

Глава 3

АРХИТЕКТУРА SILVERLIGHT

Структура приложения

Создадим простое приложение на Silverlight 4, используя Visual Studio 2010, так, как это было описано в предыдущей главе (включая дополнительный Web-проект, предназначенный для тестирования нашего приложения). Структура созданного решения показана ниже.

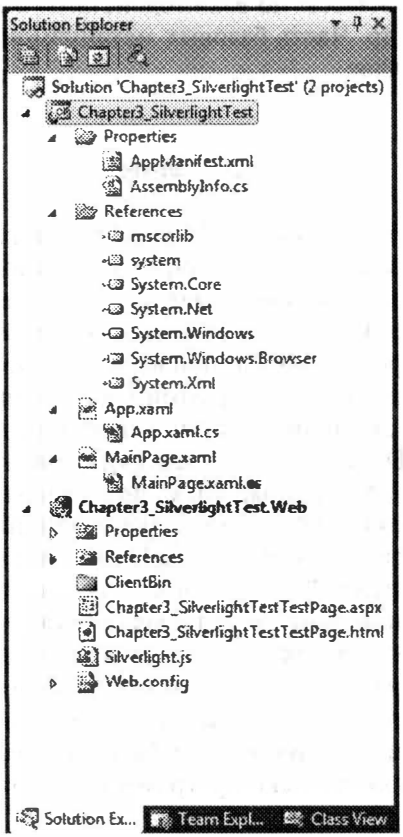


Рис. 3.1. Структура приложения

Первые два файла, на которые стоит обратить внимание, — это **App.xaml** и **App.xaml.cs**. Тут описывается класс App, который является наследником от класса **Application**. Именно класс **Application** обеспечивает точку входа при запуске Silverlight-приложения. Любое приложение на Silverlight должно иметь реализацию класса, который наследуется от **Application**.

Рассмотрим **App.xaml**.

App.xaml

```
<Application
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="Chapter3_SilverlightTest.App">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Фактически, XAML файл для класса **Application** можно было бы и не использовать, так как объект, порожденный от **Application** класса, не имеет визуального представления. Но в Silverlight-приложениях существует такой мощный механизм, как ресурсы. Именно ресурсы позволяют описать стили, шаблоны элементов управления и многое другое. Поскольку сложные Silverlight-приложения имеют больше чем одну «страницу», то **App.xaml** очень хорошо подходит для описания ресурсов, имеющих глобальную область видимости.

В свою очередь, именно в файле **App.xaml.cs** выполняется вся черновая работа по созданию приложения, загрузке визуальной части и обработке ошибок приложения. Рассмотрим этот код.

App.xaml.cs

```
namespace Chapter3_SilverlightTest
{
    public partial class App : Application
    {

        public App()
        {
            this.Startup += this.Application_Startup;
            this.Exit += this.Application_Exit;
            this.UnhandledException +=
this.Application_UnhandledException;

            InitializeComponent();
        }

        private void Application_Startup(object sender,
StartupEventArgs e)
        {
            this.RootVisual = new MainPage();
```

```
        private void Application_Exit(object sender, EventArgs e)
        {

        }

        private void Application_UnhandledException(object sender,
            ApplicationUnhandledExceptionEventArgs e)
        {
            if (!System.Diagnostics.Debugger.IsAttached)
            {
                e.Handled = true;
                Deployment.Current.Dispatcher.BeginInvoke(
                   delegate { ReportErrorToDOM(e); });
            }
        }

        private void ReportErrorToDOM(
            ApplicationUnhandledExceptionEventArgs e)
        {
            try
            {
                string errorMsg = e.ExceptionObject.Message +
                   e.ExceptionObject.StackTrace;
                errorMsg = errorMsg.Replace('"', '\'').Replace(
                   "\r\n", @"\n");

                System.Windows.Browser.HtmlPage.Window.Eval(
                       "throw new Error(\"Unhandled Error in
Silverlight Application " +
                       errorMsg + "\");");
            }
            catch (Exception)
            {
            }
        }
    }
}
```

Как видно, класс App — это **partial** класс, содержащий конструктор и ряд обработчиков событий, связанных с приложением. Использование ключевого слова **partial** объясняется тем, что во время компиляции кода Visual Studio сгенерирует вторую часть этого класса, реализующую объявление переменных и загрузку XAML кода. Но об этом в следующей главе. А сейчас рассмотрим события, связанные с **Application**:

- **Startup** — это событие генерируется перед тем, как приложение станет доступно пользователю. В обработчике можно реализовать такие механизмы: включить механизмы журналирования, инициализировать ресурсы и переменные приложения, восстановить состояние приложения из предыдущей сессии, отобразить визуальную составляющую приложения, и т. д. Так, в примере выше метод, обрабатывающий событие **Startup** служит для визуальной составляющей приложения. Обработчик создает экземпляр класса **MainPage**, объявленный в файле **MainPage.xaml.cs**

(MainPage.xaml) и устанавливает свойство **RootVisual**. Это свойство содержит ссылку на основное окно приложения;

- **Exit** — это событие генерируется всякий раз, как пользователь прекращает работу с приложением. Это может случиться в результате закрытия страницы, где работает приложение, обновления страницы, перехода на другую страницу, перезагрузки или выключения системы. Обычно в методе, обрабатывающем данное событие, выполняются действия, связанные с сохранением состояния приложения;
- **UnhandledException** — это событие позволяет перехватить и обработать все исключения, которые имели место, но не были обработаны в основном коде. Благодаря этому событию приложение может обработать исключение и продолжить работать дальше, если исключение было не критичным. Речь идет об исключениях, которые генерирует управляемый код.

Следующие два файла Silverlight-приложения — **MainPage.xaml** и **MainPage.xaml.cs**. Эти файлы описывают визуальный элемент, который будет использоваться в качестве интерфейса нашего приложения.

Рассмотрим код **MainPage.xaml.cs**:

```
namespace Chapter3_SilverlightTest
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();
        }
    }
}
```

Как видно, класс MainPage содержит лишь конструктор, вызывающий метод **InitializeComponent**. Самого объявления метода в этом файле нет. Вторая часть класса, которая содержит **InitializeComponent** и объявление переменных, будет находиться в файле, сгенерированном компилятором. Основное назначение класса состоит в определении логики работы интерфейса, например, в определении обработчиков событий, связанных с элементами управления в интерфейсе приложения.

Отметим, что класс MainPage является наследником от **UserControl**, который предоставляет инфраструктуру для создания комбинированных элементов управления. Ведь весь наш интерфейс — это не что иное, как комбинация более простых элементов управления (но без претензии на повторное использование).

Перейдем к файлу **MainPage.xaml**:

```
<UserControl x:Class="Chapter3_SilverlightTest.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/
          markup-compatibility/2006"
    mc:Ignorable="d"
```

```
    d:DesignHeight="300" d:DesignWidth="400">

    <Grid x:Name="LayoutRoot" Background="White">

    </Grid>
</UserControl>
```

Именно тут происходит вся основная работа по созданию интерфейса. Так, с помощью языка XAML декларируются все элементы управления, включая их стили, свойства и обработчики событий.

Как видно из кода, **UserControl** является родительским элементом для всего интерфейса. В свою очередь, на основе содержимого этого элемента, будет создана вторая часть класса MainPage. Тут будут сгенерированы такие переменные как LayoutRoot, которые инициализируются во время выполнения приложения и загрузки XAML. Имя генерируемого класса задается с помощью атрибута **x:Class**.

Рассматривая элемент **UserControl**, Вы можете обнаружить четыре объявления пространств имен. Первое имя, определенное как имя по умолчанию, подключает все стандартные элементы управления и является обязательным. Второе пространство имен **x** определяет несколько служебных атрибутов, таких как **Class** и **Name**. Эти атрибуты исключаются из XAML во время компиляции, но используются при генерации частичного класса.

Следующие два пространства имен особо и не нужны. Первое из этих двух пространств имен определяет атрибуты, которые использует только визуальный дизайнер Visual Studio и Expression Blend. Яркий пример этому — уже задекларированные атрибуты **DesignHeight** и **DesignWidth**. Удалив эти атрибуты, интерфейс приложения в дизайнере Visual Studio тут же будет ужат до одной точки. Это сделает сложным добавление элементов на поверхность с помощью Drag & Drop, но никак не повлияет на работу приложения. Второе же пространство имен используется для определения атрибута **Ignorable**, обеспечивающего игнорирование пространства имен **d**. Обычно я удаляю оба пространства имен, преведя код к следующему виду:

```
<UserControl x:Class="Chapter3_SilverlightTest.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <Grid x:Name="LayoutRoot" Background="White">

    </Grid>
</UserControl>
```

Последнее, на что нужно обратить внимание в нашем коде, — это определение элемента **Grid**. Дело в том, что элемент **UserControl** способен являться контейнером только для одного элемента. Поэтому обычно этим элементом является один из специальных контейнеров, обеспечивающих компоновку многих элементов. Элемент **Grid** является самым развитым из таких контейнеров. При необходимости его можно заменить на любой другой.

Последние два файла, которые входят в Silverlight-проект, — это **AssemblyInfo.cs** и **AppManifest.xml**. Первый файл нам не интересен, так как при-

сутствует во всех типах проектов и позволяет задавать различные атрибуты, применимые к сборкам (версию, культуру и т. д.). А вот второй файл характерен только для Silverlight-приложений. Тут задается вся структура нашего Silverlight-приложения: сборки, ресурсы, xaml файлы. Именно манифест приложения ищет встраиваемый компонент Silverlight, когда загружает приложение. Без манифеста не существует ни одного приложения. Большую часть манифеста генерирует Visual Studio, но Вы можете добавлять в него свои элементы.

Чтобы понять структуру конечного приложения и манифеста, перейдем к следующему разделу.

Развертывание приложения

Итак, чтобы Silverlight-приложение заработало, необходимо выполнить два действия:

- откомпилировать и упаковать приложение в пакет;
- встроить Silverlight элемент на любую Web-страницуи передать ему в качестве параметров путь к созданному пакету.

Естественно, что первое действие за нас выполняет Visual Studio. Используйте команду **Build->Build Solution**, чтобы получить готовое приложение. Затем перейдите в директорию приложения и в папках Bin->Debug или Bin->Release найдите созданный пакет, который будет иметь расширение **.xap**.

Несмотря на странное расширение, **.xap** файл — не что иное, как обычный ZIP-архив. Переименуем полученный **.xap** в **.zip** и попробуйте открыть. Структура архива будет выглядеть следующим образом (рис. 3.2).

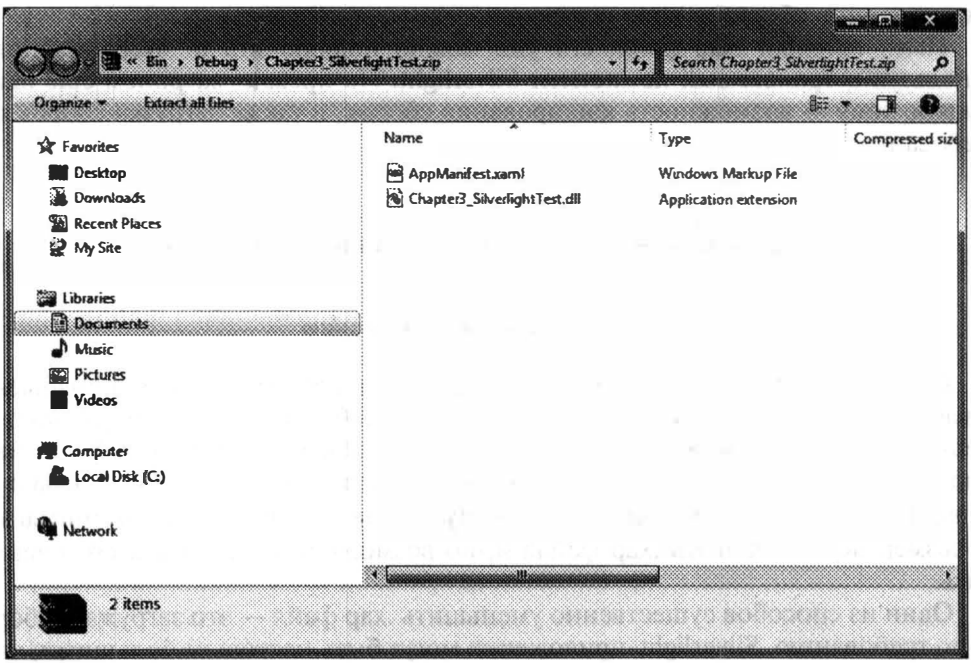


Рис. 3.2. Структура .xap-файла

Как видно, .xap-файл содержит сборку и манифест. Манифест описывает все сборки, входящие в приложение, а также тот класс, наследник от **Application**, с которого должна начаться работа приложения.

AppManifest.xaml

```
<Deployment
   xmlns="http://schemas.microsoft.com/client/2007/deployment"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   EntryPointAssembly="Chapter3_SilverlightTest"
   EntryPointType="Chapter3_SilverlightTest.App"
   RuntimeVersion="4.0.50204.0">
  <Deployment.Parts>
    <AssemblyPart x:Name="Chapter3_SilverlightTest"
       Source="Chapter3_SilverlightTest.dll" />
  </Deployment.Parts>
</Deployment>
```

Родительский элемент тут — это **Deployment**. Здесь присутствуют следующие атрибуты:

- **EntryPointAssembly** — указывается имя сборки, которая содержит класс приложения, наследник от **Application**;
- **EntryPointType** — это свойство содержит имя класса приложения;
- **RuntimeVersion** — тут хранится версия Silverlight, которую использует приложение.

В свою очередь, элемент **Deployment** содержит раздел **Deployment.Parts**, который описывает все составляющие пакета. Как видно, тут одна сборка. Выше я говорил, что в пакете поставляются и XAML файлы. Это так, но они встраиваются внутрь сборки в виде ресурсов.

Имея готовый **.xap**-файл, его уже можно подключить в Web-страницу, используя встраиваемый компонент Silverlight. Но прежде, затронем еще одну важную тему — возможность кэширования сборок, а также загрузка сборок по гребованию.

Кэширование сборок и загрузка по требованию

Загрузка сборки по требованию

С помощью Silverlight Вы можете разработать настолько функциональные и насыщенные приложения, что пользователь забудет о том, что работает с Web-приложением. Между тем, Вы не должны забывать о том, что большинство Ваших клиентов будут загружать приложения через Интернет и могут обладать небольшими каналами. Поэтому всегда нужно обращать внимание на размер получившегося .xap файла и, по возможности, пытаться его уменьшить.

Один из способов существенно уменьшить .xap файл — это загружать сборки по требованию. Silverlight-приложения могут быть настолько большими, что к некоторому функционалу пользователь доберется не сразу. При этом, выпол-

няя первую загрузку приложения без лишних проволочек, Вы оставите хорошее первое впечатление.

Чтобы продемонстрировать загрузку сборки по требованию, добавим в созданное выше решение новый проект на основе шаблона **Silverlight Class Library** (рис. 3.3).

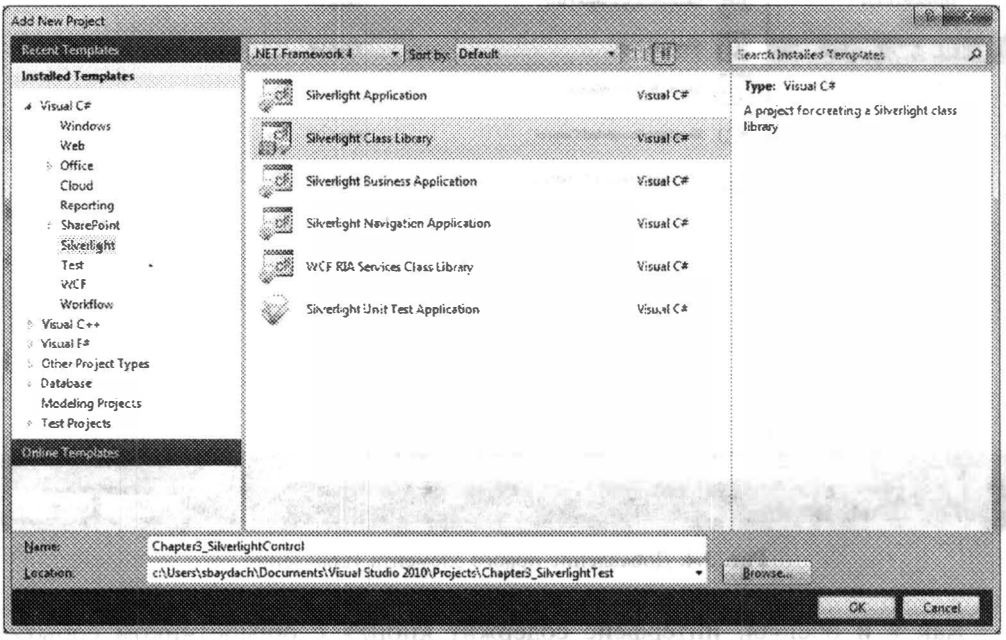


Рис. 3.3. Добавление нового проекта в решение

Внутри нового проекта будет создан файл Class1.cs, который нужно сразу же удалить.

Реализуем в новом проекте простейший интерфейс, содержащий кнопку и текстовое поле. Созданный элемент управления мы будем загружать по требованию и добавлять в интерфейс основного приложения.

Чтобы добавить в созданный проект новый элемент управления, добавьте новый элемент на основе шаблона **Silverlight User Control** (рис. 3.4).

Как видите, созданный набор файлов полностью соответствует файлам, описывающим MainPage в основном проекте. Только в этот раз наш проект не содержит классы приложения.

Перепишем интерфейс нашего элемента следующим образом:

```
<UserControl x:Class="Chapter3_SilverlightControl.MyControl"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <StackPanel x:Name="LayoutRoot" Background="White">
        <Button Content="Say Hello" Click="Button_Click"></Button>
        <TextBlock Name="helloText"></TextBlock>
    </StackPanel>
</UserControl>
```

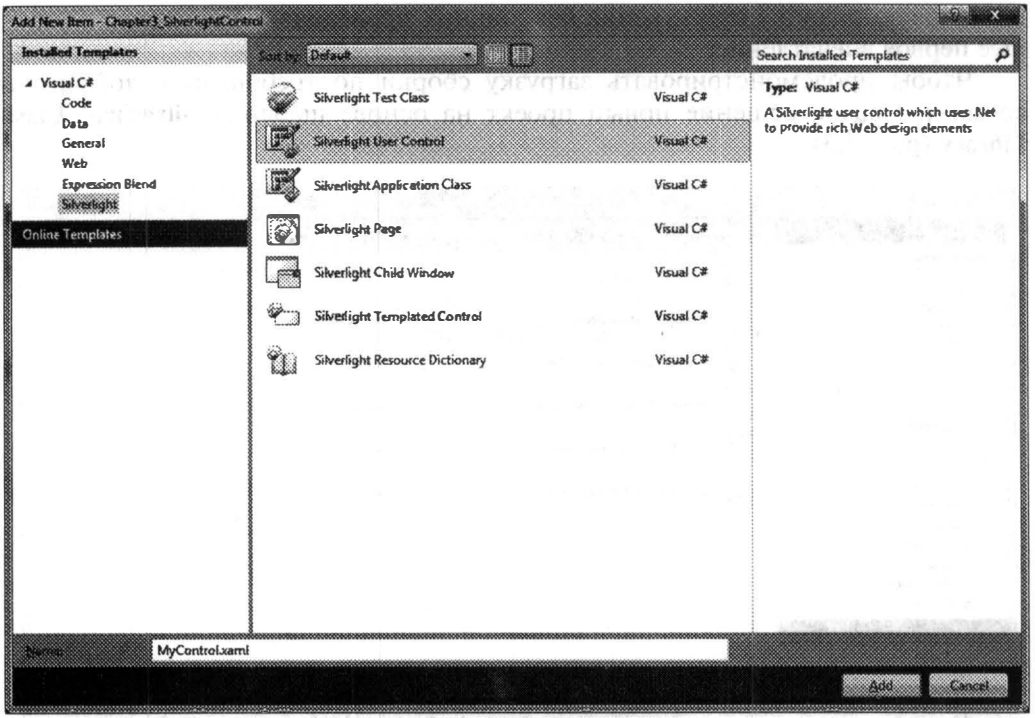


Рис. 3.4. Добавление нового элемента управления

Как я и обещал, интерфейс содержит кнопку с обработчиком события **Click**, а также текстовое поле, куда мы будем выдавать сообщение при нажатии на кнопку.

Вот как реализован обработчик события:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    helloText.Text = "Hello";
}
```

Тут мы просто устанавливаем значение нашего текстового поля.

Наш элемент управления готов. Теперь, чтобы основной проект смог использовать его, щелкните на заголовке основного Silverlight-приложения и выберите **Add Reference**. После добавления сборки в приложение, установите у нее свойство **Copy Local** в **False**. Это действие не??? позволит предотвратить размещение сборки в нашем пакете, но позволит использовать ее в нашем приложении (установив все необходимые ссылки в основной сборке) (рис. 3.5).

Теперь поработаем над основным приложением. Сначала реализуем простой интерфейс:

```
<UserControl x:Class="Chapter3_SilverlightTest.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <StackPanel x:Name="LayoutRoot" Background="White">
```

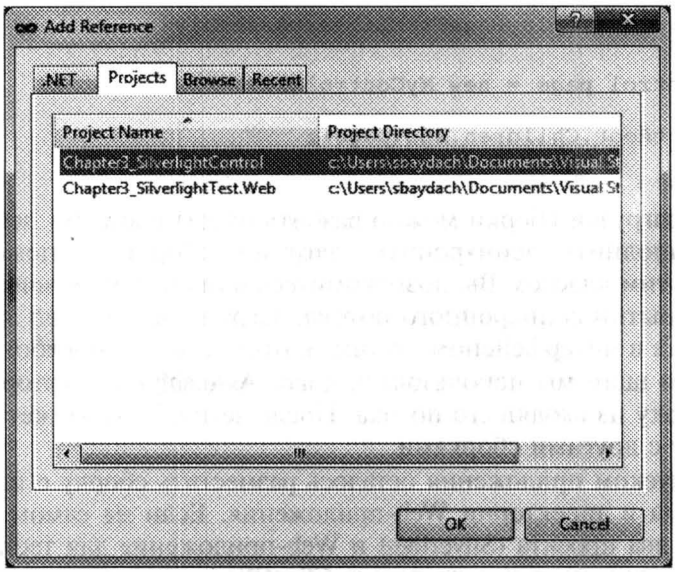


Рис. 3.5. Установка ссылки на проект с созданным элементом

```
        <Button Content="Load Assembly"
Click="Button_Click"></Button>
    </StackPanel>
</UserControl>
```

Как видно, тут всего одна кнопка, которая и будет загружать наш элемент.

А вот обработчик события **Click**, вместе с вспомогательными методами выглядит более внушительно:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    WebClient wc = new WebClient();
    wc.OpenReadCompleted +=
        new OpenReadCompletedEventHandler(wc_OpenReadCompleted);
    wc.OpenReadAsync(
        new Uri("Chapter3_SilverlightControl.dll",
           UriKind.Relative));
}

private void wc_OpenReadCompleted(
    object sender, OpenReadCompletedEventArgs e)
{
    if ((e.Error == null) && (e.Cancelled == false))
    {
        AssemblyPart assemblyPart = new AssemblyPart();
        assemblyPart.Load(e.Result);

        DisplayPageFromLibraryAssembly();
```

```
private void DisplayPageFromLibraryAssembly()
{
    MyControl page = new MyControl();

    LayoutRoot.Children.Add(page);
}
```

Процесс загрузки сборки можно разбить на два шага. На первом шаге необходимо выполнить асинхронную загрузку сборки с помощью класса **WebClient**. С этим классом Вы познакомитесь в главе 7. Фактически его задача состоит в открытии асинхронного потока, загрузке данных по заданному **Uri**, и вызов метода в интерфейсном потоке, который может обработать данные.

На втором шаге мы использовали класс **AssemblyPart**, который позволяет загрузить сборку из входящего потока. После чего сборка может быть использована наряду с другими сборками.

Перед запуском приложения осталось разместить сборку с нашим элементом управления в директории Web-приложения. Если на самом первом этапе Вы создавали два проекта (Silverlight и Web-приложение для тестирования), то сборку нужно разместить в директорию **ClientBin** нашего приложения. Если Вы решили не создавать Web-приложение, то сборку необходимо скопировать в директорию Debug или Release, в зависимости от настроек текущей компиляции.

Кэширование сборки

Чтобы упростить жизнь конечному пользователю, в Silverlight существует возможность разбить **.xap** пакет на несколько файлов. Это нужно в первую очередь для того, чтобы обеспечить механизм кэширования. Хочу сразу сказать, что Silverlight не предлагает каких-то механизмов хранения сборок в кэше или расширения ядра. Хотя, если браузер пользователя настроен таким образом, что кэширует данные с сервера, то **.xap** файл также будет кэшироваться и не загружаться при следующей попытке запустить приложение. Однако, если по каким-то причинам Вы часто модифицируете приложение, то пользователь будет вынужден загружать всякий раз новый **.xap**. Поэтому совершенно логично разбить **.xap** файл на несколько частей, позволив загружать пользователю лишь те сборки, которые претерпели изменение.

Чтобы реализовать механизм выделение сборок из **.xap** файла, необходимо выполнить два шага.

Первый шаг состоит в описании специального XML файла, который содержит информацию о сборке и архиве, куда помещается сборка. Имя этого файла должно совпадать с именем сборки и иметь расширение **extmap.xml**. Вот пример такого файла для сборки **System.Windows.Controls.dll**.

```
<?xml version="1.0"?>
<manifest xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <assembly>
    <name>System.Windows.Controls</name>
    <version>2.0.5.0</version>
    <publickeytoken>31bf3856ad364e35</publickeytoken>
```

```
    <relpath>System.Windows.Controls.dll</relpath>
    <extension downloadUri="System.Windows.Controls.zip" />
  </assembly>

</manifest>
```

Эти данные используются для поиска файла в кэше и для определения его актуальности.

На втором этапе необходимо включить механизм выделения сборок из **.xap** файла в Visual Studio. Для этого перейдите на страницу свойств проекта и установите флажок **Reduce XAP size by using application library caching**.

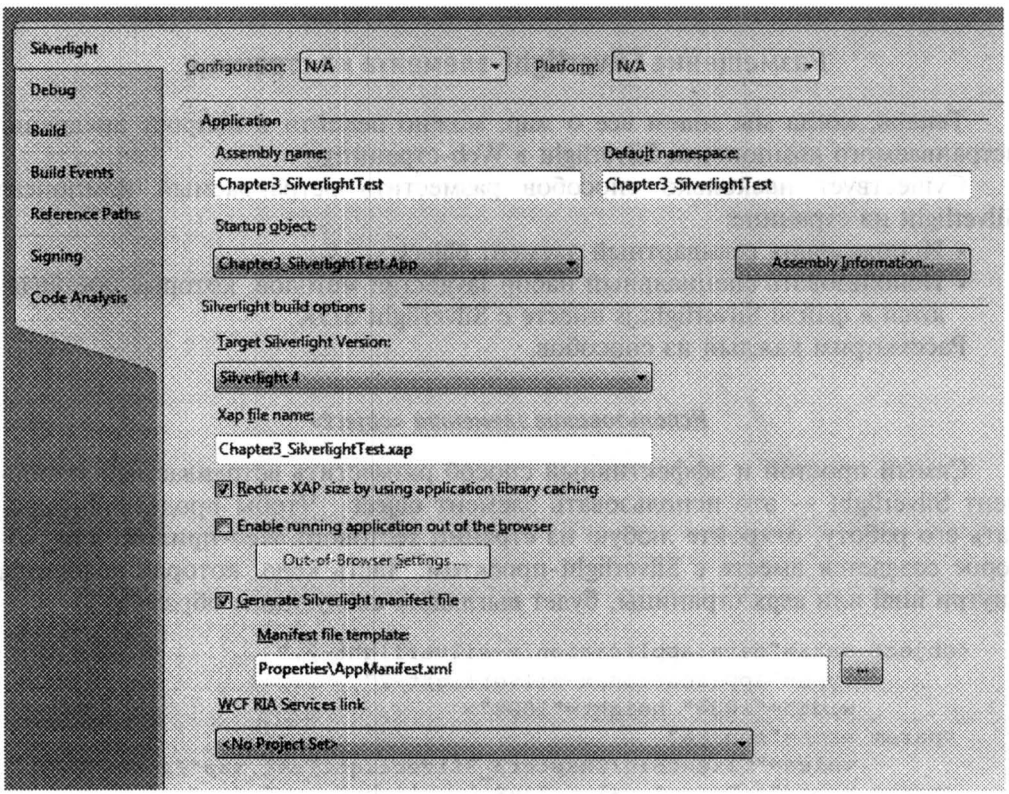


Рис. 3.6. Установка свойств проекта

Теперь можно добавлять и саму сборку.

Если все сделано правильно, то при добавлении новой сборки в директории приложения появится соответствующий **.zip** файл, а в манифесте появится новый элемент **ExtensionPart**:

```
<Deployment
xmlns="http://schemas.microsoft.com/client/2007/deployment"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   EntryPointAssembly="Chapter3_SilverlightTest"
   EntryPointType="Chapter3_SilverlightTest.App"
   RuntimeVersion="4.0.50204.0">
```

```
  <Deployment.Parts>
    <AssemblyPart x:Name="Chapter3_SilverlightTest"
      Source="Chapter3_SilverlightTest.dll" />
  </Deployment.Parts>
  <Deployment.ExternalParts>
    <ExtensionPart Source="System.Xml.Linq.zip" />
  </Deployment.ExternalParts>
</Deployment>
```

На этом и закончим описание работы с **.xap** файлами и перейдем к вопросу о том, как пакеты интегрируются в Web-страницы существующих приложений.

Размещение Silverlight-элемента на странице

Теперь, когда мы знаем все о .xap, можно перейти к вопросу внедрения встраиваемого компонента Silverlight в Web-страницу.

Существует несколько способов разместить встраиваемый компонент Silverlight на странице:

- Использовать стандартный элемент **object**;
- Использовать специальный набор JavaScript методов, которые поставляются в файле Silverlight.js вместе с Silverlight SDK;

Рассмотрим каждый из способов.

Использование элемента <object>

Самый простой и эффективный способ разместить встраиваемый компонент Silverlight — это использовать элемент **object** . Чтобы продемонстрировать его работу, откройте любую из страниц тестового Web-приложения, которое создается вместе с Silverlight-проектом. Часть кода, которая находится внутри **html** или **aspx** страницы, будет выглядеть следующим образом:

```
<object data="data:application/x-silverlight-2,"
        type="application/x-silverlight-2"
        width="100%" height="100%">
 <param name="source"
        value="ClientBin/Chapter3_SilverlightTest.xap"/>
 <param name="onError" value="onSilverlightError" />
 <param name="background" value="white" />
 <param name="minRuntimeVersion" value="4.0.50204.0" />
 <param name="autoUpgrade" value="true" />
 <a href="http://go.microsoft.com/fwlink/?LinkID=149156&v=4.0.50204.0"
        style="text-decoration:none">
        <img src="http://go.microsoft.com/fwlink/?LinkId=161376"
           alt="Get Microsoft Silverlight"
style="border-style:none"/>
 </a>
</object>
```

Итак, описывая элемент **object**, разработчик должен указать тип элемента (Silverlight-элемент не ниже второй версии), а также размеры. Атрибуты **Width**

и **Height** достаточно важны, так как фактически они определяют размер Silverlight-окна. Если приложение написано правильно, то размер Silverlight-приложения подстраивается под размеры, выделенные встраиваемому компоненту. Отсюда следует одно важное правило: старайтесь не указывать явные размеры элементов управления в Silverlight-приложениях.

Следом за основными свойствами **object** идут параметры. Первый и самый важный параметр — это **source**. Именно он позволяет задавать ссылку на **.xap** файл, который и будет загружать встраиваемый компонент Silverlight.

Следующий не менее важный параметр — это **onError**. Этот параметр позволяет указать ссылку на JavaScript метод, обрабатывающий ошибки, выброшенные встраиваемым элементом или управляемым кодом (которые не были обработаны в управляемом коде). Обычно метод, который генерируется для обработки ошибок, просто собирает информацию об ошибке и выдает ее на экран. Вот код этого метода:

```
function onSilverlightError(sender, args)
{
    var appSource = "";

    if (sender != null && sender != 0) {
      appSource = sender.getHost().Source;
    }

    var errorType = args.ErrorType;
    var iErrorCode = args.ErrorCode;

    if (errorType == "ImageError" || errorType == "MediaError") {
      return;
    }

    var errMsg =
       "Unhandled Error in Silverlight Application " +
       appSource + "\n" ;

    errMsg += "Code: "+ iErrorCode + "    \n";
    errMsg += "Category: " + errorType + "       \n";
    errMsg += "Message: " + args.ErrorMessage + "     \n";

    if (errorType == "ParserError") {
        errMsg += "File: " + args.xamlFile + "     \n";
        errMsg += "Line: " + args.lineNumber + "     \n";
        errMsg += "Position: " + args.charPosition + "     \n";
    }
    else if (errorType == "RuntimeError") {
        if (args.lineNumber != 0) {
            errMsg += "Line: " + args.lineNumber + "     \n";
            errMsg += "Position: " +  args.charPosition + "
\n";
        }
        errMsg += "MethodName: " + args.methodName + "     \n";
    }

    throw new Error(errMsg);
```

Если Вы не реализуете обработчик ошибок, то пользователь не будет получать никакой нотификации об ошибке, но вместо интерфейса приложения сможет увидеть только серый экран.

Оставшиеся параметры задают фон встраиваемого элемента Silverlight, минимальную версию, необходимую для запуска и необходимость автоматического обновления компонента в случае более старой версии. Хочу отметить, что фон встраиваемого компонента Silverlight может быть полезен лишь в том случае, когда Silverlight-приложение меньше самого компонента.

Следом за параметрами следует html-код, который будет отображен в случае, если встраиваемый компонент Silverlight отсутствует на машине пользователя. Рекомендую тут заменить стандартную иконку со ссылкой на инсталляцию Silverlight собственным сообщением, содержащим скриншоты Вашего приложения и объяснением преимуществ Silverlight в доступной для пользователя форме.

Еще один параметр, о котором также стоит вспомнить, — это **initParams**. Благодаря этому параметру можно передать в Silverlight-приложение любое количество параметров. Для этого в значение **initParams** устанавливается текстовая строка, где все параметры перечисляются через запятую в формате **<имя=значение>**.

Получить переданные параметры можно в любом месте Silverlight-приложения, используя ссылку на объект **Application**. Вот пример такого кода:

```
curList = App.Current.Host.InitParams["media"];
```

Элемент **object** может принимать еще несколько параметров, которые будут рассмотрены по ходу книги.

Разобравшись с элементом object, перейдем к механизму вставки с помощью JavaScript.

Немного о классах в JavaScript

Прежде чем рассматривать вопрос использования **SilverLight.js** рассмотрим несколько конструкций в JavaScript, позволяющих определить классы. Обычно не многие программисты используют эти конструкции в своих приложениях, поэтому остановимся на них подробнее.

Начнем с того, что JavaScript позволяет создавать классы. Этот факт не позволяет говорить о JavaScript как об объектно-ориентированном языке, так как классы не поддерживают полиморфизм и наследование. Все же наличие специальных конструкций позволяет реализовывать некоторые принципы инкапсуляции и группировать данные и функции, что упрощает задачу создания библиотек на JavaScript.

Ниже показан код, который объявляет класс и создает экземпляр этого класса.

CreateClass.html

```
<html>
<head>
<script language="javascript" type="text/javascript">
```

```
  if (!window.MyLib)
  window.MyLib={};

  MyLib.MyClass= function()
  {
  alert("Constructor");
  }

  new MyLib.MyClass();

</script>
</head>
<body>
</body>
</html>
```

Синтаксическая конструкция достаточна проста. Первым выражением определяется своеобразный контейнер для классов, а вторым — сам класс, который состоит из конструктора, не принимающего параметров. Для создания экземпляра класса используется оператор **new** (что не удивительно).

Расширим нашу конструкцию, добавив методы в класс.

CreateClass.html

```
<html>
<head>
<script language="javascript" type="text/javascript">

  if (!window.MyLib)
  window.MyLib={};

  MyLib.MyClass= function()
  {
  alert("Constructor");
  }

  MyLib.MyClass.prototype=
  {
  DoAlert: function()
  {
        alert("Hello");
  },

  DoAlertWithParam: function(param)
  {
        alert("Hello "+param);
  }
  }

  var mcl=new MyLib.MyClass();
  mcl.DoAlert();
  mcl.DoAlertWithParam("Sergey");

</script>
</head>
```

```
<body>
</body>
</html>
```

Этот код говорит сам за себя. Для определения функций класса необходимо объявить дополнительный раздел **prototype** и включить туда все функции, разделив их запятой.

Именно понятие классов в JavaScript и положено в основу **Silverlight.js**.

Использование Silverlight.js

Использование **object** достаточно эффективно, но если есть необходимость создавать объект с помощью JavaScript, то придется написать достаточно большой блок кода. Однако Microsoft выполнила всю работу за разработчика, создав **Silverlight.js**, позволяющий создавать встраиваемый компонент Silverlight, вызвав уже готовые методы.

Чтобы использовать Silverlight.js необходимо добавить его в Ваш проект. Для этого нужно обратиться по следующему пути **C:\Program Files (x86)\Microsoft SDKs\Silverlight\v4.0\Tools**. Именно тут и находится реализация Silverlight.js. Рассмотрим структуру этого файла.

Файл **Silverlight.js** содержит объявление контейнера **Silverlight** и объявление нескольких классов, таких как **createObject** и **createObjectEx** (далее будем называть конструкторы этих классов функциями с соответствующими именами). Таким образом, при реализации **Silverlight.js** были использованы возможности JavaScript, описанные в предыдущем разделе.

Замечание. В данном случае использование контейнера Silverlight выглядит как использование класса, а использование конструкторов классов **createObject** и **createObjectEx** выглядит как вызов статических методов. Это абсолютно нормально. Поскольку в JavaScript как таковых конструкторов нет, то при небольшом размере контейнера можно упростить работу с ним, заключив каждый метод в конструктор одноименного класса.

Рассмотрим пример использования **Silverlight.js**:

Default.html

```
<html>
<head>
   <title>CreateObject test</title>
   <script type="text/javascript" src="Silverlight.js"></script>
</head>
<body>
<div id="silverlightControlHost">
   <script type="text/javascript">
      Silverlight.createObjectEx(
      {
         source: "SLApp.xap",
         parentElement:
document.getElementById("silverlightControlHost"),
         id: "SilverlightControl",
```

```
            properties:
            {
               width: "100%",
               height: "100%",
               version: "4.0.41108.0"
            },
            events:
            {
               onLoad: null,
               onError: null
            }
         }
         );
      </script>
   </div>
   </body>
   </html>
```

В данном примере мы подключили скрипт **SilverLight.js**, взятый из SDK и использовали функцию **createObjectEx** для создания встраиваемого объекта и запуска приложения. Кроме метода **createObjectEx** можно использовать метод **createObject**, который даст тот же эффект, но имеет другую нотацию в записи.

Рассмотрим параметры метода **createObjectEx**. Если вам нравится аналогичный метод без суффикса, то к описанию его параметров можно обратиться в SDK. Ниже приводится полная сигнатура метода **createObjectEx**.

```
Silverlight.createObjectEx(
{
     source: 'SLApp.xap',
     parentElement:parentElement,
     id:'myPlugin',
     properties:
     {
         width:'100',
         height:'100',
         inplaceInstallPrompt:false,
         background:'white',
         isWindowless:'false',
         framerate:'24',
         version:'1.0'
     },
     events:
     {
         onError:null,
         onLoad:null
     },
     initParams:null,
     context:null
```

Итак, первый параметр ссылается на Silverlight-приложение, которое будет загружено во встраиваемый компонент. Второй параметр содержит id элемента, в который встраивается Silverlight-компонент, обычно это **div** элемент. Третий параметр задает id для встраиваемого компонента, через который к нему можно обращаться из других частей JavaScript кода. Следующая группа параметров задает свойства встраиваемого компонента, такие как длина, ширина и др. Параметр **events** определяет JavaScript методы, которые вызываются в случае ошибки и после загрузки компонента. Параметр **initParams** задает список параметров, передаваемых Silverlight-приложению.

Вот и все, что можно рассказать о вставке встраиваемого компонента Silverlight в контекст Web-страницы. Оба способа не зависят от самой Web-страницы, то есть Silverlight-приложение легко внедрить в html, php, asp.net или любое другое Web-приложение.

Анимация во время загрузки

Выше мы рассматривали, каким образом уменьшить размер пакета и сделать ожидание пользователя во время загрузки Silverlight-приложения как можно менее длительным. Между тем, часто не удается уменьшить размер пакета до желаемого, а если пользователь имеет медленный канал, то размер пакета в 1 мб может вызвать у пользователя неприятные ощущения. Поэтому, если не удается сделать приложение небольшого размера, то следует позаботиться хотя бы о том, чтобы пользователь получал информацию о текущем состоянии загрузки пакета.

Информировать пользователя о состоянии загрузки пакета можно благодаря дополнительным параметрам, реализованным во встраиваемом компоненте Silverlight. Тут есть следующие параметры:

- **SplashScreenSource** — этот параметр позволяет указать имя XAML файла, который будет загружен и инициализирован перед загрузкой основного пакета;
- **OnSourceDownloadProgressChanged** — этот параметр позволяет задать имя JavaScript метода, реагирующего на изменения процента загрузки основного пакета;
- **OnSourceDownloadComplete** — этот параметр позволяет задать JavaScript метод, который будет вызван после окончания загрузки основного пакета приложения.

Итак, **SplashScreenSource** позволяет задать имя XAML файла. Тут нельзя указывать **XAP** файл, да и XAML должен соответствовать формату Silverlight 1.0. Несмотря на это, тут доступна анимация, основные графические объекты и даже есть возможность отображать видео.

Если Вы хотите отобразить более сложное приложение, то тут **SplashScreenSource** не подходит. Вместо этого Вы можете загрузить **XAP** файл со своим приложением, установив свойство **Source**, после чего, используя объект **Downloader** (JavaScript), загрузить основной **XAP** файл и поменять свойство **Source** у встраиваемого компонента Silverlight. Но на практике такой необходимости не возникает.

Итак, чтобы продемонстрировать работу **SplashScreenSource**, создадим новый проект Silverlight и добавим в него несколько файлов с видео. После этого запустим приложение. Несмотря на то, что **SpashScreenSource** не установлен, Silverlight компонент отображает стандартную анимацию, показывая сколько процентов осталось до конца загрузки.

Рис. 3.7. Стандартная анимация в Silverlight

Попробуем реализовать свою анимацию. Для этого добавим в Web-приложение, содержащее страницы для тестирования, элемент, соответствующий шаблону **Silverlight 1.0 Jscript Page**.

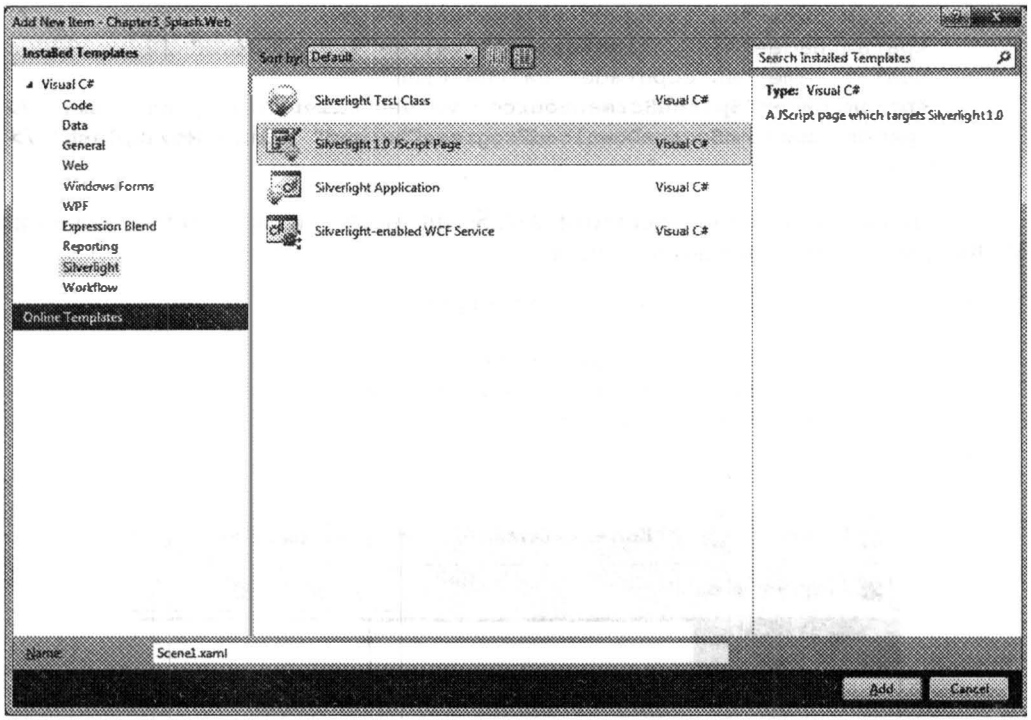


Рис. 3.8. Создание страницы с анимацией

Перетяните созданную страницу в папку ClientBin (куда копируется пакет с Silverlight-приложением) и установите свойство **Build Action** в **None**. Мы не планируем компилировать эту страницу, нам ее достаточно разместить в директории приложения.

Реализуем содержимое созданного XAML следующим образом:

```
<Canvas Background="Black" Width="302" Height="32"
   xmlns=http://schemas.microsoft.com/client/2007
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 >
  <Rectangle Canvas.Left="1" Canvas.Top="1"
      Width="300" Height="30" Fill="White"/>
  <Rectangle x:Name="progressBarFill" Canvas.Left="1"
      Canvas.Top="1" Height="30" Fill="Red"/>
</Canvas>
```

Этот код определяет два прямоугольника. Второй (закрашенный прямоугольник) будет заполнять первый по мере загрузки пакета.

Теперь следует перейти к коду страницы html или aspx и модифицировать элемент **object** следующим образом:

```
<object data="data:application/x-silverlight-2,"
   type="application/x-silverlight-2" width="100%" height="100%">
   <param name="source" value="ClientBin/Chapter3_Splash.xap"/>
   <param name="onError" value="onSilverlightError" />
   <param name="background" value="white" />
   <param name="minRuntimeVersion" value="4.0.50204.0" />
   <param name="autoUpgrade" value="true" />
   <param name="SplashScreenSource" value="ClientBin/Splash.xaml" />
   <param name="onSourceDownloadProgressChanged" value="MoveSplash" />
</object>
```

На последнем шаге реализуем JavaScript метод, изменяющий значение Width для красного прямоугольника.

```
function MoveSplash(sender, eventArgs)
{
   var slPlugin = sender.getHost();
   slPlugin.content.findName("progress").width =
       eventArgs.progress*300;
}
```

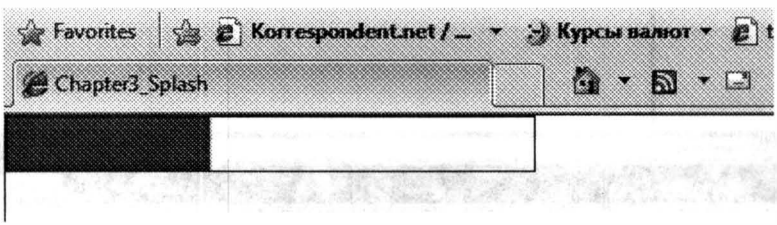


Рис. 3.9. Результат работы приложения

Запустив приложение, Вы сможете увидеть двигающийся прямоугольник:

Вот и все, что нужно знать об анимации во время загрузки. Разобравшись с графикой и анимацией в Silverlight, Вы сможете создавать более сложные «заставки».

Взаимодействие со встраиваемым элементом

С выходом первой версии Silverlight разработчик не мог и мечтать об использовании управляемого кода. Вместо управляемого кода вся логика реализовывалась на JavaScript. Это означает, что JavaScript имел механизмы не только изменять основные свойства встраиваемого компонента, но и получать доступ к дереву объектов Silverlight. Эти механизмы сохранились и сейчас.

Использование JavaScript

Итак, JavaScript имеет возможность взаимодействовать с деревом элементов Silverlight. Это возможно благодаря набору методов, которые предоставляет встраиваемый компонент Silverlight.

Прежде чем перейти к самим методам, следует отметить, что настройки встраиваемого компонента позволяют запретить доступ JavaScript к дереву объектов Silverlight. Для этого можно использовать еще один параметр для встраиваемого компонента, это **enableHtmlAccess**. По умолчанию этот параметр установлен в **true**, но если Вы хотите запретить доступ, то его можно установить в **false** с помощью элемента **param** или непосредственно в JavaScript. При этом, если Вы используете JavaScript, то все настройки, задаваемые с помощью элемента **param,** можно изменять, используя следующий синтаксис: *имя элемента.settings.имя параметра.*

```
slPlugin.settings.enableHtmlAccess = false;
```

Итак, перейдем к методам, среди которых можно выделить два основных:

- **createFromXaml** — этот метод позволяет создать объекты, готовые для вставки в Silverlight-дерево, принимая в качестве параметров текст. Естественно, что текст должен быть правильно сформированным XAML. При этом корневой элемент должен быть только один. После создания объекта метод возвращает на него ссылку. Теперь объект можно добавить в общее дерево, используя любой из контейнеров в приложении;
- **findName** — этот метод позволяет найти элемент в дереве объектов Silverlight, по заданному имени в атрибуте **x:Name**. Если элемент найден, то возвращается ссылка на него, и Вы можете приступать к изменению его свойств. Если же элемент не найден, то возвращается значение null.

Оба метода должны иметь следующий синтаксис: *имя элемента.content.имя метода.*

Продемонстрируем небольшой JavaScript код:

```
function addXAML()
{
    var slPlugin = document.getElementById("host");

    var myXAML = slPlugin.content.createFromXaml(
       '<Button Content="Hello" Height="100"
Width="100"></Button>');

    slPlugin.content.findName("LayoutRoot").children.add(myXAML);
```

Как видно, мы получаем ссылку на встраиваемый компонент (для этого задайте id у **object**), а затем вызываем нужный метод согласно синтаксису.

Данный метод создания объектов не очень эффективен по сравнению с загрузкой сборки по требованию, так как не позволяет загружать события и ресурсы, но иногда может пригодиться.

Переход в полноэкранный режим

Нужно отметить, что получать доступ к свойствам встраиваемого элемента можно и из управляемого кода. Для этого используется текущий объект приложения, через который становится доступно свойство **Host**.

Продемонстрируем этот механизм, используя возможность Silverlight-приложений переходить в полноэкранный режим. В данном режиме приложение отображается на весь экран, полностью скрывая фрейм браузера. Несмотря на то, что возможности этого режима достаточно ограничены, он очень привлекателен при отображении видео в хорошем качестве. Ограничения же полноэкранного режима не позволяют пользователю использовать клавиатуру для ввода данных (кроме функциональных клавиш, например **Esc**). Это сделано для обеспечения безопасности пользователя (например, представьте экран ОС, требующий ввести логин и пароль для входа в систему).

Итак, реализуем интерфейс приложения так, как показано ниже:

```
<UserControl x:Class="Chapter3_FullScreen.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <Grid x:Name="LayoutRoot" Background="White">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
        </Grid.RowDefinitions>
        <MediaElement Name="myMedia"  AutoPlay="True"
           Source="WildLife.wmv" Grid.Row="0"
           Width="400" Height="300"></MediaElement>
        <Button Name="fullButton" Content="Full Screen"
           Width="100" Grid.Row="1"
           Click="Full_Click"></Button>
    </Grid>
</UserControl>
```

Тут мы определили элемент управления для отображения видео и кнопку, позволяющую перейти в полноэкранный режим. Рассмотрим теперь код обработчиков событий:

```
public partial class MainPage : UserControl
{
    public MainPage()
    {
        InitializeComponent();
        App.Current.Host.Content.FullScreenChanged +=
```

```
                Full_Change;
        }

        private void Full_Click(object sender, RoutedEventArgs e)
        {
            App.Current.Host.Content.IsFullScreen = true;
        }

        private void Full_Change(object sender, EventArgs e)
        {
            if (App.Current.Host.Content.IsFullScreen)
            {
                fullButton.Visibility = Visibility.Collapsed;
                myMedia.Width =
App.Current.Host.Content.ActualWidth;
                myMedia.Height =
App.Current.Host.Content.ActualHeight;
            }
            else
            {
                fullButton.Visibility = Visibility.Visible;
                myMedia.Width = 400;
                myMedia.Height = 300;
            }
        }
    }
```

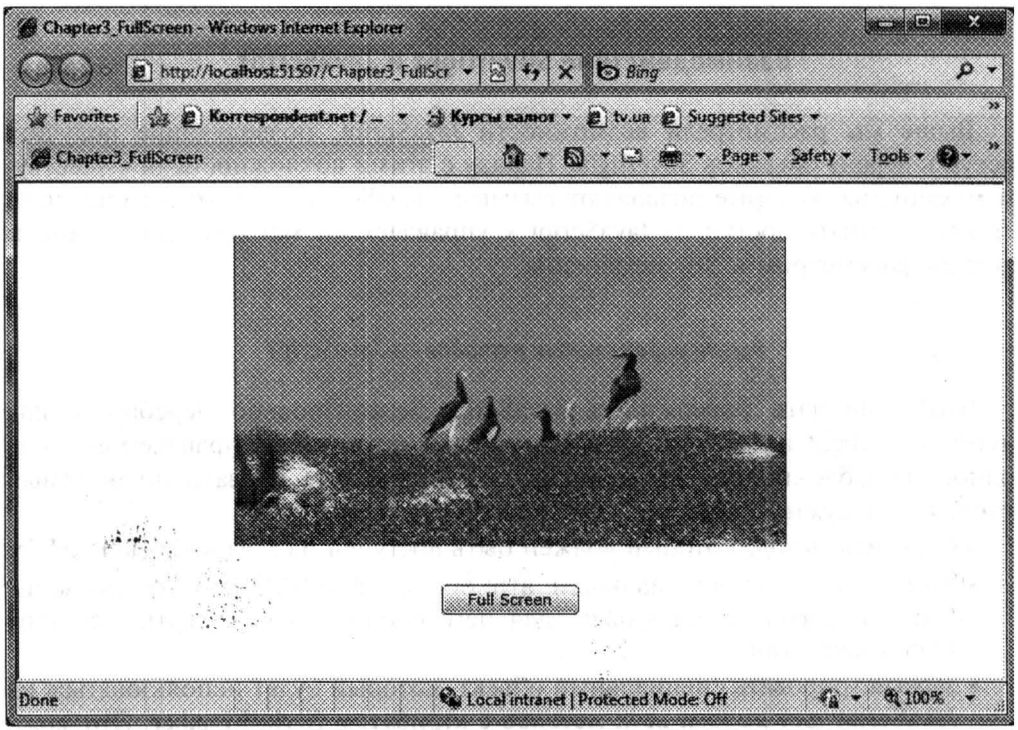


Рис. 3.10. Результат работы приложения в нормальном режиме

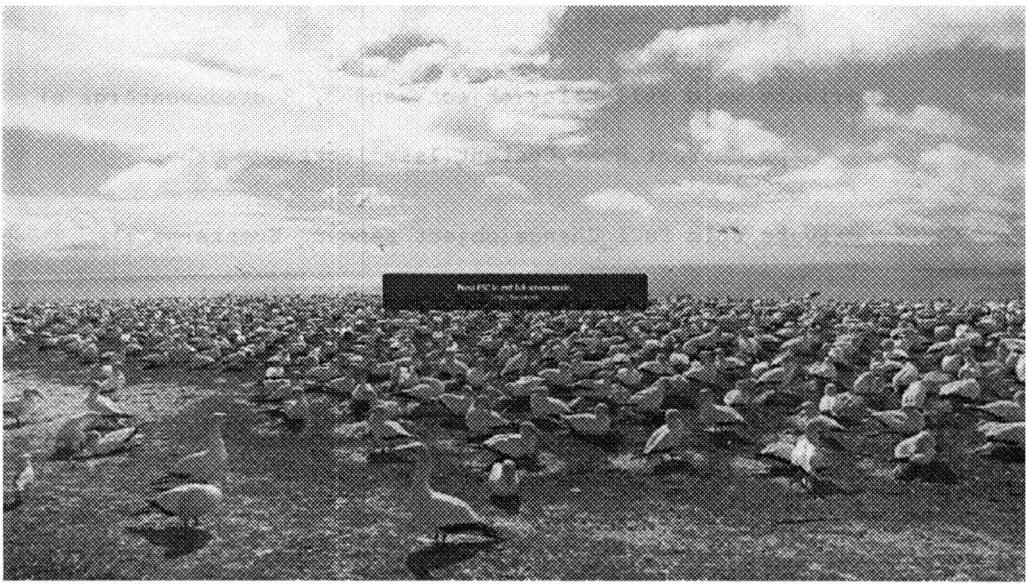


Рис. 3.11. Результат работы приложения в полноэкранном режиме

Как видно, тут мы оперируем свойством **IsFullScreen**, которое позволяет перевести наше приложение в полноэкранный режим. Аналогично можно получить доступ и к другим свойствам.

Взаимодействие Silverlight и JavaScript

Выше мы рассмотрели возможности JavaScript, позволяющие получить доступ к дереву объектов Silverlight. Наряду с этими возможностями существуют механизмы, которые позволяют вызывать JavaScript из управляемого кода, а также получать доступ из JavaScript к управляемым методам. Для полноты картины рассмотрим и эти механизмы.

Вызов управляемых методов из JavaScript

Чтобы не дать возможность JavaScript бесконтрольно перебирать код внутри Silverlight-приложения, вызов любых методов из управляемого кода полностью заблокирован. Но если Вы все-таки хотите вызвать управляемый метод, то тут нужно выполнить следующие действия:

- объявить метод, который должен быть доступен из JavaScript, как **public**;
- пометить метод специальным атрибутом **ScriptMember**, что позволит ядру Silverlight сгенерировать для него специальную обвертку, доступную в JavaScript;
- зарегистрировать специальный объект, который будет использоваться из JavaScript для вызова всех методов с атрибутом **ScriptMember**. Это можно сделать с помощью класса **HtmlPage** и метода **RegisterScriptableObject**.

Реализуем описанные шаги в новом Silverlight-приложении, содержащем ишь одно текстовое поле:

```
<UserControl x:Class="Chapter3_FullScreen.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Loaded="UserControl_Loaded">
    <Grid x:Name="LayoutRoot" Background="White">
        <TextBlock Name="tb"></TextBlock>
    </Grid>
</UserControl>

public partial class MainPage : UserControl
{
    public MainPage()
    {
        InitializeComponent();
    }

    [ScriptableMember]
    public void SetTextBlock(string text)
    {
        tb.Text = text;
    }

    private void UserControl_Loaded(object sender, RoutedEventArgs
e)
    {
        HtmlPage.RegisterScriptableObject("MyMethods", this);
    }
}
```

Как видно, мы зарегистрировали объект для доступа к управляемым методам в обработчике события **Loaded** для основного контейнера, а затем реализовали обычный метод, устанавливающий значение текстового поля.

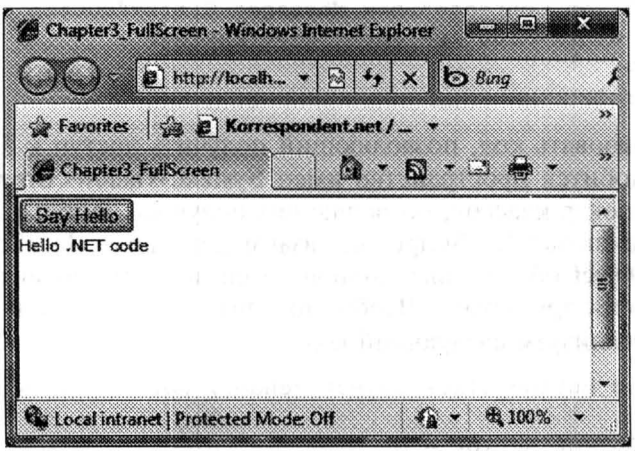


Рис. 3.12. Результат работы приложения

А вот и JavaScript, вызывающий наш метод:

```
function SayHello(alertMsg)
{
    var slPlugin = document.getElementById("host");
    slPlugin.content.MyMethods.SetTextBlock("Hello .NET code");
}
```

Как видно, доступ к созданному объекту также осуществляется через свойство **content**.

Результат работы приложения показан на рис. 3.12.

Вызов JavaScript методов из управляемого кода

Доступ к JavaScript методам из управляемого кода выглядит не таким сложным. Для того, чтобы вызвать любой JavaScript метод, вовсе нет необходимости делать какие-то изменения в JavaScript. Можно обойтись одним кодом на C#.

Рассмотрим небольшой пример. Создайте новое приложение и разместите внутри html страницы, содержащей компонент Silverlight, следующий JavaScript:

```
function ShowAlert(alertMsg)
{
    alert(alertMsg);
}
```

Это обычная функция, принимающая параметр и выдающая сообщение в виде всплывающего окна.

Реализуем интерфейс Silverlight-приложения, который будет состоять из одной кнопки:

```
<UserControl x:Class="Chapter3_FullScreen.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid x:Name="LayoutRoot" Background="White">
        <Button Content="Show Message" Click="Button_Click"
           Width="100" Height="50"></Button>
    </Grid>
</UserControl>
```

Чтобы реализовать код, позволяющий получить доступ к JavaScript методам, нам понадобится пространство имен **System.Windows.Browser**. Тут находится полный спектр классов, позволяющих получить доступ к объектной модели браузера, включая JavaScript. Для взаимодействия с JavaScript нам понадобится **ScriptObject** объект, выполняющий связывания управляемого кода и конкретного JavaScript метода. Чтобы получить этот объект и инициировать вызов метода, реализуем следующий код:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    ScriptObject script =
      (ScriptObject)HtmlPage.Window.GetProperty("ShowAlert");
```

```
        script.InvokeSelf("Hello from JScript");
    }
```

Тут мы получили ссылку на **ScriptObject** с помощью метода **GetProperty**, ассоциированного с окном Web-приложения. А затем воспользовались методом **InvokeSelf**, чтобы инициировать вызов сценария и передать ему параметры.

Результат работы приложения показан на рис. 3.13.

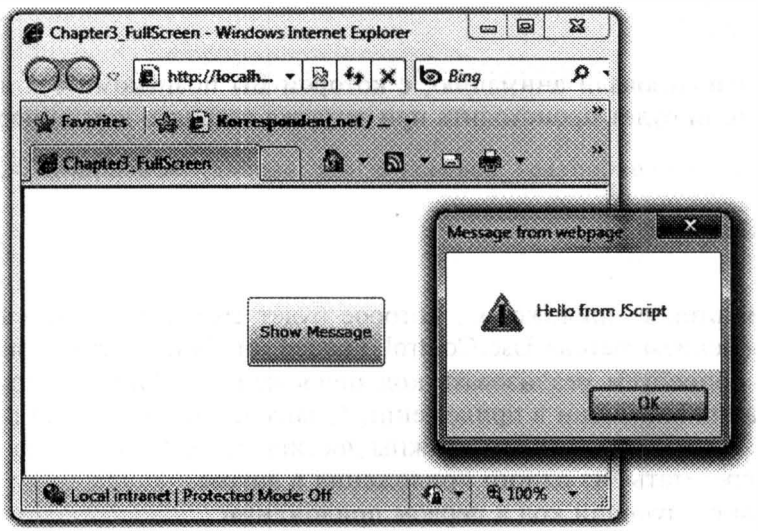


Рис. 3.13. Результат работы приложения

Взаимодействие между Silverlight-приложениями

Последняя тема в этой главе раскрывает возможности взаимодействия Silverlight-приложений друг с другом путем обмена сообщениями. Эта возможность появилась лишь в Silverlight 3, и я пока не видел ее хорошего применения.

Все начинается с пространства имен **System.Windows.Messaging**, которое и позволяет наладить взаимодействие между несколькими приложениями Silverlight.

Создадим простое приложение, которое отображает эллипс, двигающийся снизу вверх:

```
<UserControl x:Class="SilverlightApplication57.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300" Loaded="UserControl_Loaded">
    <UserControl.Resources>
        <Storyboard x:Name="sb1" Completed="sb1_Completed">
            <DoubleAnimation Storyboard.TargetName="el1"

Storyboard.TargetProperty="(Canvas.Top)"
                              From="300" To="-30" Duration="0:0:5">
```

```
            </DoubleAnimation>
        </Storyboard>
    </UserControl.Resources>
    <Canvas x:Name="LayoutRoot" Background="White">
        <Ellipse x:Name="ell" Width="30" Height="30" Fill="Blue"
           Canvas.Left="200" Canvas.Top="300">
        </Ellipse>
    </Canvas>
</UserControl>
```

Тут мы использовали анимацию, с которой Вы познакомитесь в главе 8. Запуск анимации будет происходить при загрузке главного контейнера:

```
private void UserControl_Loaded(object sender, RoutedEventArgs e)
{
    sb1.Begin();
}
```

Создадим второе приложение, которое будет содержать точно такой же код, за исключением метода UserControl_Loaded (он будет отсутствовать).

Давайте попробуем реализовать код приложений таким образом, чтобы после окончания анимации в приложении 1, запускалась анимация во втором приложении. Таким образом, мы должны достигнуть эффекта, когда наш шарик будет «перелетать» из одного приложения в другое.

Реализуем следующий код в первом приложении:

```
LocalMessageSender msgSender = new LocalMessageSender("DownSide");

public MainPage()
{
    InitializeComponent();
    LocalMessageReceiver receiver = new
LocalMessageReceiver("UpSide");
    receiver.MessageReceived +=
       new
EventHandler<MessageReceivedEventArgs>(receiver_MessageReceived);
    receiver.Listen();
}

void receiver_MessageReceived(object sender,
MessageReceivedEventArgs e)
{
    sb1.Begin();
}

private void sb1_Completed(object sender, EventArgs e)
{
    msgSender.SendAsync("start");
}
```

Тут мы создали объект типа **LocalMessageSender**, в задачи которого входит отправка сообщения «слушателю» с указанным именем (DownSide). Этот ме-

год отправляет сообщение сразу после окончания анимации с помощью метода **SendAsync** (фактически передает управление другому приложению).

Объект **LocalMessageReceiver**, напротив, ожидает сообщение от других приложений. Как только сообщение приходит (второе приложение закончило анимацию), то тут же анимация запускается повторно (полетел еще один шарик).

Код во втором приложении выглядит аналогично:

```
LocalMessageSender msgSender = new LocalMessageSender("UpSide");
public MainPage()
{
    InitializeComponent();
    LocalMessageReceiver receiver = new
LocalMessageReceiver("DownSide");
    receiver.MessageReceived += new
EventHandler<MessageReceivedEventArgs>(receiver_MessageReceived);
    receiver.Listen();
}
void receiver_MessageReceived(object sender,
MessageReceivedEventArgs e)
{
    sb1.Begin();
}
private void sb1_Completed(object sender, EventArgs e)
{
    msgSender.SendAsync("start");
}
```

Разместим созданные приложения на HTML странице следующим образом (рис. 3.14).

Рис. 3.14.

Откомпилировав и запустив данный пример, мы получим желаемый эффект (рис. 3.15).

Рис. 3.15.

Вот теперь Вы знаете о работе с Silverlight-приложениями практически все и можно переходить к особенностям кодирования.

Заключение

Слишком много тем было рассмотрено в этой главе, но лишь они отделяют нас от начала кодирования. В любом случае, теперь Вы знаете, как устроено Silverlight-приложение, и как встраивать компонент Silverlight внутрь Web-страницы. Кроме того, мы рассмотрели вопросы взаимодействия JavaScript и Silverlight. Несмотря на то, что Silverlight 4 обычно не требует использование JavaScript, последний является мостиком между DOM и Silverlight-приложением. Использовать этот мостик бывает необходимо, когда часть логики приложения уже написана с использованием JavaScript, а Silverlight-приложение встраивается на более позднем этапе. Между тем, следует избегать связывать работу Silverlight-приложения с JavaScript. Ведь использование JavaScript ставит крест на использовании приложения вне браузера, кроме этого, такое приложение сложнее переносить.

Глава 4

ИСПОЛЬЗОВАНИЕ XAML

Введение в XAML

Как Вы смогли заметить из предыдущих глав, Silverlight использует специальный язык для описания интерфейса — XAML.

XAML (eXtensible Application Markup Language) представляет собой декларативный язык, построенный на базе XML. Основное назначение этого языка состоит в описании векторного интерфейса приложения. Хотя XAML также применяется и в технологии Workflow Foundation для описания рабочих процессов.

Чтобы лучше понять назначение XAML, достаточно вспомнить, как создавались интерфейсы на языке программирования C#. Так, кусок кода, создающий небольшую кнопку, мог выглядеть следующим образом:

```
this.button1 = new System.Windows.Forms.Button();

this.button1.Location = new System.Drawing.Point(120, 60);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(110, 40);
this.button1.Text = "Hello";

this.Controls.Add(this.button1);
```

С одной стороны, код выше является громоздким, а с другой, он не дает понимания того, как устроен интерфейс. Очень сложно проследить зависимости между контейнерами и понять структуру интерфейса. Кроме того, если Вы сталкивались с визуальным дизайнером WinForms-приложений, то знаете, что код, сгенерированный дизайнером, лучше не модифицировать. Во-первых, дизайнер может его просто не распознать, а во-вторых, все «ручные» изменения могут быть удалены при любой модификации интерфейса. Это связано с тем, что разобрать и обработать код, написанный на C#, достаточно сложно. Кроме того, код может содержать вставки, не связанные с построением интерфейса.

Рассмотрим аналогичный код на XAML:

```
<Canvas x:Name="LayoutRoot">
   <Button Content="Hello" Canvas.Top="60" Canvas.Left="120"
     Width="110" Height="40" x:Name="button1"></Button>
</Canvas>
```

Как видно, XAML является более интуитивным, чем код на C#, не только для разработчиков, но и для других членов команды, например, дизайнеров. Это позволяет использовать XAML для создания прототипов интерфейсов и переводить их на стадию разработки, используя один и тот же код. По атрибутам в XAML легко определить все параметры нашей кнопки и определить ее положение в общей иерархии интерфейсных объектов. Поскольку по коду XAML очень легко построить дерево объектов (различных анализаторов XML существует большое множество), а любые изменения по отношению к объекту четко привязаны к соответствующему тегу (атрибуты или дочерние элементы), то разработать дизайнер, который позволит не только создавать интерфейс с помощью мыши, но и редактировать код, достаточно просто. При этом XAML содержит только описание интерфейса и никаких вставок.

Компания Microsoft предлагает два продукта, которые позволяют редактировать XAML, — это Visual Studio (в 2008 версии нет полноценного визуального редактора, а вот в 2010 уже есть) и Expression Blend. Если Visual Studio лучше использовать для редактирования кода на C#, и небольших редакций XAML, то Expression Blend — полноценная утилита по созданию интерфейсов. Несомненно, для создания XAML можно использовать и обычный текстовый редактор, например, Notepad. Поскольку интерфейс на XAML представляет собой обычный текст, то его достаточно легко генерировать программным путем.

Представляя XAML как язык описания интерфейса, следует отметить, что не все элементы этого языка имеют визуальное представление. Существуют элементы, которые задают анимацию или трансформацию, работу с мультимедиа и др. С ними мы будем знакомиться в других разделах.

А сейчас перейдем к изучению синтаксиса XAML.

Основные конструкции

Если Вы владеете XML, то изучение XAML для Вас не составит особого труда. Как и в XML, любой элемент представлен тегом, который может иметь атрибуты и дочерние элементы. Атрибуты перечисляются в теге элемента со значениями в виде текстовых строк, а дочерние элементы задаются между открывающим и закрывающим тегами основного элемента:

```
<Canvas Name="LayoutRoot">
   <Button Content="Hello" Canvas.Top="60" Canvas.Left="120"
     Width="110" Height="40" Name="button1"></Button>
</Canvas>
```

Код выше описывает элемент **Canvas** с одним атрибутом и одним дочерним элементом (Button). В основном теге **Button** установлено шесть атрибутов и ни одного дочернего элемента. Как и в XML, тег **Button** можно было бы записать следующим образом (явное отсутствие закрывающего тега):

```
<Button Content="Hello" Canvas.Top="60" Canvas.Left="120"
  Width="110" Height="40" Name="button1" />
```

XAML является (как и XML) чувствительным к регистру символов. При этом регистр имеет значение не только для описания элементов и атрибутов, но и для описания значений атрибутов. Так, код ниже создает два объекта с именами Button1 и button1:

```
<Button Content="Hello" Canvas.Top="60" Canvas.Left="120"
  Width="110" Height="40" Name="button1" />
<Button Content="Hello" Canvas.Top="160" Canvas.Left="120"
  Width="110" Height="40" Name="Button1" />
```

Язык XAML является достаточно гибким. Как и в XML, язык XAML позволяет задавать свойства элементов, как через атрибуты, так и с помощью дочерних элементов. Так, следующие два блока кода выполняют одинаковую функцию:

Блок 1

```
<Button Content="Hello" Canvas.Top="60" Canvas.Left="120"
  Width="110" Height="40" Name="button1"></Button>
```

Блок 2

```
<Button Content="Hello" Canvas.Top="60" Canvas.Left="120"
Name="button1">
  <Button.Width>
    110
  </Button.Width>
  <Button.Height>
    40
  </Button.Height>
</Button>
```

При этом через атрибуты можно задавать не только простые значения, но и создавать более сложные объекты. Рассмотрим следующие два блока кода:

Блок 1

```
<Rectangle Width="100" Height="50" Fill="Red"></Rectangle>
```

Блок 2

```
<Rectangle Width="100" Height="50">
    <Rectangle.Fill>
        <SolidColorBrush Color="Red" />
    </Rectangle.Fill>
</Rectangle>
```

Первый блок кода имеет более простую запись, но, как и во втором блоке, для установки значения свойства **Fill**, создается кисть **SolidColorBrush**. Второй блок также не «идеален», ведь тут с помощью атрибута создается объект типа **Color**, который также можно расписать через набор дочерних элементов.

Создание объектов через атрибуты возможно с помощью специальных конвертеров. На самом деле, даже при простой установке значения свойства

Width, работает конвертор. Ведь **Width** является целочисленным свойством, а мы устанавливаем текст.

Если говорить о преимуществах создания объектов через атрибуты или вложенные элементы, то нужно отметить, что в отличие от WPF, Silverlight не проводит предварительную компиляцию XAML. Это означает, что на сторону клиента, XAML будет передаваться в исходном виде. Думаю, Вы согласитесь, что запись через атрибуты более компактная.

Рассмотрим два эквивалентных блока кода:

Блок 1

```
<TextBlock>
   Hello
</TextBlock>
```

Блок 2

```
<TextBlock Text="Hello"></TextBlock>
```

Как видно, в первом блоке свойство **Text** элемента **TextBlock** задается неявно. Такой подход реализован для простоты (вспомните, например, синтаксис HTML) использования XAML. Любой класс, описывающий элемент, может содержать лишь одно такое свойство (контентное свойство), которое задается с помощью атрибута **ContentPropertyAttribute** при описании класса.

Последний вопрос, который мы рассмотрим в этом разделе, это создание коллекций. Рассмотрим следующий блок кода:

```
<Rectangle Width="100" Height="50">
    <Rectangle.Fill>
            <LinearGradientBrush>
                <LinearGradientBrush.GradientStops>
                    <GradientStopCollection>
                        <GradientStop Offset="0.0" Color="Red" />
                        <GradientStop Offset="1.0" Color="Green" />
                    </GradientStopCollection>
                </LinearGradientBrush.GradientStops>
            </LinearGradientBrush>
        </Rectangle.Fill>
</Rectangle>
```

Этот код задает градиентную заливку для прямоугольника. Первое, что бросается в глаза, это невозможность задать градиентную заливку через атрибуты. Ведь тут требуется создать целую коллекцию объектов, каждый из которых имеет большое количество свойств. Между тем и этот синтаксис можно сократить.

Первое, на что нужно обратить внимание, это на то, что **GradientStops** свойство является контентным, а значит, его можно не писать. Тогда весь контент внутри **LinearGradientBrush** будет ассоциироваться именно с **GradientStops**:

```
<Rectangle Width="100" Height="50">
    <Rectangle.Fill>
       <LinearGradientBrush>
```

```
            <GradientStopCollection>
                <GradientStop Offset="0.0" Color="Red" />
                <GradientStop Offset="1.0" Color="Green" />
            </GradientStopCollection>
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

Кроме того, анализатор XAML достаточно «сообразительный», чтобы распознать тип свойства **GradientStops** и приготовится к созданию коллекции объектов. Поэтому тип коллекции явно задавать не нужно. Таким образом, наш код стал еще меньше:

```
<Rectangle Width="100" Height="50">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <GradientStop Offset="0.0" Color="Red" />
            <GradientStop Offset="1.0" Color="Green" />
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

Между тем, несмотря на одинаковый результат, первоначальный код и последний блок выполняются по-разному. Так, если Вы явно указываете коллекцию, то при запуске приложения, сначала генерируется объект типа коллекции, а затем создаются элементы, которые помещаются в созданный объект, а сам объект присваивается свойству (в данном случае **Fill**). Это позволяет задать не только коллекцию элементов, но и имя самой коллекции, и обращаться к ней в коде. Но используя такой подход, следует помнить, что многие коллекции не могут быть созданы явно. Их создание скрыто внутри дочернего элемента, и XAML анализатор просто вызывает метод **Add**, без явного создания объекта типа коллекции. Поэтому синтаксис с явной записью типа коллекции очень часто просто невозможен. Например, следующий код не будет работать, поскольку создать явно объект типа **UIElementCollection** невозможно:

```
<StackPanel x:Name="LayoutRoot" Background="White">
    <StackPanel.Children>
        <UIElementCollection>
            <Button Width="100" Height="50"></Button>
        </UIElementCollection>
    </StackPanel.Children>
</StackPanel>
```

А вот следующие два блока будут работать замечательно:

Блок 1

```
<StackPanel x:Name="LayoutRoot" Background="White">
    <StackPanel.Children>
        <Button Width="100" Height="50"></Button>
    </StackPanel.Children>
</StackPanel>
```

Блок 2 (Children — контентный элемент)

```
<StackPanel x:Name="LayoutRoot" Background="White">
    <Button Width="100" Height="50"></Button>
</StackPanel>
```

Пространства имен в XAML

Если Вы знакомы с языком программирования C#, то знаете, что все классы разбиты на отдельные логические группы. Разбиение достигается с помощью пространств имен. Так, класс **Button** находится в пространстве имен **System.Windows.Controls**. Соответственно полное имя класса **Button** можно записать как **System.Windows.Controls.Button**, но подобная запись встречается редко. Вместо этого используют директиву **using**.

В XML, как и в C#, также присутствует понятие пространств имен. Отличие состоит в том, что в XML обычно используется URI в качестве имени, кроме того может быть только одно пространство имен по умолчанию, а для остальных задается специальная приставка, требующаяся при записи элементов из этого пространства. Использование URI обосновано тем, что с помощью него значительно проще задать уникальное имя в пределах глобального Интернета.

Чтобы задать пространство имен по умолчанию, необходимо использовать специальный атрибут **xmlns**. Это ключевое слово, которое может быть указано в любом из элементов. При этом область действия указанного элемента — не только все дочерние элементы, но и сам элемент, где используется атрибут.

Для создания именованного пространства имен используют аналогичный синтаксис, но с указанием имени:

```
xmlns:x="http://baydachnyy.com/schemas"
```

Рассмотрим небольшой код работающего SilverLight-приложения:

```
<UserControl x:Class="SilverlightApplication6.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <UserControl.Resources>
        <Style x:Key="btnStyle" TargetType="Button">
            <Setter Property="Background" Value="Green"></Setter>
        </Style>
    </UserControl.Resources>

    <StackPanel Background="White">
            <Button Width="100" Height="50"
                    Style="{StaticResource
btnStyle}">Hello</Button>
    </StackPanel>
</UserControl>
```

В этом коде используются два пространства имен. Причем эти пространства имен можно встретить во всех Silverlight- и WPF-приложениях.

Первое пространство имен **http://schemas.microsoft.com/winfx/2006/xaml/presentation**, описывает все стандартные элементы, которые используются при построении SilverLight-интерфейсов. Нужно отметить, что при использовании кода на C# стандартные элементы входят в различные пространства имен, в отличие от XAML. В XAML же используется одно пространство для всех элементов. Это было сделано для упрощения синтаксиса. В противном случае пришлось бы создавать множество именованных пространств, что вызвало бы путаницу (в C# их ведь вообще можно не писать, а в XML (XAML) такой возможности нет).

Второе пространство имен **http://schemas.microsoft.com/winfx/2006/xaml** имеет служебное назначение. Тут содержится определение ряда атрибутов. Дизайнер дает этому пространству имен имя **x**, я рекомендую всегда использовать это имя, так как оно уже устоялось.

Естественно, что Вы не сможете обойтись только стандартными пространствами имен. Так, при подключении собственных (или приобретенных) сборок необходимо указать XAML анализатору, где искать Ваши классы. Для этой цели используется все тот же атрибут **xmlns**, но вместо URI он принимает имя пространства имен (в понятии .NET) и имя сборки. Вот как выглядит запись при подключении пространства имен **System.Windows.Controls** из сборки **System.Windows.Controls.Data.Input**:

```
xmlns:my="clr-namespace:System.Windows.Controls;
   assembly=System.Windows.Controls.Data.Input"
```

В данном случае мы подключили сборку, которая входит в SDK и содержит несколько элементовне входящих в стандартную поставку. Пример такого элемента — **Label**:

```
<my:Label>Hello</my:Label>
```

В заключение темы отметим, что если Вы работаете с Visual Studio, то обратитесь к директории «C:\Program Files (x86)\Microsoft Visual Studio 10.0\Xml\Schemas» (или аналогичной, в зависимости от места инсталляции). Тут находится большинство схем, описывающих XML пространства имен, и позволяющих работать системе IntelliSense.

Подключение кода и обработчиков событий

После того, как мы рассмотрели основные синтаксические конструкции XAML, давайте посмотрим, как XAML интегрируется с кодом на C#. Для этой цели создайте новый SilverLight-проект в Visual Studio, и обратите внимание на атрибуты корневого элемента **UserControl**:

```
<UserControl x:Class="SilverlightApplication7.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
```

Visual Studio автоматически создает два файла: **MainPage.xaml** и **MainPage.cs**. При этом в файле **MainPage.cs** создается класс, который и будет содержать весь код на C#, ассоциированный с нашим приложением (главным окном). Как видно из кода выше, ссылка на класс **MainPage** задается в XAML файле с помощью атрибута **Class**.

Прежде чем разбираться, как компилируется XAML и код на C#, добавим в созданное приложение кнопку с именем myButton:

```
<Button Name="myButton" Width="100" Height="50">Hello</Button>
```

А теперь попробуйте открыть код файла **MainPage.cs**. Класс MainPage будет выглядеть следующим образом:

```
public partial class MainPage : UserControl
{
    public MainPage()
    {
        InitializeComponent();
    }
}
```

Попробуйте немного поэкспериментировать с кодом и ввести имя созданной кнопки. Как ни странно, система IntelliSense работает. Между тем, определение объекта типа **Button** в нашем коде на C# отсутствует. Метод **InitializeComponent** также отсутствует. Создается такое впечатление, что на основании XAML файла генерируется обычный класс, который совмещается с **MainPage**, тем более что **MainPage** имеет атрибут **partial** (частичный тип). Подобная модель работает в ASP.NET и довольно успешно.

Однако в Silverlight механизм немного другой. Чтобы разобраться с работой этого механизма, просто откомпилируйте приложение. Во время компиляции будет создана директория **obj**, где, среди прочих файлов, находится **MainPage.g.cs**. Содержимое этого файла выглядит следующим образом:

```
    public partial class MainPage :
System.Windows.Controls.UserControl {

        internal System.Windows.Controls.Grid LayoutRoot;

        internal System.Windows.Controls.Button myButton;

        private bool _contentLoaded;

        /// <summary>
        /// InitializeComponent
        /// </summary>
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        public void InitializeComponent() {
            if (_contentLoaded) {
                return;
            }
            _contentLoaded = true;

            System.Windows.Application.LoadComponent(this,
```

```
            new System.Uri(
            "/SilverlightApplication7;component/MainPage.xaml",
            System.UriKind.Relative));
        this.LayoutRoot =
            ((System.Windows.Controls.Grid)(this.FindName(
            "LayoutRoot")));
        this.myButton =
            ((System.Windows.Controls.Button)(
            this.FindName("myButton")));
    }
}
```

Таким образом, XAML файл поступает на сторону клиента в своем исходном виде (в составе сборки, как ресурс) и просто загружается с помощью метода **LoadComponent**. Хочу обратить особое внимание на то, что XAML даже не сжимается, как в WPF (там он поставлялся в виде BAML).

После создания дерева объектов и привязки его к текущему окну (**LoadComponent** использует «this» для загрузки дерева и ассоциации его с текущим экземпляром окна), происходит инициализация переменных, имена и типы которых совпадают с теми, которые мы указали в XAML файле. Что интересно, в WPF метод **LoadComponent** берет на себя и инициализацию ссылок на объекты в XAML (BAML). В Silverlight это делается отдельно:

```
this.myButton =
    ((System.Windows.Controls.Button)(
    this.FindName("myButton")));
```

Таким образом, **MainPage.g.cs** компилируется совместно с **MainPage.cs**, а XAML файл используется Visual Studio для генерации недостающего кода.

Если вернуться к корневому элементу в XAML файле, то можно сделать вывод, что атрибут **Class** — это всего лишь подсказка для Visual Studio, чтобы сгенерировать класс с нужным именем.

Между тем, нужно отдать должное системе IntelliSense, ведь она генерирует промежуточный класс «на лету», скрывая ненужные детали от разработчика.

Если Вы решились создавать приложения без Visual Studio, то придется изрядно повозиться с утилитой **msbuild**, которая позволяет генерировать промежуточные классы (ее Visual Studio и использует).

В конце раздела остановимся также на механизме подключения обработчиков событий. Тут все просто: достаточно записать имя события в качестве атрибута, а в качестве значения указать имя метода-обработчика события. При генерации кода, загружающего XAML, компилятор перенесет привязку обработчика события к объекту, в сгенерированный метод **InitializeComponent**.

```
<Button Name="myButton" Width="100" Height="50"
    Click="Button_Click">
      Hello
</Button>
```

Дополнительные преимущества дает Visual Studio, тут при выборе события, появляется возможность создать новый обработчик события или выбрать один из существующих (рис. 4.1).

Рис. 4.1. IntelliSense в Visual Studio

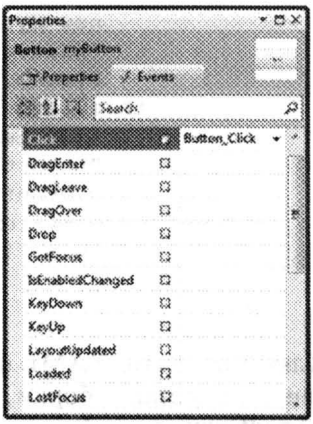


Рис. 4.2. Окно свойств в Visual Studio

Если Вы привыкли пользоваться визуальным дизайнером, то тут доступно специальное окно событий. Двойной щелчок генерирует нужное событие и перебрасывает разработчика в редактор кода.

Расширение разметки

Существует ряд сценариев, которые не позволяют указать в качестве свойства одного из атрибутов непосредственное значение. Например, если значение одного свойства в одном элементе зависит от текущего значения другого свойства во втором элементе, то для записи значения атрибутов используют расширения разметки. Расширения разметки позволяют установить значение атрибута нестандартным образом, чаще всего такая необходимость возникает при установке значения свойств динамически.

В Silverlight существует четыре расширения разметки:

- **Binding** — позволяет сделать привязку свойства к любому динамическому объекту, например, объекту заданного класса. Расширение очень эффективно при привязке данных к элементу;
- **StaticResources** — позволяет указать ссылку на ресурсы;
- **TemplateBinding** — используется аналогично **Binding**, но при работе с шаблонами элементов;
- **RelativeSource** — используется для связывания атрибутов с данными самого же объекта.

Рассмотрим два примера, использующих расширения разметки.

Пример 1

```
<UserControl
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <UserControl.Resources>
        <Style x:Name="btnStyle" TargetType="Button">
```

```
            <Setter Property="Background" Value="Green"></Setter>
        </Style>
    </UserControl.Resources>

    <Grid x:Name="LayoutRoot" Background="White">
        <Button Name="myButton" Width="100" Height="50"
                Style="{StaticResource btnStyle}">
            Hello
        </Button>
    </Grid>
</UserControl>
```

В первом примере мы использовали расширение разметки для установки описанного в ресурсах стиля.

Пример 2

```
<UserControl
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <StackPanel x:Name="LayoutRoot" Background="White">
        <Slider Name="sld1" Width="300" Height="50"
           Minimum="100" Maximum="200"
           Value="10">
        </Slider>
        <TextBox Text="Hello" Width=
          "{Binding Value, ElementName=sld1, Mode=TwoWay}">
         </TextBox>
    </StackPanel>
</UserControl>
```

Данный пример демонстрирует привязку свойства **Width** элемента **TextBox** к значению ползунка.

Большее количество примеров будет встречаться по ходу книги. Так, расширение **Binding** будет интенсивно использоваться в главе о связывании элементов и данных.

Зависимые свойства

Один из примеров этой главы содержал следующий код:

```
<Button Content="Hello" Canvas.Top="60" Canvas.Left="120"
  Width="110" Height="40" Name="button1" />
```

Если посмотреть на этот код внимательно, то можно выделить два атрибута, которые отличаются от общепринятых: **Canvas.Top**, **Canvas.Left**. Не сложно догадаться, что эти атрибуты устанавливают положение кнопки относительно родительского контейнера. В свою очередь, в Silverlight существует множество контейнеров, каждый из которых может снабжать различным набором свойств дочерние объекты (отступы от границ и между элементами, строка и столбец для размещения элемента и др.). Очевидно, что все эти

свойства определить в классе **Button** невозможно. Хранить значения свойств в классе контейнере для каждого элемента также неправильно, ведь контейнер не может следить за жизненным циклом элемента и управлять списком значений. Поэтому в Silverlight и WPF в иерархию классов был введен специальный класс **DependencyObject**. Этот класс является прямым наследником от **Object** и, следовательно, неявно наследуется всеми объектами, имеющими визуальное представление. Задача класса **DependencyObject** состоит в обеспечении поддержки любого количества зависимых свойств. Имея два метода **SetValue** и **GetValue**, этот класс позволяет любому объекту хранить динамический список свойств, ассоциирующихся с родительским контейнером. Таким образом, используя запись **Canvas.Top="60"**, мы фактически вызываем следующий метод:

```
button1.SetValue(Canvas.TopProperty, 60)
```

В свою очередь, элемент **Canvas,** размещая объекты, запрашивает их зависимые свойства с помощью метода **GetValue** и выбирает необходимые значения.

Динамическая загрузка XAML

Последняя тема, которую мы рассмотрим в этой главе, — это динамическая загрузка XAML фрагментов и добавление их в дерево Silverlight-элементов. Подобный метод мы рассматривали в предыдущей главе — **CreateFromXaml**. Этот метод можно использовать только из JavaScript, что усложняет задачу разработчика. Но для любителей управляемого кода, в Silverlight, существует специальный управляемый класс **XmlReader**, реализующий статический метод **Load**.

Метод **Load** работает практически аналогично методу **CreateFromXaml**. Разница состоит в том, что метод Load возвращает объект типа **object**, который должен быть преобразован к нужному типу. Как и при работе с **CreateFromXaml**, метод **Load** принимает правильно сформированный XAML, содержащий только один родительский элемент.

Рассмотрим небольшой пример:

```
public MainPage()
{
    InitializeComponent();

    Button b=
        (Button)System.Windows.Markup.XamlReader.Load(
        "<Button "+

"xmlns=\"http://schemas.microsoft.com/winfx/2006/xaml/presentation\
" "+
        "Width=\"100\" Height=\"50\" Content=\"Hello\"></Button>");

    b.Click += Button_Click;

    LayoutRoot.Children.Add(b);
```

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hello");
}
```

Тут с помощью метода **Load** мы формируем элемент **Button**, к которому выполняем привязку обработчика события **Click**, и добавляем сформированный объект в родительский контейнер. Обратите внимание на то, что при загрузке фрагмента необходимо указать все XML пространства имен, включая и пространство имен по умолчанию.

Еще один нюанс, который удалось обнаружить при динамическом создании интерфейсов на основе XAML, — это плохая работа с контентными свойствами. Поэтому рекомендую задавать все контентные свойства в виде атрибутов, то есть не используя «упрощения», к которым Вы привыкли в Visual Studio.

Заключение

Итак, мы рассмотрели основные синтаксические конструкции XAML. Если Вы являетесь разработчиком (не дизайнером), то Вам придется сталкиваться с XAML довольно часто. Создавая интерфейсы для Forms-приложений, разработчик достаточно редко заглядывал в сгенерированный код, однако интерфейсы Silverlight-приложений могут быть на порядок сложнее. К сожалению, визуальный редактор Silverlight-интерфейсов появился в Visual Studio только с 2010 версии, поэтому многие разработчики используют его только для быстрого просмотра, разрабатываемого на XAML интерфейса (но не для построения интерфейса), по привычке отдавая предпочтение прямому редактированию XAML . В любом случае, как Вы смогли убедиться, в XAML нет ничего сложного. Фактически теперь нужно сосредоточиться на изучении элементов управления, которые доступны в Silverlight, а также на их представлении в XAML. Этому и посвящена следующая глава.

Глава 5

ЭЛЕМЕНТЫ УПРАВЛЕНИЯ И СОБЫТИЯ

Немного об элементах управления

Разобравшись с архитектурой Silverlight-приложений, можно перейти к созданию разнообразных интерфейсов. Обычно когда рассматривают Silverlight, первым делом начинают работу с анимацией и графикой. Конечно, эффекты в Silverlight-приложениях могут быть очень полезны, но если Вы выбрали эту технологию, то наверняка хотите разработать приложение, содержащее элементы управления, навигацию, и реализовать приложение, подобное Windows-клиенту. Поэтому продолжим наш путь, рассматривая именно элементы управления.

Разрабатывая приложения, Вы можете встретиться с четырьмя категориями элементов управления:

- элементы, поставляемые вместе с Silverlight — такие элементы гарантировано присутствуют на машине у пользователя, а их использование не увеличивает размер приложения;
- элементы, входящие в SDK (пакет для разработчика) — тут присутствует множество интересных элементов, которые «не поместились» в инсталляцию Silverlight. Очень часто без этих элементов можно обойтись, и чаще всего они используются в корпоративных приложениях. Для использования элементов из SDK Вам придется подключать дополнительные библиотеки, что увеличит размер Вашего приложения. Последнее не очень существенно в корпоративных приложениях;
- элементы, разрабатываемые сообществом — Microsoft поддерживает специальный проект для развития Open Source решений — codeplex.com Данный ресурс содержит множество полезных компонентов для Silverlight, включая специальный Toolkit, в разработке которого принимают участие и сотрудники Microsoft;
- элементы сторонних производителей — множество компаний разрабатывают свои библиотеки элементов управления с целью заработать деньги. Вы сможете приобрести один из доступных пакетов на рынке.

В этой книге мы рассмотрим лишь некоторые элементы из первых трех групп элементов управления, и совсем не будем останавливаться на последней группе. Хотя рассмотрим еще, как создавать собственные элементы управления (возможно, создание элементов управления принесет доход и Вам).

Разделив элементы на категории по способу распространения, можно увидеть иерархию классов, которые лежат в основе абсолютно всех элементов управления:

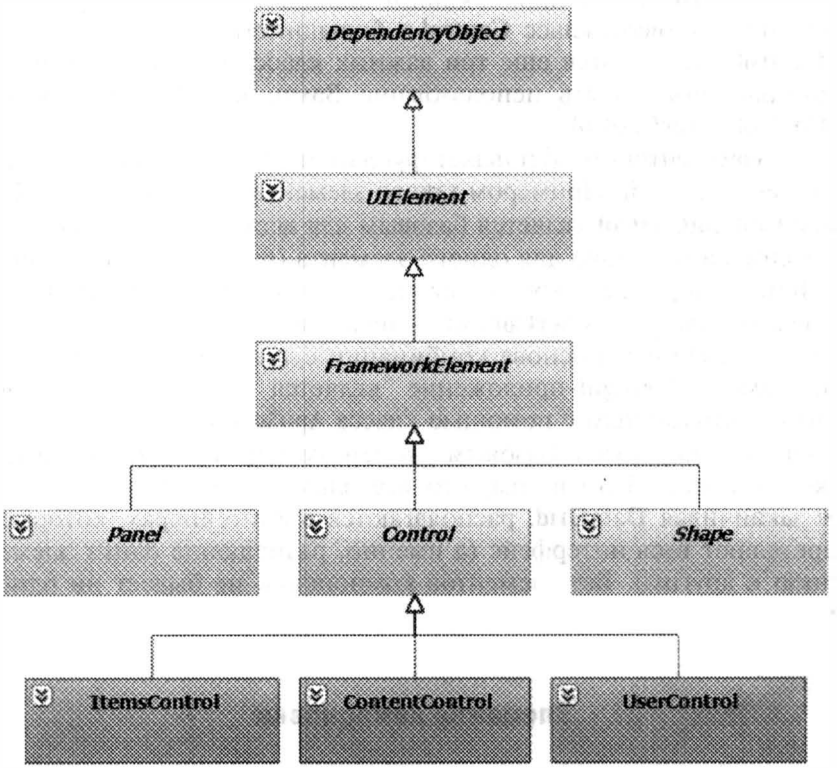


Рис. 5.1. Иерархия базовых классов

Как видно из рисунка, все визуальные элементы наследуются от класса **DependencyObject**, который является прямым наследником **object**. Значение этого класса мы обсуждали, когда рассматривали основы XAML. Именно этот класс позволяет задавать зависимые свойства.

Следующим в иерархии классов является **UIElement**, который задает основные события, связанные с вводом, включая получение и потерю фокуса. Также этот класс задает фундамент для компоновки элементов, хотя более конкретная реализация механизмов компоновки находится в следующем классе по иерархии — **FrameworkElement**.

В свою очередь, от **FrameworkElement** наследуются еще несколько абстрактных классов: **Panel**, **Control**, **Shape**. Так, класс **Panel** является базовым для элементов компоновки, определяющих размещение элементов управления в интерфейсе приложения. Класс **Shape** является базовым для различных графических примитивов, таких как линия или прямоугольник. А класс **Control** является базовым классом для всех элементов управления с визуальным представлением. Причем благодаря функциональности класса **Control**, визуальное представление любого из элементов-наследников можно изменить с помощью специальных шаблонов.

Несколько элементов управления, которые не укладываются ни под одну из описанных выше категорий, наследуются напрямую от **FrameworkElement**. Примером может служить **MediaElement**, служащий для отображения видео.

Нужно отметить, что при создании собственных элементов управления Вы будете использовать класс **Control** в большинстве случаев. Между тем, от класса **Control** наследуются еще три важных класса (на этот раз не абстрактных), которые могут быть использованы Вами, как базовые: **ItemsControl**, **ContentControl**, **UserControl**.

Класс **ItemsControl** предоставляет фундамент для всех элементов, работающих с группой записей. Примером такого элемента может служить **ListBox**.

Класс **ContentControl** является базовым для всех элементов, которые являются контейнерами только для одного элемента (пусть даже контейнера). Так, элемент **Button**, порожден (хоть и неявно) от класса **ContentControl**.

Последний класс — **UserControl** — позволяет создавать пользовательские элементы управления на основе комбинации существующих элементов. Фактически, само Silverlight-приложение является своеобразным элементом управления, загружаемым с помощью класса **Application**.

Прежде чем перейти к базовым элементам управления, рассмотрим элементы компоновки. Дело в том, что все элементы управления, начиная от **Button** и заканчивая **DataGrid**, располагаются в контейнерах, которые полностью определяют весь интерфейс (а именно, размещение одних элементов по отношению к другим). Без элементов компоновки не бывает ни одного приложения.

Элементы компоновки

Как Вы смогли заметить, все окна в Silverlight создаются на основе элемента управления **UserControl**. С одной стороны, он может быть загружен и отображен с помощью объекта **Application**, а с другой — способен содержать лишь один элемент. Если посмотреть глубже, то можно обнаружить, что **UserControl** отличается от класса **Control** только наличием свойства **Content**, которое способно содержать ссылку на объект типа **UIElement**. То есть **UserControl** — это быстрый способ загрузить и отобразить любой из визуальных элементов, определив пространства имен и события, связанные с этим элементом. Естественно, что ни один разработчик не захочет ограничить свой интерфейс единственным элементом управления. Но ведь никто не запрещает в качестве этого единственного элемента использовать контейнер. Именно поэтому первым элементом, который описывают в XAML внутри **UserControl**, является один из контейнеров.

В отличие от элементов управления, обладающих свойством **Content** и способных содержать лишь один элемент, все контейнеры определяют свойство **Children**, которое содержит ссылку на коллекцию элементов типа **UIElement**. Иными словами, контейнеры поддерживают механизм управления любым количеством визуальных элементов. При этом основными контейнерами являются те, которые не только способны хранить набор элементов в качестве содержимого, но и определяют их размещение на панели — компонуют.

В Silverlight 4 присутствует три контейнера — элемента компоновки: **Canvas**, **StackPanel** и **Grid**. Наряду с контейнерами компоновки существует еще несколько полезных классов, которые описывают специализированные панели (**TabPanel**), а также элементы компоновки, которые не входят в стандартную поставку Silverlight, но мы начнем именно со стандартных элементов компоновки.

Элемент управления Canvas

Самый «плохой» элемент компоновки в Silverlight — это **Canvas**. Он позволяет размещать элементы управления с привязкой к абсолютным, явно заданным позициям относительно контейнера. И именно в абсолютных позициях состоит его недостаток. Несомненно, разрабатывая Windows Forms приложения, разработчики привыкли именно к абсолютным позициям элементов управления, но в Silverlight подобное размещение элементов управления считается плохим тоном. Ведь речь идет не только о векторной графике, но и о приложении, работающем внутри окна браузера. Это означает, что разработчик совершенно не может полагаться на размер окна. Да и использовать преимущества векторного интерфейса достаточно сложно, если все элементы имеют абсолютную привязку. Тем не менее, существуют ситуации, когда использование элемента Canvas наиболее оптимальное решение.

Рассмотрим небольшой пример использования элемента **Canvas**:

```
<UserControl x:Class="SilverlightApplicationTest.Page"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Canvas x:Name="LayoutRoot" Background="Yellow">
        <Button Canvas.Top="10" Canvas.Left="10"
           Content="Button 1">
        </Button>
        <Button Canvas.Top="100" Canvas.Left="100"
           Content="Button 2">
        </Button>
    </Canvas>
</UserControl>
```

В этом примере на странице отображается две кнопки с абсолютными позициями (10,10) и (100,100).

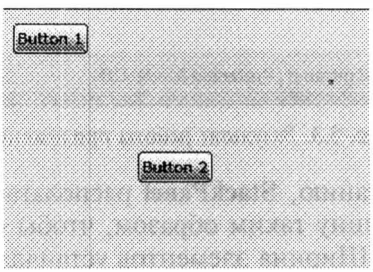


Рис. 5.2. Результат работы приложения

При попытке растянуть окно браузера, элемент управления **Canvas** будет заполнять все окно, но кнопки при этом останутся на месте.

Обычно если используется элемент управления **Canvas**, элементам управления устанавливаются еще два «плохих» свойства — **Width** и **Height**, которые позволяют задать длину и ширину элемента управления.

Элемент управления StackPanel

Один из простейших элементов компоновки — **StackPanel**. Этот элемент располагает все дочерние элементы в одну строку или колонку, заполняя все доступное пространство. Рассмотрим пример, отображающий несколько кнопок внутри **StackPanel**:

```
<UserControl x:Class="Chapter5_Test.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <StackPanel x:Name="LayoutRoot" Background="White">
        <Button Content="Button 1"></Button>
        <Button Content="Button 2"></Button>
        <Button Content="Button 3"></Button>
        <Button Content="Button 4"></Button>
    </StackPanel>
</UserControl>
```

Результат работы этого приложения показан ниже:

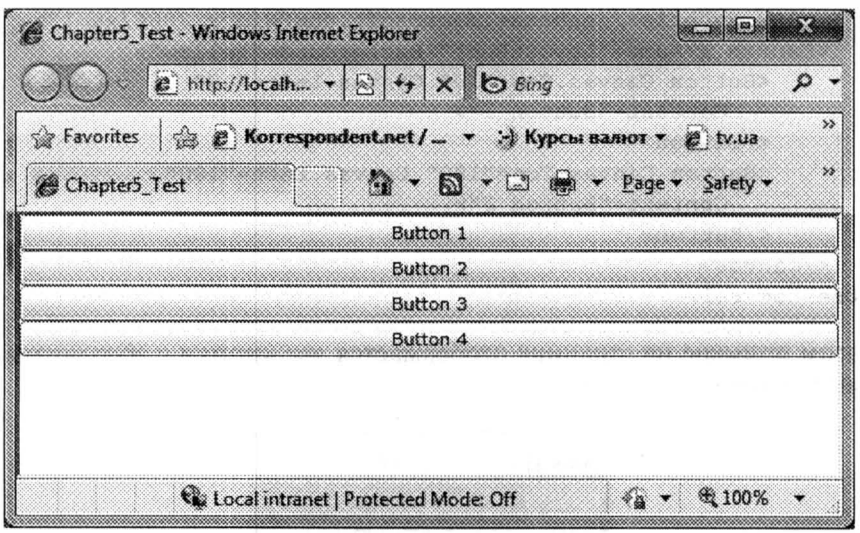


Рис. 5.3. Результат работы приложения

Как видно, по умолчанию, **StackPanel** располагает элементы по вертикали, и устанавливает их длину таким образом, чтобы они могли заполнить все пространство **StackPanel**. Ширина элементов устанавливается исходя из их содержимого — так, чтобы элемент мог отобразить все свое содержимое.

Чтобы изменить расположение элементов с вертикального на горизонтальное, элемент **StackPanel** обладает свойством **Orientation**:

```
<UserControl x:Class="Chapter5_Test.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <StackPanel x:Name="LayoutRoot" Background="White"
        Orientation="Horizontal">
        <Button Content="Button 1"></Button>
        <Button Content="Button 2"></Button>
        <Button Content="Button 3"></Button>
        <Button Content="Button 4"></Button>
    </StackPanel>
</UserControl>
```

Результат работы этого приложения показан ниже:

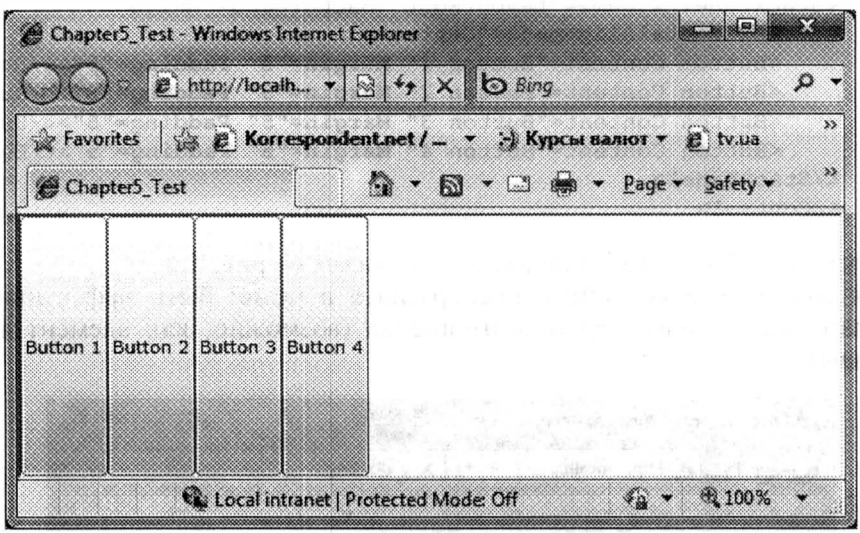


Рис. 5.4. Результат работы приложения

Теперь элемент **StackPanel** растянул кнопки так, чтобы они занимали все пространство по вертикали.

Естественно, что описанное выше поведение элемента компоновки **StackPanel** может показаться не очень удобным. Поэтому чтобы получить все его преимущества, необходимо комбинировать его базовые возможности с установкой набора свойств **StackPanel** и внутренних элементов, которые наследуются от **FrameworkElement**. Вот некоторые из полезных свойств:

- **HorizontalAlignment** — определяет позиционирование дочерних элементов контейнера, когда доступно дополнительное пространство по горизонтали. Может принимать одно из следующих значений: **Left**, **Right**, **Center** или **Stretch** (по умолчанию);
- **VerticalAlignment** — определяет позиционирование дочерних элементов контейнера, когда доступно дополнительное пространство по вертикали.

Может принимать одно из четырех значений: **Top**, **Bottom**, **Center** или **Stretch** (по умолчанию);

- **Margin** — позволяет задать дополнительное место вокруг элемента управления. Этот параметр способен принимать значения, задающие расстояние от каждой из границ элемента;
- **MinWidth** — устанавливает минимальные размеры элемента (длину);
- **MinHeight** — устанавливает минимальные размеры элемента (ширину);
- **MaxWidth** — устанавливает максимальные размеры элемента (длину);
- **MaxHeight** — устанавливает максимальные размеры элемента (ширину);
- **Padding** — задает отступы внутри элемента управления, отделяя его содержимое от границ.

Модифицируем наш пример, добавив несколько из описанных свойств:

```
<UserControl x:Class="Chapter5_Test.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <StackPanel x:Name="LayoutRoot" Background="White"
        HorizontalAlignment="Center">
        <Button Content="Button 1" Margin="5" Padding="5"></Button>
        <Button Content="Button 2" Margin="5" Padding="5"></Button>
        <Button Content="Button 3" Margin="5" Padding="5"></Button>
        <Button Content="Button 4" Margin="5" Padding="5"></Button>
    </StackPanel>
</UserControl>
```

Результат работы этого приложения показан на рис. 5.5.

Как видно, наш контейнер преобразился и может быть эффективно использован как элемент других контейнеров (возможно, как элемент другой **StackPanel**).

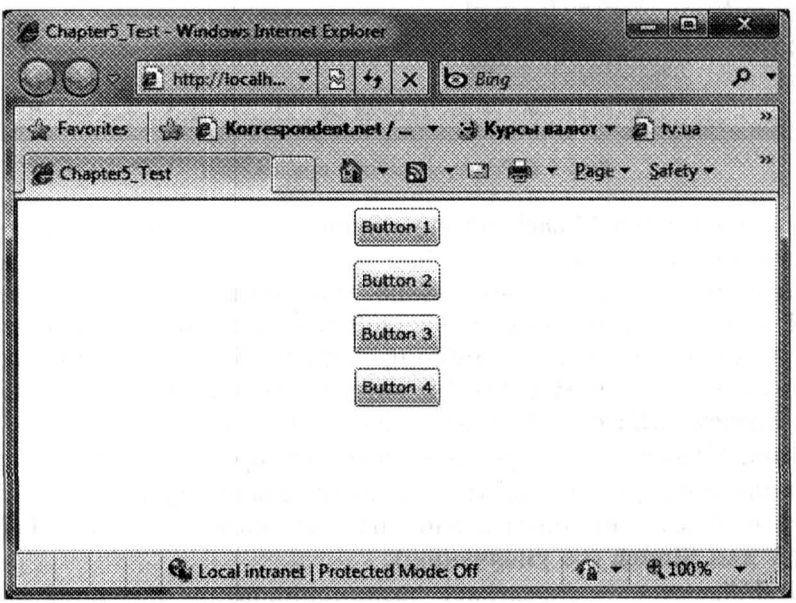


Рис. 5.5. Результат работы приложения

Элемент управления Grid

Элемент управления **Grid** является одним из наиболее мощных элементов компоновки. Фактически он позволяет разбить все доступное пространство на набор ячеек, задавая определенное количество столбцов и строк. При этом чтобы отобразить элемент в той или иной ячейке, разработчик должен явно указать ее номер. Ниже показан небольшой пример использования элемента компоновки **Grid**. Тут мы специально задали свойство **ShowGridLines**, чтобы иметь возможность видеть размер ячеек:

```
<UserControl x:Class="Chapter5_Test.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid x:Name="LayoutRoot" Background="White"
ShowGridLines="True">
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Button Content="Button 1" Grid.Row="0"
Grid.Column="0"></Button>
        <Button Content="Button 2" Grid.Row="0"
Grid.Column="1"></Button>
        <Button Content="Button 3" Grid.Row="1"
Grid.Column="2"></Button>
        <Button Content="Button 4" Grid.Row="1"
Grid.Column="3"></Button>
    </Grid>
</UserControl>
```

На рис. 5.6 показан результат использования элемента компоновки **Grid**:

Естественно, использование элемента **Grid** в виде, показанном выше, не очень эффективно. Поэтому чтобы привести интерфейс к более привычному виду, разработчик может воспользоваться свойствами, описанными в разделе, посвященном **StackPanel**. Кроме этого, разработчик может задать размеры колонок и строк одним из трех способов:

- **Абсолютные размеры** — этот способ наименее интересен, так как позволяет задавать длину и ширину колонок явно, что не приветствуется в Silverlight;
- **Автоматические размеры** — для установки автоматических размеров используется специальное значение Auto, которое выделяет ровно столько места, сколько необходимо для отображения содержимого. Ниже показан пример с использованием данного способа.

```
<UserControl x:Class="Chapter5_Test.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

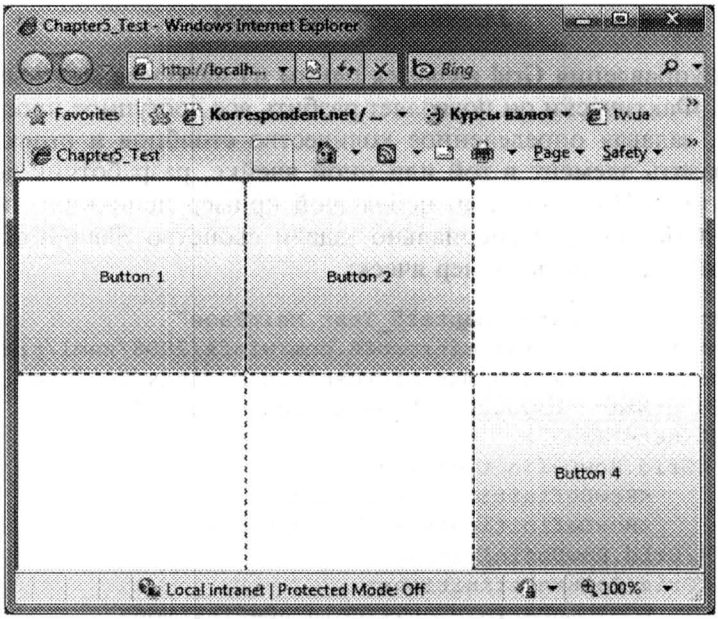


Рис. 5.6. Использование элемента управления Grid

```
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid x:Name="LayoutRoot" Background="White"
ShowGridLines="True">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Button Content="Button 1" Grid.Row="0"
Grid.Column="0"></Button>
        <Button Content="Button 2" Grid.Row="0"
Grid.Column="1"></Button>
        <Button Content="Button 3" Grid.Row="1"
Grid.Column="2"></Button>
        <Button Content="Button 4" Grid.Row="1"
Grid.Column="3"></Button>
    </Grid>
</UserControl>
```

Как видно из рисунка, все кнопки приняли наиболее оптимальный для себя размер (рис. 5.7):

- **Пропорциональные размеры** — этот способ установки размеров позволяет установить пропорции, в соответствии с которыми будут определяться размеры. Так пример ниже демонстрирует разбиение места для строк в

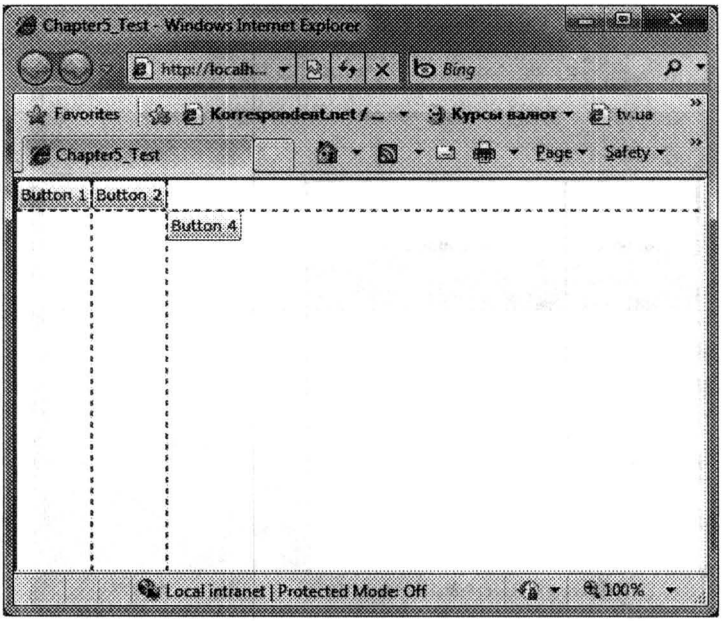


Рис. 5.7. Установка размера колонок и строк с помощью Auto

соотношении 1 к 2, а место для столбцов в соотношении 1 к 2 для второго и третьего столбцов по отношению к первому:

```
<UserControl x:Class="Chapter5_Test.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid x:Name="LayoutRoot" Background="White"
ShowGridLines="True">
        <Grid.RowDefinitions>
            <RowDefinition Height="*"></RowDefinition>
            <RowDefinition Height="2*"></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"></ColumnDefinition>
            <ColumnDefinition Width="2*"></ColumnDefinition>
            <ColumnDefinition Width="2*"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Button Content="Button 1" Grid.Row="0"
Grid.Column="0"></Button>
        <Button Content="Button 2" Grid.Row="0"
Grid.Column="1"></Button>
        <Button Content="Button 3" Grid.Row="1"
Grid.Column="2"></Button>
        <Button Content="Button 4" Grid.Row="1"
Grid.Column="3"></Button>
    </Grid>
</UserControl>
```

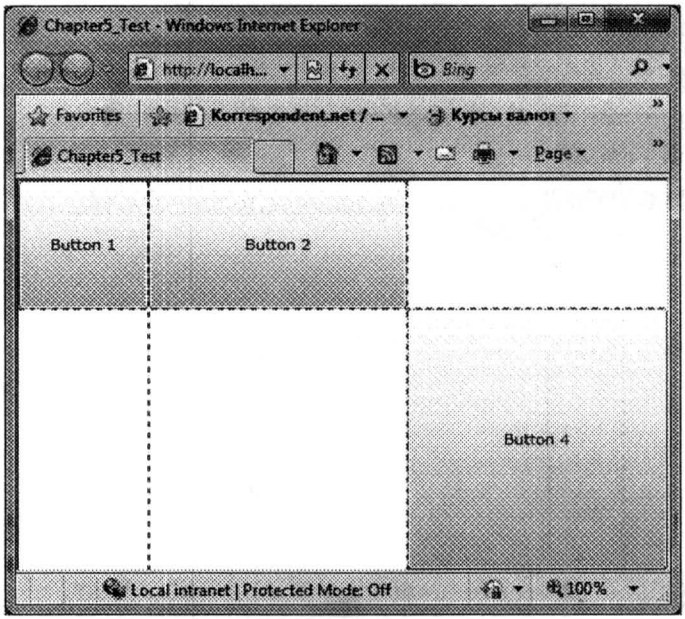


Рис. 5.8. Установка пропорциональных размеров колонок и строк

Следующая интересная функциональность **Grid** — это возможность сли ия колонок или строк. Рассмотрим следующий пример:

```
<UserControl x:Class="Chapter5_Test.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentatic
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid x:Name="LayoutRoot" Background="White"
ShowGridLines="True">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <TextBlock Text="Employee" Grid.Row="0" Grid.Column="0"
            Grid.ColumnSpan="2" HorizontalAlignment="Center"
            FontSize="16" FontWeight="Bold" Margin="5">
        </TextBlock>

        <TextBlock Text="First Name:" Grid.Row="1"
            Grid.Column="0" Margin="5"></TextBlock>
        <TextBox Grid.Column="1" Grid.Row="1" MinWidth="100"
            Margin="5"></TextBox>

        <TextBlock Text="Last Name:" Grid.Row="2" Grid.Column="0'
```

```
            Margin="5"></TextBlock>
        <TextBox Grid.Column="1" Grid.Row="2" MinWidth="100"
            Margin="5"></TextBox>

        <TextBlock Text="EMail:" Grid.Row="3" Grid.Column="0"
            Margin="5"></TextBlock>
        <TextBox Grid.Column="1" Grid.Row="3" MinWidth="100"
            Margin="5"></TextBox>

        <Button Grid.Column="0" Grid.Row="4" Grid.ColumnSpan="2"
            Content="Send" Margin="5"
            HorizontalAlignment="Center" Padding="5">
        </Button>
    </Grid>
</UserControl>
```

Как видно из рисунка, мы смогли получить достаточно симпатичный интерфейс даже без установки явных размеров и позиций элементов:

Рис. 5.9. Результат работы приложения

Наконец, перед завершением раздела, рассмотрим еще одну возможность элемента компоновки **Grid** — разделение окна с помощью специального элемента **GridSplitter**. Сразу начнем рассматривать этот элемент с небольшого примера:

```
<UserControl x:Class="Chapter5_Test.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:controls=
 "clr-namespace:System.Windows.Controls;assembly=System.Windows.Con
trols">
    <Grid x:Name="LayoutRoot" Background="White"
ShowGridLines="True">
```

```
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Button Content="Button" Grid.Column="0"
Grid.Row="0"></Button>
        <Button Content="Button" Grid.Column="0"
Grid.Row="1"></Button>
        <Button Content="Button" Grid.Column="2"
Grid.Row="0"></Button>
        <Button Content="Button" Grid.Column="2"
Grid.Row="1"></Button>

        <controls:GridSplitter VerticalAlignment="Stretch"
            HorizontalAlignment="Center" Width="3" Grid.Row="0"
            Grid.Column="1" Grid.RowSpan="2">
        </controls:GridSplitter>
    </Grid>
</UserControl>
```

Результат работы этого примера показан ниже (рис. 5.10).

Обратите внимание на то, что для создания разделения с помощью **GridSplitter** лучше всего выделить отдельную строку (или столбец), куда и разместить сам элемент. Чтобы сделать разделение всего интерфейса, воспользуйтесь слиянием колонок или строк с помощью **RowSpan** или **ColSpan**. Выделенная строка или колонка должна иметь автоматический размер, а сам

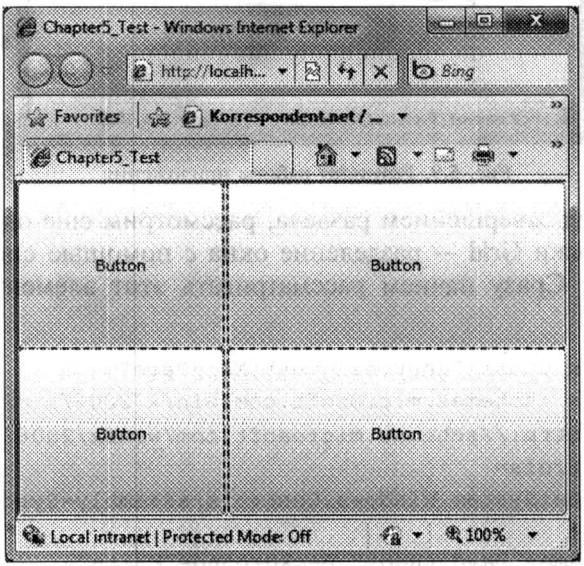


Рис. 5.10. Использование GridSplitter

GridSplitter должен задавать явную ширину, это позволит сделать элемент видимым и удобным для пользователя. Наконец, чтобы определить ориентацию разделителя, используют такие свойства как **HorizontalAlignment** и **VerticalAlignment**. Также нужно заметить, что **GridSplitter** не входит в набор стандартных элементов управления Silverlight, а поставляется с SDK. Поэтому для его использования Вам потребуется подключить дополнительную сборку (**System.Windows.Controls.dll**) и объявить соответствующее пространство имен.

Базовые элементы управления

Рассмотрев все элементы компоновки, перейдем к стандартному набору элементов управления, которые являются неотъемлемой частью любого интерфейса.

Класс Control

Прежде чем переходить к таким элементам как кнопки и текстовые поля, рассмотрим класс **System.Windows.Control**, который является базовым для всех элементов управления, наделяя их базовыми свойствами. Начнем с трех свойств, которые определяют цветовые характеристики элементов управления:

- **Background** — тут содержатся свойства фона, который, как правило, является поверхностью элемента управления;
- **Foreground** — это свойство задает цвет текста, который может присутствовать в качестве содержимого элементов управления;
- **Opacity** — это свойство определяет прозрачность элемента управления. Прозрачность задается в процентах и варьируется от 0 до 1.

Свойства **Background** и **Foreground** могут принимать объект типа **Brush**, описывающий кисть. В главе, посвященной работе с графикой, мы рассмотрим доступные кисти в Silverlight, а сейчас рассмотрим пример работы с самой простой кистью — **SolidBrush**:

```
<UserControl x:Class="Chapter5_Test.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <StackPanel>
        <Button Background="#FFFF0000" Foreground="#FF00FF00"
            Content="Hello"></Button>

        <Button Background="Red" Foreground="LightGreen"
            Content="Hello"></Button>

        <Button Content="Hello">
            <Button.Background>
                <SolidColorBrush Color="Red"></SolidColorBrush>
            </Button.Background>
            <Button.Foreground>
                <SolidColorBrush
Color="LightGreen"></SolidColorBrush>
```

```
            </Button.Foreground>
        </Button>
    </StackPanel>
</UserControl>
```

В этом примере цвет задается сразу тремя способами: в шестнадцатеричном представлении, используя возможности конвертера значений свойств в XAML, явным созданием объекта **SolidColorBrush**. Аналогично свойства можно устанавливать и из C# кода:

```
btn.Background = new SolidColorBrush(Colors.Red);
```

Тут Colors представляет собой специальный класс, определяющий несколько популярных цветов.

Если используемый элемент работает с текстом, то для установки шрифта текста могут пригодиться следующие свойства:

- **FontFamily** — задает имя шрифта, который Вы хотите использовать;
- **FontSize** — размер шрифта в единицах Silverlight;
- **FontStyle** — позволяет задать стиль шрифта, такой как **Normal** или **Italic**;
- **FontWeight** — задает вес текста, например **Bold**;
- **FontStretch** — используется для поддержки Open Type шрифтов, в частности позволяет сжимать или растягивать текст.

Ниже приведен пример использования перечисленных свойств, используемых для установки надписи кнопки в полужирный Arial шрифт, отображаемый курсивом:

```
<Button Background="Red" Foreground="LightGreen"  Content="Hello"
   FontFamily="Arial"
   FontSize="16"
   FontStyle="Italic"
   FontWeight="Bold">
</Button>
```

Следующие несколько свойств позволяют задать размеры элемента, а также отступы от соседних элементов и от содержимого:

- **Width** — длина элемента управления;
- **Height** — ширина элемента управления;
- **Padding** — расстояние от содержимого до границ элемента;
- **Margin** — расстояние от каждой из границ до границ соседних элементов.

Последнее свойство, которое может быть полезно при работе с элементами, — это Cursor. Оно позволяет задать форму курсора:

```
btn.Cursor = Cursors.Wait;
```

Кнопки

В Silverlight представлено целых пять видов кнопок:

- **Button** — это классическая кнопка, содержащая в качестве контента любой объект типа **UIElement** и инициирующая событие **Click**;

- **ToggleButton** — этот тип кнопки используется для имитации эффекта залипания. Кнопка имеет два состояния: нажата и отпущена. Чтобы обрабатывать изменение состояния кнопки, используется событие **Checked**;
- **RepeatButton** — в отличие от классической кнопки, этот вид кнопки позволяет непрерывно генерировать событие **Click**, если кнопка остается нажатой, то есть удерживается пользователем в этом состоянии;
- **CheckBox** — эта кнопка является прямым наследником от **ToggleButton** и реализует простейший флажок. В отличие от **ToggleButton,** флажок можно устанавливать в промежуточное состояние, но только из кода и только если свойство **IsThreeState** установлено в **true**;
- **RadioButton** — этот элемент управления полностью аналогичен **CheckBox**, но с возможностью размещения целого набора элементов в группы, позволяя сделать единственный выбор.

Вот небольшой пример, демонстрирующий создание группы элементов **RadioButton**:

```
<Border BorderThickness="3" BorderBrush="Black" Padding="5">
    <StackPanel>
        <RadioButton Content="Choice 1" IsChecked="True"
            GroupName="Group 1" Margin="5">
        </RadioButton>
        <RadioButton Content="Choice 1" IsChecked="False"
            GroupName="Group 1" Margin="5">
        </RadioButton>
        <RadioButton Content="Choice 1" IsChecked="False"
            GroupName="Group 1" Margin="5">
        </RadioButton>
        <RadioButton Content="Choice 1" IsChecked="False"
            GroupName="Group 1" Margin="5">
        </RadioButton>
    </StackPanel>
</Border>
```

Как мы и говорили выше, особенностью элементов управления в Silverlight состоит в том, что в качестве содержимого у любого из элементов (за редким исключением) может выступать контейнер или другой элемент. Ниже пример кнопки, отображающей видео:

Рис. 5.11. Кнопка, отображающая видео

```
<Button MaxHeight="50" MaxWidth="100">
    <Button.Content>
        <MediaElement Source="Wildlife.wmv"></MediaElement>
    </Button.Content>
</Button>
```

Текстовые элементы управления

До выхода Silverlight 4 можно было говорить лишь о двух текстовых элементах управления: **TextBox** и **PasswordBox**. Начиная с четвертой версии Silverlight, разработчикам стал доступен элемент **RichTextArea**, который мы рассматривали в первой главе.

Среди основных свойств текстовых элементов управления можно выделить:

- **AcceptsReturn** — позволяет выполнять перевод каретки при вводе текста;
- **IsReadOnly** — определяет, будет ли текстовое поле доступно для ввода;
- **SelectedText** — возвращает выделенный текст;
- **SelectionLength** — позволяет получить или задать размер текущего выделения;
- **SelectionStart** — позволяет получить или задать позицию символа, с которого нужно произвести выделение;
- **SelectionBackground** — определяет цвет фона выделенного текста;
- **SelectionForeground** — задает цвет шрифта выделенного текста;
- **TextWraping** — определяет, будет ли текст переходить на другую строку, если он не помещается в видимой части одной строки.

```
<TextBox AcceptsReturn="True" TextWrapping="Wrap" Width="300"
Height="200">
</TextBox>
```

Еще одной небольшой особенностью Silverlight является возможность установки каретки для текстового поля и поля для задания пароля (**TextBox** и **PasswordBox**). Дело в том, что элементы в Silverlight реализуются таким образом, что представление отделено от логики. Поэтому разработчик может всегда заменить внешний вид уже существующего элемента. Например, можно создать треугольную кнопку или овальное поле для ввода текста. Но в элементах **TextBox** и **PasswordBox** одна составляющая не поддавалась изменениям — каретка. Она была исключительно черной. Даже если разработчик просто менял цвет поля ввода, каретку не всегда было видно. Теперь каретка может использовать любую из доступных кистей для заливки. Разработчик может просто поменять цвет, а может и установить видеоролик в качестве каретки. Вот пример использования свойства:

```
<TextBox Text="Text" CaretBrush="Blue"></TextBox>
<TextBox Text="Text" >
    <TextBox.CaretBrush>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Color="Yellow" Offset="0.0" />
            <GradientStop Color="Red" Offset="0.25" />
            <GradientStop Color="Blue" Offset="0.75" />
            <GradientStop Color="LimeGreen" Offset="1.0" />
        </LinearGradientBrush>
    </TextBox.CaretBrush>
    <TextBox.RenderTransform>
        <ScaleTransform ScaleX="3" ScaleY="3"/>
    </TextBox.RenderTransform>
</TextBox>
```

Элементы управления списками

Если предыдущие элементы наследовались от **ContentControl**, то все элементы, позволяющие управлять списками элементов, наследуются от **ItemsControl**. В Silverlight 4 выделяют четыре таких элемента:

- **ListBox** — позволяет отобразить список значений с возможностью выбора одного или нескольких значений;
- **ComboBox** — позволяет отобразить выпадающий список, с возможностью сделать единственный выбор;
- **TabControl** — этот элемент управления не является стандартным, а входит в SDK, в сборку **System.Windows.Controls.dll**. Его задача — отобразить несколько панелей с заголовками и обеспечить возможность перехода по панелям;
- **TreeView** — как и предыдущий элемент, **TreeView** содержится в SDK и позволяет отобразить содержимое в виде дерева.

Среди свойств элементов **ListBox** и **ComboBox** можно выделить следующие:

- **Items** — позволяет задать коллекцию элементов типа **ListBoxItem**, который будут использоваться для отображения информации в **ListBox**;
- **ItemsSource** — используется для привязки **ListBox** к коллекции элементов, реализующих интерфейс **IEnumerable**;
- **DisplayMemberPath** — задает свойство элемента коллекции, определенной через **ItemsSource**, которое будет использоваться для отображения в **ListBox**. Если свойство не задано, то у объекта будет вызван метод **ToString**;
- **ItemTempate** — позволяет задать шаблон элемента внутри списка. Иными словами, позволяет определить формат выдачи;
- **SelectedItem**(s) — возвращает выбранный (или выбранные) элементы в **ListBox**. Очень важное свойство, позволяющее преобразовать полученный объект к типу, хранимому в **ListBox**. Это позволяет нам не хранить ссылку на исходную коллекцию элементов;
- **SelectedValue** — возвращает выбранное значение (тут уже речь идет о тексте).

Рассмотрим небольшой пример описания **ListBox** и привязки к нему коллекции элементов:

```
<UserControl x:Class="Chapter5_Test.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             Loaded="UserControl_Loaded">
    <StackPanel HorizontalAlignment="Center">
        <ListBox Name="myList" Width="200" Height="100">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <CheckBox IsChecked="{Binding IsActive}"
                              Content="{Binding Name}">
                    </CheckBox>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
```

```
    </StackPanel>
</UserControl>
```

Тут мы определили шаблон элемента управления. Теперь опишем содержимое классов, создающее набор тестовых данных и реализующее привязку коллекции с данными к списку:

```
public partial class MainPage : UserControl
{
    public MainPage()
    {
        InitializeComponent();
    }

    private void UserControl_Loaded(object sender, RoutedEventArgs e)
    {
        List<Employee> l=new List<Employee>();

        Employee emp = new Employee();
        emp.Name = "Sergiy Baydachnyy";
        emp.IsActive = true;
        l.Add(emp);

        emp = new Employee();
        emp.Name = "Viktor Baydachnyy";
        emp.IsActive = false;
        l.Add(emp);

        myList.ItemsSource = l;
    }
}

public class Employee
{
    public string Name { get; set; }
    public bool IsActive { get; set; }
}
```

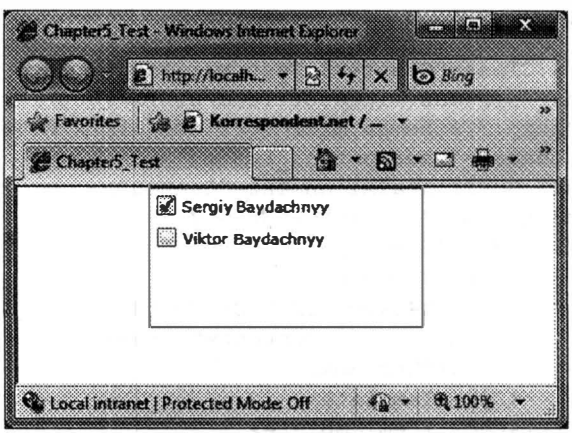


Рис. 5.12. Использование элемента управления ListBox

Результат работы приложения показан на рис. 5.12.

Кроме перечисленных свойств, элемент управления **ListBox** обладает свойством **SelectionMode**. Это свойство может содержать одно из трех значений: **Single**, **Multiple**, **Extended**. При установке свойства **Multiple** пользователь может выбирать несколько значений простым кликом, а при выборе **Extended** также доступны функциональные клавиши и поведение больше похоже на полюбившийся **ListBox** из Windows Forms или WPF.

Элементы управления, основанные на диапазоне значений

В Silverlight 4 присутствует три элемента управления, которые основаны на диапазоне значений, — это **Slider**, **ScrollBar** и **ProgressBar**. Все три элемента основаны на классе **RangeBase** и обладают следующими свойствами:

- **Minimum** — определяет минимальное значение диапазона;
- **Maximum** — определяет максимальное значение диапазона;
- **Value** — определяет текущее положение ползунка или процент заполнения элемента **ProgressBar**.

Отличие между этими элементами состоит только в том, что **Slider** и **ScrollBar** позволяют перемещать ползунок внутри диапазона, а **ProgressBar** способен отображать только свое состояние, которое может меняться только в коде.

Ниже показан пример, демонстрирующий работу сразу трех элементов **Slider**:

```
<StackPanel x:Name="LayoutRoot" Background="White">
    <MediaElement Source="WildLife.wmv" Width="400" Height="300">
        <MediaElement.Projection>
            <PlaneProjection
             RotationX=
              "{Binding Value, ElementName=rotateXSlider,
Mode=OneWay}"
             RotationY=
              "{Binding Value, ElementName=rotateYSlider,
Mode=OneWay}"
             RotationZ=
               "{Binding Value, ElementName=rotateZSlider,
Mode=OneWay}">
            </PlaneProjection>
        </MediaElement.Projection>
    </MediaElement>
    <Slider Width="400" Minimum="0" Maximum="360"
        Name="rotateXSlider">
    </Slider>
    <Slider Width="400" Minimum="0" Maximum="360"
        Name="rotateYSlider">
    </Slider>
    <Slider Width="400" Minimum="0" Maximum="360"
        Name="rotateZSlider">
    </Slider>
</StackPanel>
```

Элемент управления ToolTip

Следующий интересный элемент управления — это **ToolTip**, который позволяет отобразить всплывающее окно с подсказкой. Он может быть привязан к любому **UIElement** и содержать практически все что угодно для отображения:

```
<Button Content="Detach" Width="100" Click="Button_Click">
  <ToolTipService.ToolTip>
    <ToolTip Placement="Right">
      <ToolTip.Content>
        <MediaElement Source="4.wmv"></MediaElement>
      </ToolTip.Content>
    </ToolTip>
  </ToolTipService.ToolTip>
</Button>
```

ToolTip может быть привязан как к одной из границ элемента, так и к координатам курсора мыши. Привязка задается с помощью свойства **Placement**.

Использование диалоговых окон

Кроме обычных элементов управления, составляющих основу любого интерфейса, Silverlight обладает двумя специальными диалоговыми окнами, которые позволяют приложению работать с файловой системой пользователя, — это **OpenFileDialog** и **FileSaveDialog**.

Ниже показан простой пример, позволяющий записать файл, содержащий текст Hello, на диск:

```
SaveFileDialog file= new SaveFileDialog();
file.Filter = "Text File | *.txt";
file.DefaultExt = ".txt";
file.ShowDialog();
if (file.File.Name != "")
{
     System.IO.StreamWriter s =
         new System.IO.StreamWriter(file.OpenFile());
     s.Write("Hello");
     s.Close();
}
```

Заключение

Если взять все элементы управления в Silverlight, то их можно насчитать больше сотни. Учитывая то, что практически все элементы позволяют задавать свое собственное представление (глава 11), то даже с этим набором элементов возможности по созданию интерфейсов практически безграничны. Нужно учитывать, что набор элементов управления расширяется с выходом очередной версии Silverlight (глава 1), а использование таких проектов, как http://codeplex.com/Toolkit, позволяет получить доступ к востребованным элементам еще до выхода очередного обновления SDK.

Глава 6

ПРИВЯЗКА К ДАННЫМ

Каждый раз, разрабатывая приложение, разработчик сталкивается с необходимостью отобразить данные. При этом всегда хочется, чтобы код был минимальным и не требовал больших усилий.

В Silverlight 4 есть мощный механизм привязки данных к свойствам элементов управления, который позволяет значительно минимизировать код и упростить работу с данными. Рассмотрим этот механизм привязки к данным и начнем с самого простого случая, когда свойство одного из элементов управления связано со свойством другого элемента управления.

Привязка к свойству элемента управления

Начнем с того, что для реализации привязки используется объект типа **Binding**. Независимо от того, связываете ли Вы элементы или элемент и данные, всегда используется именно **Binding**. При этом **Binding** можно совершенно спокойно использовать как в коде, так и в разметке XAML.

Естественно, что использование **Binding** в XAML — самая распространенная ситуация. Для этих целей в XAML существует специальное расширение разметки, о котором мы немного уже упоминали в главе 4. Рассмотрим небольшой пример:

```
<UserControl x:Class="Chapter6_Binding.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <StackPanel x:Name="LayoutRoot" Background="White">
        <Image Source="Hydrangeas.jpg" Width="400">
            <Image.Projection>
                <PlaneProjection RotationY=
                   "{Binding Value, ElementName=slider}">
                </PlaneProjection>
            </Image.Projection>
        </Image>
        <Slider Minimum="0" Maximum="360" Name="slider"
            Width="400" Margin="10"></Slider>
    </StackPanel>
</UserControl>
```

Тут мы создали элемент управления **Image**, который хотим «вращать» по оси Y. Для создания эффекта размещения элемента в трехмерном пространстве используется объект **PlaneProjection**, содержащий свойство **RotationY**, которое задает угол поворота элемента по оси Y. Чтобы придать динамичности нашему интерфейсу, вторым элементом мы добавили ползунок, который и будет задавать угол поворота. Тут мы используем два параметра:

- **Path** — позволяет задать свойство источника, с которым происходит связывание. Поскольку это свойство является свойством по умолчанию, то явно Path можно не писать;
- **ElementName** — задает имя элемента-источника;

В данном примере независимо от того, как Вы модифицируете свойство **Value** бегунка, свойство **RotationY** будет обновляться автоматически.

Привязку можно реализовать и по-другому:

```
<UserControl x:Class="Chapter6_Binding.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <StackPanel x:Name="LayoutRoot" Background="White">
        <Image Source="Hydrangeas.jpg" Width="400">
            <Image.Projection>
                <PlaneProjection
x:Name="projection"></PlaneProjection>
            </Image.Projection>
        </Image>
        <Slider Minimum="0" Maximum="360" Name="slider"
            Width="400" Margin="10"
            Value=
             "{Binding RotationY, ElementName=projection,
Mode=TwoWay}">
        </Slider>
    </StackPanel>
</UserControl>
```

В этом примере в качестве источника выступает изображение. Отличие состоит в том, что выполняя привязку ползунка к изображению, мы указали дополнительное свойство **Mode**. Это свойство может принимать одно из трех значений:

- **OneTime** — значение свойства устанавливается на основании значения свойства источника, но установка происходит лишь в момент создания объектов. Любые изменения в будущем игнорируются;
- **OneWay** — значение свойства устанавливается на основании значения свойства источника. При изменении свойства источника будет обновляться свойство основного объекта;
- **TwoWay** — значение свойства устанавливается на основании значения свойства источника. При изменении свойства источника или свойства основного объекта, будут происходить взаимные обновления.

В нашем примере мы установили значение **TwoWay** свойству **Mode**. Таким образом, несмотря на то, что картинка является источником, ее поворот успешно задается ползунком.

Выбор источника зависит от конкретного приложения, так, в примере выше, выбор источника был не принципиален, но если мы решим добавить дополнительный элемент, модифицирующий значение угла поворота, то установить свойству два элемента **Binding** нам не удастся, а вот изменить направление привязки — без проблем. Пример ниже расширяет наш интерфейс текстовым полем, которое также задает угол поворота.

```
<UserControl x:Class="Chapter6_Binding.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <StackPanel x:Name="LayoutRoot" Background="White">
        <Image Source="Hydrangeas.jpg" Width="400">
            <Image.Projection>
                <PlaneProjection
x:Name="projection"></PlaneProjection>
            </Image.Projection>
        </Image>
        <Slider Minimum="0" Maximum="360" Name="slider"
            Width="400" Margin="10"
            Value=
            "{Binding RotationY, ElementName=projection,
Mode=TwoWay}">
        </Slider>
        <TextBox Width="200"  Text=
            "{Binding RotationY, ElementName=projection,
            Mode=TwoWay}"></TextBox>
    </StackPanel>
</UserControl>
```

Обратите внимание, что в этом примере действие значения введенного в **TextBox** возымеет силу лишь при потере фокуса элементом **TextBox**. Это связано с тем, что поведение свойств при реализации привязки может отличаться. Для того чтобы задать поведение свойства, используют метаданные (атрибуты). В свою очередь, метаданные для **TextBox** заданы таким образом, что требуют потерю фокуса текстового поля, прежде чем производить обновление. Для элемента **TextBox** это вполне обосновано. В принципе, подобное поведение можно изменить, используя дополнительное свойство объекта **Binding** — **UpdateSourceTrigger**. В отличие от WPF, это свойство способно принимать лишь одно из двух значений:

- **Default** — это значение задано по умолчанию и позволяет элементу самому определить, как ему обновляться и обновлять цель привязки;
- **Explicit** — это значение позволяет указать, что источник не будет считаться обновленным, до тех пор, пока не будет вызван специальный метод **UpdateSource** объекта **BindingExpression**.

Рассмотрим поведение **Binding** при заданном **UpdateSourceTrigger** на следующем примере:

```
<UserControl x:Class="Chapter6_Binding.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <StackPanel x:Name="LayoutRoot" Background="White">
```

```
        <Image Source="Hydrangeas.jpg" Width="400">
            <Image.Projection>
                <PlaneProjection
x:Name="projection"></PlaneProjection>
            </Image.Projection>
        </Image>
        <Slider Minimum="0" Maximum="360" Name="slider"
            Width="400" Margin="10"
            Value=
               "{Binding RotationY, ElementName=projection,
Mode=TwoWay}">
        </Slider>
        <StackPanel Orientation="Horizontal">
            <TextBox Width="200" Name="txtBox" Text=
               "{Binding RotationY, ElementName=projection,
Mode=TwoWay, UpdateSourceTrigger=Explicit}"
               Margin="5">
            </TextBox>
            <Button Width="100" Content="Update" Margin="5"
               Click="Button_Click">
            </Button>
        </StackPanel>
    </StackPanel>
</UserControl>
```

Тут мы установили **UpdateSourceTrigger** в **Explicit** и добавили кнопку, которая и будет обновлять цель. Обратите внимание, что если Вы ввели что-то в текстовое поле и меняете фокус, то никаких изменений не происходит.

Чтобы обновить цель на основании данных в источнике, реализуем следующий код:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    BindingExpression exp =
       txtBox.GetBindingExpression(TextBox.TextProperty);
    exp.UpdateSource();
}
```

Тут мы получили ссылку на элемент **BindingExpression**, ассоциирующийся с нашей привязкой, используя метод **GetBindingExpression**, который доступен у любого **FrameworkElement**. В качестве параметра этот метод получает имя свойства, которое выступает источником. Остается вызвать метод **UpdateSource**, который обновляет данные об источнике, а, следовательно, осуществляется изменение данных и у цели.

Обратите внимание на то, что мы ни в одном из примеров не обрабатываем ошибки привязки. Так, если в последнем примере ввести набор символов, а затем нажать кнопку Update, то ничего не произойдет. Все ошибки привязки «глотаются» приложением. Чтобы обнаружить такую ошибку, можно запустить приложение в режиме отладки и обратиться к окну **Output**.

Ниже мы поговорим о том, как обрабатывать подобные ошибки.

Рис. 6.1. Содержимое окна Output при возникновении ошибки

Наконец, если Вы хотите установить привязку к данным в коде, то это также можно сделать без проблем. Вот как будет выглядеть код для **TextBox** из нашего примера:

```
Binding binding = new Binding();
binding.ElementName = "projection";
binding.Path = new PropertyPath("RotationY");
binding.Mode = BindingMode.TwoWay;
binding.UpdateSourceTrigger = UpdateSourceTrigger.Explicit;
txtBox.SetBinding(TextBox.TextProperty, binding);
```

Перейдем теперь от привязки к элементам к привязке к объектам, которые не являются элементами управления.

Привязка к объекту

На практике значительно чаще приходится выполнять привязку элементов к данным, которые содержатся в объекте некоторого класса. Эти данные могут быть получены из базы данных или сгенерированы во время работы приложения.

Создадим простой класс, описывающий информацию о сотруднике:

```
class Employee
{
    public string FirstName
    {
        get;
        set;
    }

    public string LastName
    {
        get;
        set;
    }

    public string EMail
    {
        get;
        set;
```

```
        public int Age
        {
            get;
            set;
        }
    }
```

Этот класс содержит 4 свойства с модификатором public. Это основное требование при привязке к данным: свойства объектов, которые выступают в качестве источника, должны быть общедоступными.

Следующий код создает простую форму и связывает эту форму с объектом типа Employee, который мы определили в ресурсах (аналогично можно определить и в коде):

```
<UserControl x:Class="Chapter6_Binding.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:data="clr-namespace:Chapter6_Binding">
    <UserControl.Resources>
        <data:Employee x:Key="empl"
           FirstName="Sergiy"
           LastName="Baydachnyy"
           Age="31"
           EMail="sbaidachni@gmail.com">
        </data:Employee>
    </UserControl.Resources>
    <Grid x:Name="LayoutRoot" Background="White" Width="400">
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>

        <TextBlock Text="First Name:" Grid.Row="0" Grid.Column="0">
        </TextBlock>
        <TextBox Text=
            "{Binding FirstName, Source={StaticResource empl},
Mode=TwoWay}"
            Grid.Row="0" Grid.Column="1">
        </TextBox>

        <TextBlock Text="Last Name:" Grid.Row="1" Grid.Column="0">
        </TextBlock>
        <TextBox Text=
            "{Binding LastName, Source={StaticResource empl},
Mode=TwoWay}"
            Grid.Row="1" Grid.Column="1">
        </TextBox>
```

```
        <TextBlock Text="EMail:" Grid.Row="2" Grid.Column="0">
        </TextBlock>
        <TextBox Text=
            "{Binding EMail, Source={StaticResource empl},
Mode=TwoWay}"
            Grid.Row="2" Grid.Column="1">
        </TextBox>

        <TextBlock Text="Age:" Grid.Row="3" Grid.Column="0">
        </TextBlock>
        <TextBox Text=
           "{Binding Age, Source={StaticResource empl},
Mode=TwoWay}"
           Grid.Row="3" Grid.Column="1">
        </TextBox>
    </Grid>
</UserControl>
```

Как видите, механизм точно такой, как и при привязке к элементам. Только вместо **ElementName** используется свойство **Source**, которое задает ссылку на объект (в данном случае выбираемый из ресурсов).

Модифицируем код выше, используя еще одно полезное свойство — **DataContext**:

```
<UserControl x:Class="Chapter6_Binding.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:data="clr-namespace:Chapter6_Binding">
    <UserControl.Resources>
        <data:Employee x:Key="empl"
           FirstName="Sergiy"
           LastName="Baydachnyy"
           Age="31"
           EMail="sbaidachni@gmail.com">
        </data:Employee>
    </UserControl.Resources>
    <Grid x:Name="LayoutRoot" Background="White" Width="400"
        DataContext={StaticResource empl}>
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>

        <TextBlock Text="First Name:" Grid.Row="0" Grid.Column="0">
        </TextBlock>
        <TextBox Text="{Binding FirstName, Mode=TwoWay}"
            Grid.Row="0" Grid.Column="1">
```

```
        </TextBox>

        <TextBlock Text="Last Name:" Grid.Row="1" Grid.Column="0">
        </TextBlock>
        <TextBox Text="{Binding LastName, Mode=TwoWay}"
            Grid.Row="1" Grid.Column="1">
        </TextBox>

        <TextBlock Text="EMail:" Grid.Row="2" Grid.Column="0">
        </TextBlock>
        <TextBox Text="{Binding EMail, Mode=TwoWay}"
            Grid.Row="2" Grid.Column="1">
        </TextBox>

        <TextBlock Text="Age:" Grid.Row="3" Grid.Column="0">
        </TextBlock>
        <TextBox Text="{Binding Age, Mode=TwoWay}"
           Grid.Row="3" Grid.Column="1">
        </TextBox>
    </Grid>
</UserControl>
```

Как Вы видите, нам немного удалось упростить код. Дело в том, что если не указать имя объекта с данными явно, то механизм привязки начинает перебор свойств **DataContext** всех родительских элементов, пока найдет не пустой. Найденный **DataContext** механизм связывания и используется??? в качестве источника.

Замечание. Если Вы будете использовать **DataContext** в коде, то класс Employee должен быть только **public**. При использовании ч??ерез ресурсы можно обойтись и без **public**.

Итак, с первого взгляда, наш код работает правильно и вполне прогнозируемо. Но, если Вы попробуете изменить хотя бы одно свойство объекта типа Employee в коде, то обнаружите, что значение в соответствующем поле не поменялось. Так, уберите **DataContext** из XAML кода, а вместо него реализуйте обработчик события **Loaded**:

```
private void UserControl_Loaded(object sender, RoutedEventArgs e)
{
    Employee emp = new Employee();
    emp.FirstName = "Sergiy";
    emp.LastName = "Baydachnyy";
    emp.EMail = "sbaidachni@gmail.com";
    emp.Age = 31;

    LayoutRoot.DataContext = emp;

    emp.FirstName = "Viktor";
}
```

Тут мы создаем объект типа Employee, связываем его с формой, а затем изменяем одно из свойств. Если запустить это пример, то можно обнаружить, что изменения не попали в форму. Это связано с тем, что у механизма при-

вязки нет возможности проверять значения свойства объекта с данными. При этом для элементов управления механизм работал нормально, так как базировался на понятии зависимых свойств.

Если мы хотим видеть ожидаемое поведение, необходимо реализовать специальный интерфейс **INotifyPropertyChanged**. Реализация этого интерфейса позволит механизму привязки данных реагировать на событие, связанное с изменением всех свойств. Правда инициировать событие придется самостоятельно. Вот как будет выглядеть класс Employee, реализующий **INotifyPropertyChanged**:

```
public class Employee: INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public void OnPropertyChanged(PropertyChangedEventArgs e)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, e);
    }

    private string firstName;
    private string lastName;
    private int age;
    private string email;

    public string FirstName
    {
        get { return firstName; }
        set
        {
            firstName = value;
            OnPropertyChanged(new
PropertyChangedEventArgs("FirstName"));
        }
    }

    public string LastName
    {
        get { return lastName; }
        set
        {
            lastName = value;
            OnPropertyChanged(new
PropertyChangedEventArgs("LastName"));
        }
    }

    public string EMail
    {
        get { return email; }
        set
        {
            email = value;
```

```
            OnPropertyChanged(new
PropertyChangedEventArgs("EMail"));
        }
    }

    public int Age
    {
        get { return age; }
        set
        {
            age = value;
            OnPropertyChanged(new PropertyChangedEventArgs("Age"));
        }
    }
}
```

Вот теперь приложение будет работать так, как и ожидалось.

Привязка к коллекции

При привязке простых свойств обычно не возникает проблем. Но вот привязка коллекций требует дополнительных усилий.

Расширим класс Employee, перегрузив метод ToString:

```
public override string ToString()
{
    return String.Format("{0} {1}", firstName, lastName);
}
```

Этот метод нам понадобится при заполнении одного из элементов управления коллекцией элементов типа Employee.

Хочу сразу отметить, что коллекции способны принимать только те элементы, которые являются наследниками от **ItemsControl**. Среди таких элементов можно выделить и **ListBox**, который способен отображать список элементов.

Прежде чем писать код, давайте определимся со свойствами, которые используются при привязке коллекции к элементу. Тут существуют три свойства:

- **ItemsSource** — задает коллекцию элементов. Если не задан **DisplayMemberPath**, то будет попытка вызвать метод **ToString** у каждого из элементов. Вот почему мы и преопределили **ToString**;
- **ItemTemplate** — используется для задания шаблона отображения конкретного элемента;
- **DisplayMemberPath** — задает конкретное свойство для отображения в элементе управления.

А вот теперь перейдем к коду. Сначала расширим наш интерфейс, отобразив **ListBox**:

```
<UserControl x:Class="Chapter6_Binding.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Loaded="UserControl_Loaded">
    <StackPanel>
        <ListBox Height="300" Name="lstEmp"></ListBox>
    <Grid x:Name="LayoutRoot" Background="White" Width="400"
        DataContext="{Binding SelectedItem, ElementName=lstEmp}">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <TextBlock Text="First Name:" Grid.Row="0" Grid.Column="0">
        </TextBlock>
        <TextBox Text="{Binding Path=FirstName}"  Grid.Row="0"
            Grid.Column="1">
        </TextBox>
        <TextBlock Text="Last Name:" Grid.Row="1" Grid.Column="0">
        </TextBlock>
        <TextBox Text="{Binding Path=LastName}"  Grid.Row="1"
            Grid.Column="1">
        </TextBox>
        <TextBlock Text="EMail:" Grid.Row="2" Grid.Column="0">
        </TextBlock>
        <TextBox Text="{Binding Path=EMail}"  Grid.Row="2"
Grid.Column="1">
        </TextBox>
        <TextBlock Text="Age:" Grid.Row="3" Grid.Column="0">
        </TextBlock>
        <TextBox Text="{Binding Path=Age}"  Grid.Row="3"
Grid.Column="1">
        </TextBox>
    </Grid>
    </StackPanel>
</UserControl>
```

Обратите внимание на то, что контекст для формы с детальными данны ми мы выбираем из элемента **ListBox**, используя привязку.

Теперь реализуем код, создающий список элементов. В качестве структу ры данных можно выбрать любой класс, главное, чтобы он реализовывал ин терфейс **IEnumerable**.

```
private void UserControl_Loaded(object sender, RoutedEventArgs e)
{
    List<Employee> l = new List<Employee>();

    Employee emp = new Employee();
    emp.FirstName = "Sergiy";
    emp.LastName = "Baydachnyy";
```

```
        emp.EMail = "sbaidachni@gmail.com";
        emp.Age = 31;

        l.Add(emp);

        emp = new Employee();
        emp.FirstName = "Viktor";
        emp.LastName = "Baydachnyy";
        emp.EMail = "viktor@gmail.com";
        emp.Age = 30;

        l.Add(emp);

        lstEmp.ItemsSource = l;
    }
```

Как Вы видите, мы создали два элемента и выполнили привязку к эле менту **ListBox**, получив полностью рабочий пример!

Результат работы приложения можно увидеть на рисунке ниже:

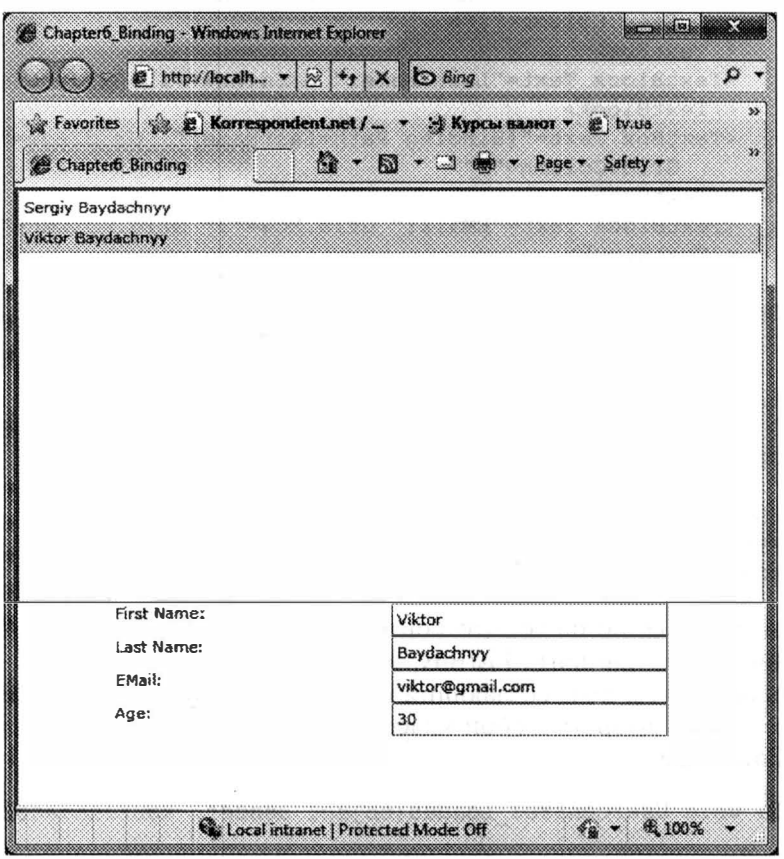


Рис. 6.2. Результат работы приложения

Однако если Вы попробуете изменить конкретный элемент через его де тальную форму, то сможете увидеть, что введенные изменения сохраняютс

но **ListBox** упорно не хочет их отображать. Это все от того, что **ListBox** реагирует лишь на изменение значений только тех свойств, с которыми он связан явно. То есть, если мы меняем свойство LastName, то инициируется событие **OnPropertyChange**, но **ListBox** никакого дела до него нет, так как он использует метод **ToString** и не связан с конкретным свойством (или свойствами). Чтобы исправить ситуацию, можно воспользоваться методом **DisplayMemberPath**, указав имя свойства явно. Все дело в том, что мы хотим отобразить комбинацию имени и фамилии, а такого свойства у нас нет. Определять же новое свойство было бы глупо. Поэтому мы воспользуемся возможностью **ListBox** по определению шаблона выдаваемых элементов, заполнив свойство **ItemTemplate**. Вот как будет выглядеть элемент **ListBox**:

```
<ListBox Height="300" Width="400" Name="lstEmp">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Border Background="Gray" BorderThickness="2"
                CornerRadius="10" Width="396" Height="30">
                <StackPanel Orientation="Horizontal" Margin="5">
                    <TextBlock Text="{Binding
FirstName}"></TextBlock>
                    <TextBlock Text=" "></TextBlock>
                    <TextBlock Text="{Binding
LastName}"></TextBlock>
                </StackPanel>
            </Border>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Тут мы не только решили проблему с отображением измененных данных, но и изменили внешний вид элемента в списке на более симпатичный.

В завершение раздела хочу отметить, что вместо класса **List** лучше использовать коллекцию **ObservableCollection**. Дело в том, что если Вы решите добавлять или удалять элементы из списка, то столкнётесь с необходимостью реализации интерфейса **INotifyCollectionChanged**. А **ObservableCollection** уже реализует этот интерфейс.

Конвертеры данных

Выше мы рассматривали простую «переброску» данных из одного места в другое. Но существует достаточно много ситуаций, когда данные не могут быть преобразованы с помощью вызова метода **ToString** или с помощью стандартных конвертеров.

Рассмотрим пример, в котором определим в нашем классе Employee дополнительное свойство Salary.

```
private double salary;

public double Salary
```

```
{
    get
    {
        return salary;
    }
    set
    {
        salary = value;
        OnPropertyChanged(new PropertyChangedEventArgs("Salary"));
    }
}
```

Поскольку речь идет о зарплате, то было бы логично выдавать значение этого свойства со знаком местной валюты. Следовательно, необходимо реализовать специальный механизм, который будет заниматься преобразованием данных. К счастью, в Silverlight это сделать достаточно просто. Нужно просто определить класс, реализующий интерфейс **IValueConverter**, который описывает всего два метода: **Convert** и **ConvertBack**. Первый метод используется для преобразования значения от источника к цели, а второй — от цели к источнику.

Ниже приведен пример реализации класса, позволяющего делать преобразование значения типа double к денежному представлению и наоборот.

```
public class MoneyConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
        object parameter, CultureInfo culture)
    {
        return ((double)value).ToString("C");

    }

    public object ConvertBack(object value, Type targetType,
        object parameter, CultureInfo culture)
    {
        double result;
        try
        {
            result = double.Parse((string)value,
                                  NumberStyles.AllowThousands |
                                  NumberStyles.AllowDecimalPoint |

NumberStyles.AllowCurrencySymbol);
        }
        catch
        {
            result = 0;
        }
        return result;
```

Чтобы подключить наш конвертер к привязке, воспользуемся свойством **Converter** объекта **Binding**. Есть только одно «но» — это свойство принимает ссылку на готовый объект, которого у нас еще нет. Поэтому определим объект созданного класса в ресурсах приложения:

```
    <UserControl.Resources>
        <data:MoneyConverter
x:Key="mConverter"></data:MoneyConverter>
    </UserControl.Resources>
```

Тут **data** представляет собой имя пространства имен, ссылающегося на пространство имен кода C#.

```
xmlns:data="clr-namespace:Chapter6_Binding"
```

А вот теперь можно определить привязку свойства Salary к очередному текстовому полю:

```
<TextBlock Text="Salary:" Grid.Row="4" Grid.Column="0"></TextBlock>
<TextBox Text="{Binding Path=Salary,
                Converter={StaticResource mConverter},
                Mode=TwoWay}"
   Grid.Row="4" Grid.Column="1">
</TextBox>
```

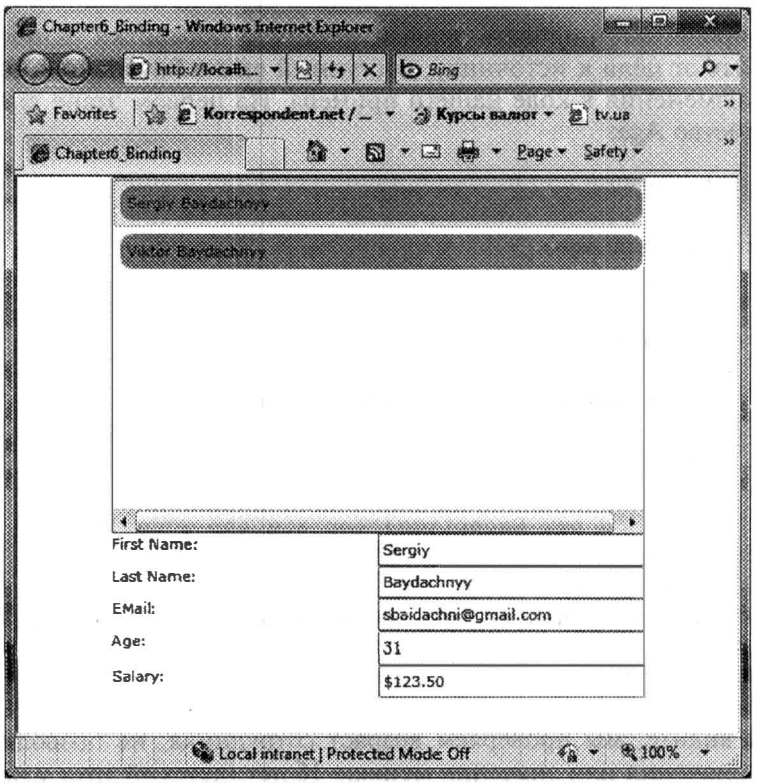


Рис. 6.3. Результат работы приложения

Результат работы приложения показан на рис. 6.3.

В этом же разделе рассмотрим еще три свойства класса Binding, которые могут быть полезны при преобразовании данных источника к цели и наоборот:

- **FallbackValue** — с помощью этого свойства можно задать значение, которое будет отображаться, в случае неудачной привязки (когда преобразование пошло как-то не так);
- **TargetNullValue** — это свойство позволяет задать значение, которое будет отображаться в том случае, если значение источника равно **NULL**;
- **StringFormat** — определяет формат выдачи значений. Например, в примере выше мы могли бы не создавать конвертер, а указать **StringFormat=C**.

Проверка данных при связывании

Полностью разобравшись с механизмом привязки данных, перейдем к последней теме, связанной с проверкой данных и обработкой ошибок при связывании. В отличие от Silverlight 3, где предлагался только один механизм, Silverlight 4 представляет сразу три механизма обработки ошибок при привязке данных. Рассмотрим каждый из них отдельно.

ValidatesOnExceptions и NotifyOnValidationError

Чтобы продемонстрировать первый механизм обработки ошибок при передаче данных от цели к источнику, доступный еще в Silverlight 3, рассмотрим небольшие изменения в коде нашего примера. На первом этапе внесем изменение в свойство Age:

```
public int Age
{
    get { return age; }
    set
    {
        if (value <= 0)
        {
            throw new Exception("Age should be positive!");
        }
        else if (value <= 16)
        {
            throw new Exception("You are too young!");
        }
        age = value;
        OnPropertyChanged(new PropertyChangedEventArgs("Age"));
    }
}
```

Как Вы видите, мы проверяем возраст сотрудника, не позволяя вводить негативные значения или значения меньше 16 лет (детский труд у нас использовать запрещено), генерируя разные исключения.

Если Вы попробуете запустить данный пример и ввести неверное значение, то ничего не произойдет, хотя и значение не сохранится. Система привязки данных полностью глотает все исключения. Она попросту не знает, как на них реагировать. Поэтому, чтобы обеспечить нотификацию пользователя о том, что он ввел неверные данные, мы используем специальное свойство **ValidatesOnExceptions**, которое также входит в класс **Binding**. Вот как будет выглядеть код XAML, обеспечивающий привязку возраста к текстовому полю:

```
<TextBlock Text="Age:" Grid.Row="3" Grid.Column="0"></TextBlock>
<TextBox Text=
    "{Binding Path=Age, Mode=TwoWay, ValidatesOnExceptions=True}"
    Grid.Row="3" Grid.Column="1">
</TextBox>
```

Добавив всего одно свойство и запустив пример, можно увидеть следующую картину при попытке ввести неверные данные:

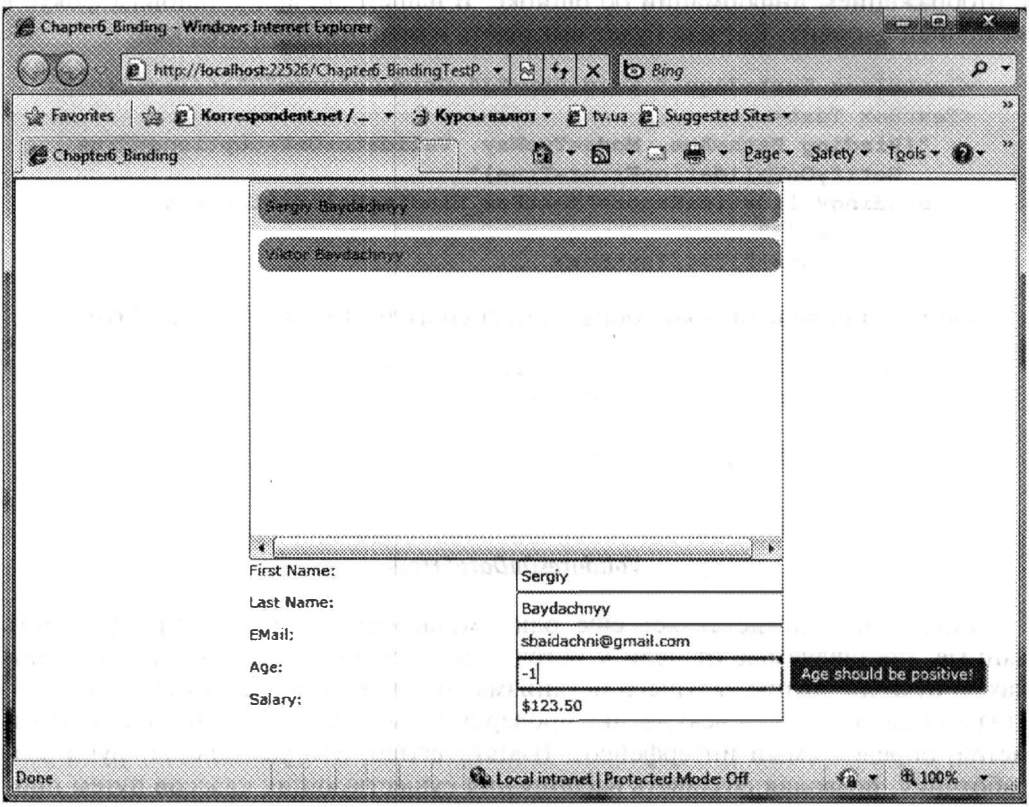


Рис. 6.4. Результат работы приложения

Как видно, элемент управления выделяется красным прямоугольником, а при наведении фокуса, возникает **ToolTip**, который отображает текст исключения.

Обычно разработчики спрашивают, можно ли изменить поведение элемента, заданное по умолчанию. Например, я хочу видеть не красный прямо-

угольник, а зеленый (а может и не прямоугольник). Несомненно, можно, но делается это через шаблоны элементов, о которых мы поговорим в главе 11.

Попробуем улучшить поведение интерфейса. Как вы смогли заметить, при вводе неверного значения, чтобы инициировать ошибку, элемент должен потерять фокус. При этом фокус успешно переходит на другой элемент управления, а исходный элемент выделяется красным прямоугольником, но не печатает сообщение. Чтобы отобразить сообщение, нужно снова установить фокус на исходный элемент. Это не очень удобно, так как требует от пользователя дополнительных действий вместо того, чтобы зафиксировать фокус в элементе, где пользователь совершил ошибку ввода данных. Чтобы исправить это, воспользуемся еще одним свойством класса **Binding — NotifyOnValidationError**. Установив это свойство в значение true, система привязки данных начинает нотифицировать элемент управления о существующих исключениях, активируя событие **BindingValidationError** элемента. Определив обработчик события **BindingValidationError**, можно выполнить дополнительные действия, связанные с отображением информации об ошибке. В нашем случае, установим фокус в исходный элемент. Вот как будет выглядеть измененный XAML код:

```
<TextBlock Text="Age:" Grid.Row="3" Grid.Column="0"></TextBlock>
<TextBox Text=
   "{Binding Path=Age, Mode=TwoWay, ValidatesOnExceptions=True,
     NotifyOnValidationError=True}"
   BindingValidationError="TextBox_BindingValidationError"
   Grid.Row="3"
   Grid.Column="1"></TextBox>
```

Остается определить код обработчика события **BindingValidationError**:

```
private void TextBox_BindingValidationError(object sender,
    ValidationErrorEventArgs e)
{
    ((TextBox)sender).Focus();
}
```

ValidatesOnDataErrors

Silverlight 4 представляет еще один механизм, позволяющий проверить данные, поставляемые от цели к источнику, и сообщить о возможных ошибках. В отличие от предыдущего механизма, тут можно не вмешиваться в свойства класса, а вынести всю логику проверки и выдачи сообщений в отдельный метод, определяемый интерфейсом **IDataErrorInfo**. Это развязывает руки разработчику, позволяя создавать расширения существующих классов путем простого определения производного класса, реализующего дополнительный интерфейс.

Модифицируем класс Employee, реализовав интерфейс **IDataErrorInfo**. Приведем весь код этого класса:

```
public class Employee: INotifyPropertyChanged, IDataErrorInfo
{
    public event PropertyChangedEventHandler PropertyChanged;
```

```
    public void OnPropertyChanged(PropertyChangedEventArgs e)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, e);
    }

    private string firstName;
    private string lastName;
    private int age;
    private string email;
    private double salary;

    public double Salary
    {
        get
        {
            return salary;
        }
        set
        {
            salary = value;
            OnPropertyChanged(new
PropertyChangedEventArgs("Salary"));
        }
    }

    public string FirstName
    {
        get { return firstName; }
        set
        {
            firstName = value;
            OnPropertyChanged(new
PropertyChangedEventArgs("FirstName"));
        }
    }

    public string LastName
    {
        get { return lastName; }
        set
        {
            lastName = value;
            OnPropertyChanged(new
PropertyChangedEventArgs("LastName"));
        }
    }

    public string EMail
    {
        get { return email; }
        set
        {
            email = value;
```

```
            OnPropertyChanged(new
PropertyChangedEventArgs("EMail"));
        }
    }

    public int Age
    {
        get { return age; }
        set
        {
            age = value;
            OnPropertyChanged(new PropertyChangedEventArgs("Age"))
        }
    }

    public override string ToString()
    {
        return String.Format("{0} {1}", firstName, lastName);
    }

    string errors = null;

    public string Error
    {
        get { return errors; }
    }

    public string this[string columnName]
    {
        get
        {
            string result = null;
            if (columnName == "Age")
            {
                if (Age <= 0)
                {
                    result = "Age should be positive!";
                }
                else if (Age <= 16)
                {
                    result = "You are too young!";
                }
            }
            return result;
        }
    }
}
```

Как видите, нам пришлось убрать проверку в свойстве Age, так как реали зация индексатора, описываемого интерфейсом **IDataErrorInfo**, базируется н проверке свойства, которое уже установлено (пусть даже в неправильное зна чение).

Чтобы инициировать выдачу сообщений об ошибках, описанных в индексаторе, нужно воспользоваться свойством класса **Binding — ValidatesOnDataErrors**:

```
<TextBlock Text="Age:" Grid.Row="3" Grid.Column="0"></TextBlock>
<TextBox Text=
   "{Binding Path=Age, Mode=TwoWay, ValidatesOnDataErrors=True,
     NotifyOnValidationError=True}"
   BindingValidationError="TextBox_BindingValidationError"
   Grid.Row="3" Grid.Column="1">
</TextBox>
```

Самое плохое тут в том, что описанные выше механизмы (**ValidatesOnDataErrors** и **ValidatesOnExceptions**) нельзя комбинировать. Иными словами, если у Вас есть готовый класс, описывающий некоторый объект, при этом его свойства выполняют проверку данных и выбрасывают исключения, то **ValidatesOnDataErrors** использовать нельзя, так как он полностью игнорируется при появлении исключения (даже если **ValidatesOnExceptions** установлен в **false**).

ValidatesOnNotifyDataErrors

К сожалению, не всегда данные можно проверить сразу же на клиенте. Очень часто необходимо обратиться к источнику данных, который расположен на удаленном сервере. Лишь после этого можно сделать вывод о корректности данных. Естественно, что описанные выше механизмы не подходят для удаленной проверки данных, так как заблокируют интерфейс на время соединения с источником данных. Поэтому Silverlight 4 представляет новый механизм, позволяющий вести асинхронную проверку данных.

Чтобы обеспечить асинхронную проверку данных, класс, описывающий данные, должен реализовывать интерфейс **INotifyDataErrorInfo**. Тут описаны свойство **HasErrors**, определяющее наличие ошибок при асинхронной проверке данных, метод **GetErrors**, возвращающий объект перечислимого типа, который содержит ошибки, а также событие **ErrorsChanged**, которое инициируется при любом изменении коллекции ошибок.

Чтобы активировать работу интерфейса **INotifyDataErrorInfo**, необходимо воспользоваться свойством класса Binding — **ValidatesOnNotifyDataErrors**, установив его значение в **true**.

Особенностью этого подхода является возможность его комбинации с описанными выше подходами.

Заключение

Если Вы знакомы с механизмом привязки данных в WPF, то смогли заметить, что Silverlight 4 практически полностью приблизился к WPF, взяв все самое лучшее из механизма привязки данных. Это дает возможность разработчику использовать одинаковые подходы при построении приложений, фактически не делая разграничений.

Если говорить о механизме привязки данных в целом, то он не ограничивается описанными возможностями. Так, если Вы решите использовать дополнительные библиотеки позволяющие реализовывать LINQ запросы или запросы к данным с помощью ADO Data Services, то привязка данных эффективно работает и тут.

Между тем, возможности привязки данных часто требуют изменения кода классов, которые предназначены для хранения данных. Поэтому при построении архитектуры будущего приложения, на привязку стоит обратить особое внимание, чтобы не переписывать код в будущем.

Глава 7

ВЗАИМОДЕЙСТВИЕ С СЕРВЕРОМ

Использование WebClient

Технология Silverlight предлагает несколько способов взаимодействия с данными, начиная от простых запросов с помощью класса **WebClient** и заканчивая WCF-службами. Начнем наши исследования с простого класса **WebClient**.

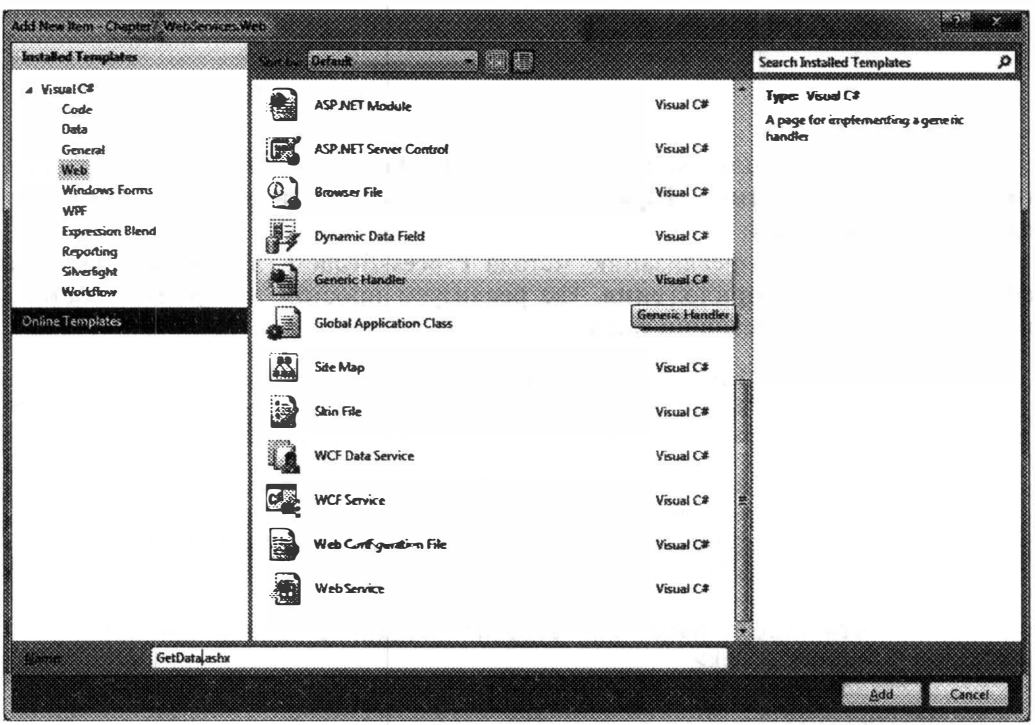


Рис. 7.1. Создание страницы на основе шаблона Generic Handler

Чтобы продемонстрировать работу класса **WebClient**, создайте новое Silverlight-приложение с тестовым Web-приложением. После чего добавьте в Web-приложение новый элемент (http обработчик) на основе шаблона **Generic Handler**.

Шаблон **Generic Handler** позволяет создать своеобразную ASP.NET страницу, не имеющую интерфейса, а предназначенную лишь для возврата набора данных. При этом в результирующий поток можно писать информацию в любом формате, будь то XML или обычный текст. Так, если Вам нужно вернуть достаточно простой набор данных, то очень часто нет необходимости заморачиваться с SOAP или WCF службами, а достаточно написать простой http обработчик.

Код созданного обработчика будет выглядеть следующим образом:

```
public class GetData : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        context.Response.ContentType = "text/plain";
        context.Response.Write("Hello World");
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

Как видно, в данном коде нет ничего сложного, достаточно записать информацию в объект типа **HttpResponse**, доступный через текущий контекст в методе **ProcessRequest**.

Не будем менять реализацию метода **ProcessRequest**, а перейдем сразу к созданию Silverlight-приложения, где реализуем простой интерфейс, состоящий из кнопки и текстового поля:

```
<UserControl x:Class="Chapter7_WebServices.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <StackPanel x:Name="LayoutRoot" Background="White"
        Orientation="Horizontal" VerticalAlignment="Top">
        <Button Content="Get Data" Width="100" Height="30"
           Click="Button_Click">
        </Button>
        <TextBlock Name="serverData" Margin="5" FontSize="16">
        </TextBlock>
    </StackPanel>
</UserControl>
```

Реализуем код, который при нажатии на кнопку обращается к http обработчику и извлекает данные в текстовое поле. Вот как будет выглядеть этот код:

```
public partial class MainPage : UserControl
{
    public MainPage()
```

```
    {
        InitializeComponent();
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        WebClient web = new WebClient();

        web.DownloadStringCompleted += web_DownloadStringCompleted;

        web.DownloadStringAsync(
           new Uri("../GetData.ashx", UriKind.Relative));
    }

    void web_DownloadStringCompleted(object sender,
       DownloadStringCompletedEventArgs e)
    {
        serverData.Text = e.Result;
    }
}
```

Чтобы обратиться к нашему обработчику, необходимо создать объект типа **WebClient** и установить ссылку на метод, реагирующий на завершение чтения данных с сервера. Особенность этого метода состоит в том, что он вызывается в интерфейсном потоке, то есть, чтобы обновить интерфейс, нам не нужно выполнять никаких дополнительных действий. В завершение всего необходимо вызвать метод **DownloadStringAsync**, который инициирует обращение к серверу в асинхронном потоке. В качестве параметров он получает адрес нашего http-обработчика. Нужно помнить, что обработчик находится на уровень выше, чем наше Silverlight-приложение, поэтому ссылку нужно формировать соответствующим образом.

Результат работы приложения показан ниже:

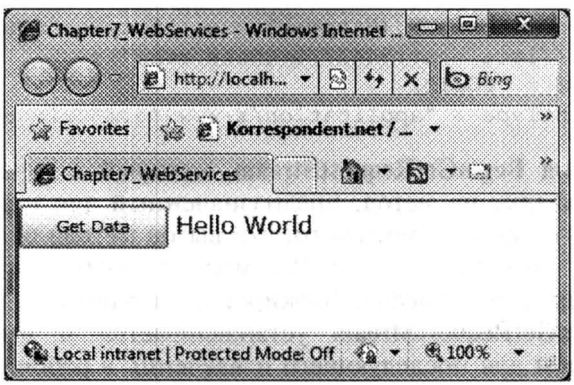


Рис. 7.2. Результат работы приложения

Аналогичным образом можно сформировать и запрос к службе, отправив его с помощью метода **UploadStringAsync**.

Если Ваши данные представлены в бинарном формате или Вам просто нужен поток (**Stream**), то тут используется другой набор методов: **OpenReadAsync**

и **OpenWriteAsync**. Эти методы работают с потоками. Вспомните код, позволяющий загрузить сборку по требованию:

```
WebClient wc = new WebClient();
wc.OpenReadCompleted +=
    new OpenReadCompletedEventHandler(wc_OpenReadCompleted);
wc.OpenReadAsync(
    new Uri("Chapter3_SilverlightControl.dll",
       UriKind.Relative));
```

Использование HttpWebRequest и HttpWebResponse

Следующие два класса, которые позволяют реализовать доступ к Web-службам, — это **HttpWebRequest** и **HttpWebResponse**. Несмотря на сложность использования, эти два класса в некоторых задачах приходятся как нельзя кстати, поскольку являются универсальными и позволяют реализовывать запросы любой сложности. При этом и формирование данных, и обработка данных происходит асинхронно.

Перепишем наш пример выше таким образом, чтобы использовать вместо **WebClient** класс **HttpWebRequest**. Чтобы реализовать код, нам потребуется выполнить следующие действия:

- Создать объект типа **HttpWebRequest**, установив основные свойства. Создать объект этого типа нельзя с помощью открытого конструктора. Для этого нужно вызвать статический метод **Create** класса **WebRequest** и выполнить преобразование созданного объекта к **HttpWebRequest**. Чтобы избежать преобразований, можно использовать метод **CreateHttp**, который сразу возвращает объект типа **HttpWebRequest**. В качестве параметров необходимо передать полную ссылку на службу или страницу, к которой формируется запрос.

```
HttpWebRequest web = (HttpWebRequest)HttpWebRequest.Create(
   "http://localhost:54427/GetData.ashx");
web.Method = "POST";
web.ContentType = "application/x-www-form-urlencoded";
```

- Вызвать метод **BeginGetRequestStream**, который в качестве параметров принимает ссылку на метод, предназначенный для подготовки запроса к серверу. Тут нужно отметить то, что вызов метода будет инициирован асинхронно. Это значит, что Вы можете подготовить сколь угодно сложный запрос, не опасаясь блокировки интерфейса. Второй параметр метода **BeginGetRequestStream** устанавливается в имени созданного **HttpWebRequest** для последующего извлечения в рабочем потоке.
- Реализовать метод, подготавливающий запрос к отправке. Тут необходимо завершить формирование запроса с помощью метода **EndGetRequestStream**, возвращающий экземпляр **Stream**, который будет отослан на сервер. На этом этапе Вы можете дописать в поток любую информацию, чтобы окончательно сформировать запрос. Это можно сделать с помощью класса **StreamWriter**. В завершении работы метода необходимо

инициировать вызов метода **BeginGetResponse**, передав в качестве параметров ссылку на метод, который будет вызван для обработки ответа с сервера. Таким образом, запрос сформирован и отправлен. Остается дождаться и обработать ответ.

```
StreamWriter postDataWriter =
   new StreamWriter(request.EndGetRequestStream(e));

//записать свои данные

postDataWriter.Close();

request.BeginGetResponse(
   new AsyncCallback(ResponsePrepare), request);
```

- Реализовать метод, принимающий поток данных с сервера. Тут нужно отметить, что этот метод также вызывается асинхронно, то есть данные, которые поступили с сервера, нельзя тут же вносить в интерфейс. Чтобы обновить интерфейс, необходимо обратиться к классу Dispatcher, инициировав вызов метода в интерфейсном потоке. Но данные следует обработать в текущем методе, чтобы не блокировать потом интерфейсный поток.
- Определить метод, обновляющий интерфейс.

Вот такая заумная, хотя и достаточно универсальная процедура. Несмотря на то, что представленные классы Вы будете использовать редко (Visual Studio позволяет генерировать прокси классы, делая всю черновую работу), попробуйте реализовать хотя бы один пример самостоятельно. После освоения механизма взаимодействия с серверными ресурсами с помощью HttpWebRequest и HttpWebResponse, любые службы будут Вам не страшны.

Приведем полный код нашего приложения, реализованный по схеме выше:

```
string responseString;

private void Button_Click(object sender, RoutedEventArgs e)
{
    HttpWebRequest web = (HttpWebRequest)HttpWebRequest.Create(
       "http://localhost:54427/GetData.ashx");
    web.Method = "POST";
    web.ContentType = "application/x-www-form-urlencoded";

    web.BeginGetRequestStream(RequestPrepare, web);
}

private void RequestPrepare(IAsyncResult e)
{
    HttpWebRequest request = (HttpWebRequest)e.AsyncState;
    StreamWriter postDataWriter =
       new StreamWriter(request.EndGetRequestStream(e));

    //записать свои данные

    postDataWriter.Close();

    request.BeginGetResponse(
```

```
        new AsyncCallback(ResponsePrepare), request);
}

private void ResponsePrepare(IAsyncResult e)
{
    HttpWebRequest request = (HttpWebRequest)e.AsyncState;
    HttpWebResponse response =
       (HttpWebResponse)request.EndGetResponse(e);

    StreamReader responseReader =
       new StreamReader(response.GetResponseStream());

    responseString = responseReader.ReadToEnd();

    Dispatcher.BeginInvoke(FillInterface);
}

private void FillInterface()
{
    serverData.Text=responseString;
}
```

Как видно, код значительно более объемный, чем в предыдущем примере.

Использование прокси-классов для взаимодействия со службами

Взаимодействуя со страницами на Web-сервере, перед разработчиком не остается выбора, как использовать классы, описанные выше. Но, если речь идет о Web-службах, построенных на основе передачи данных по SOAP протоколу, или если речь идет о WCF службах, то использовать **HttpWebRequest** и **HttpWebResponse** не очень удобно. Ведь взаимодействие с SOAP службами осуществляется с помощью XML. Последнее означает, что нужно не только правильно сформировать запрос в виде XML документа, но и потратить много усилий на разбор ответа. Благо Visual Studio делает всю черновую работу за нас, предоставляя в пользование лишь прокси класс, который содержит методы с прототипами, аналогичными внутри самой Web-службы.

Чтобы продемонстрировать работу с Visual Studio, добавим новый элемент к проекту Web-сайта — **Web Service** (рис. 7.3).

Код созданной службы будет выглядеть следующим образом:

```
[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[System.ComponentModel.ToolboxItem(false)]
public class WebService1 : System.Web.Services.WebService
{

    [WebMethod]
    public string HelloWorld()
    {
        return "Hello World";
```

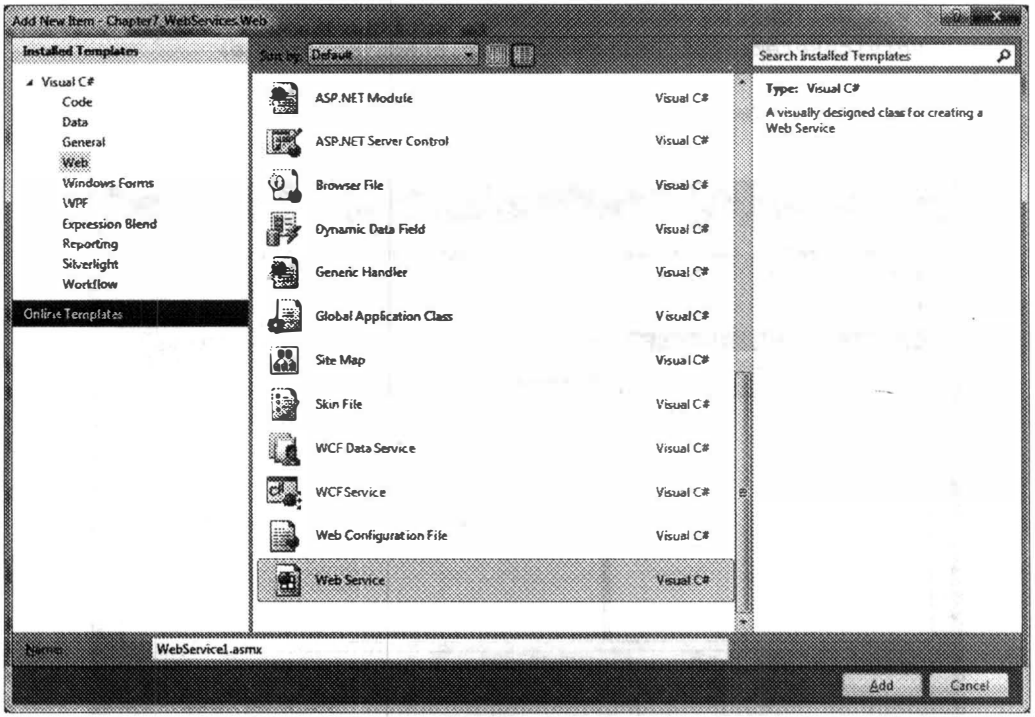


Рис. 7.3. Создание Web-службы

Как видно, даже для реализации Web-службы нет необходимости реализовывать создание XML вручную. Достаточно пометить класс, содержащий Web-службы, атрибутом **WebService**, а методы, которые мы хотим сделать доступными снаружи, атрибутами **WebMethod**.

Чтобы сгенерировать прокси класс, достаточно инициировать вызов контекстного меню для Silverlight-проекта и вызвать команду **Add Service Reference...**

В появившемся окне следует указать адрес нашей Web-службы. Благодаря описанию службы WSDL (генерируется автоматически), Visual Studio создает промежуточные классы, реализующие взаимодействие со службой, и добавляет их в Silverlight-проект.

Используя промежуточный класс, можно реализовать код по взаимодействию со службой следующим образом:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    ServiceReference1.WebService1SoapClient web =
       new ServiceReference1.WebService1SoapClient();

    web.HelloWorldCompleted += web_HelloWorldCompleted;

    web.HelloWorldAsync();
```

```
void web_HelloWorldCompleted(object sender,
    ServiceReference1.HelloWorldCompletedEventArgs e)
{
    serverData.Text = e.Result;
}
```

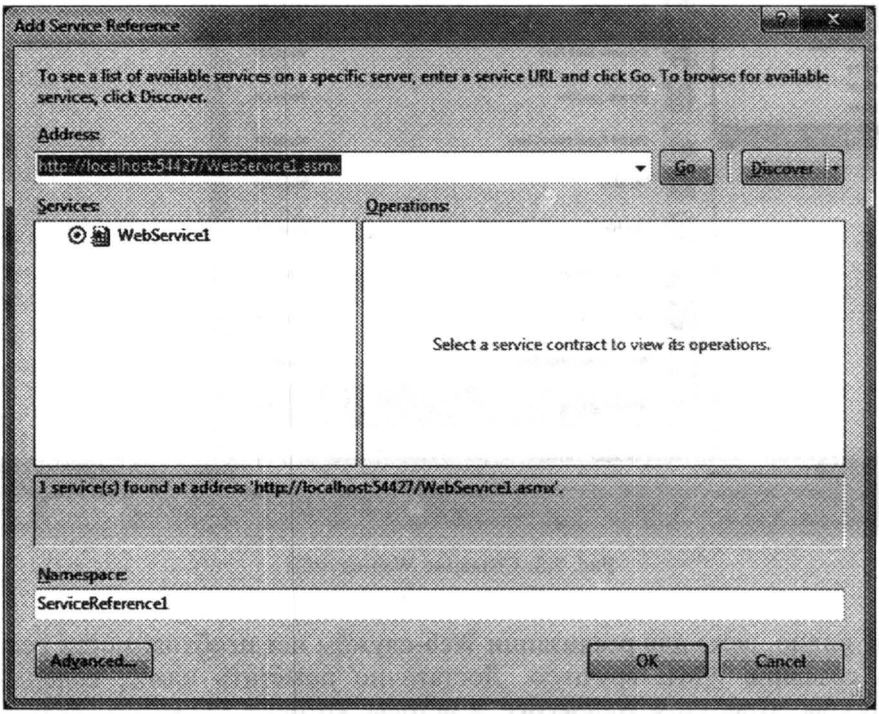


Рис. 7.4. Создание прокси класса

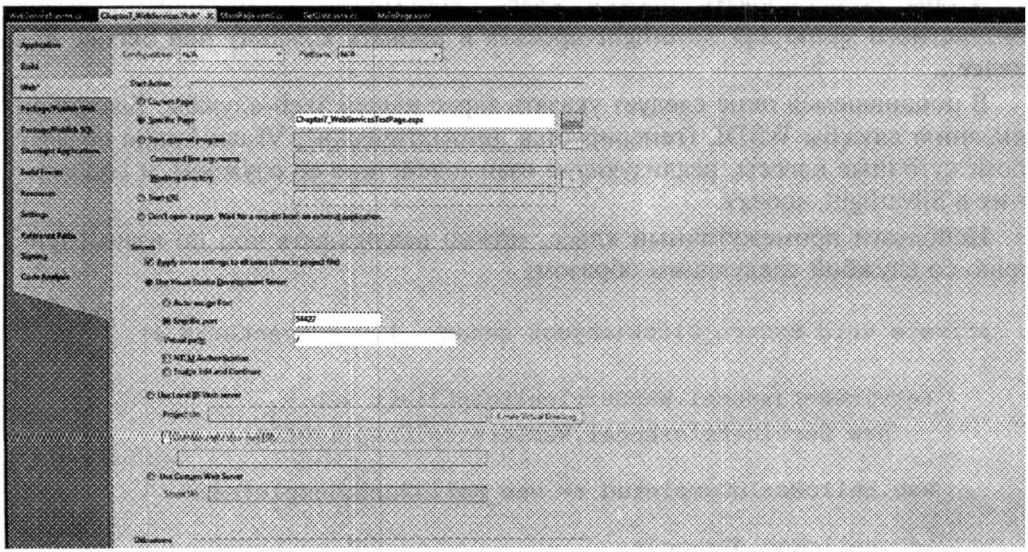


Рис. 7.5. Назначение порта приложению

Как видите, код укладывается в несколько строк, хотя за сценой происходит обработка XML и сложное формирование запроса.

При реализации данного примера Вам может понадобиться запускать Web-приложение, используя конкретный порт. Чтобы задать порт для Web-сервера, идущего в комплекте с Visual Studio, перейдите к свойствам Web-приложения и укажите явный номер порта на вкладке Web.

Доступ к службам в других доменах

Silverlight 4, как и любой другой компонент, запущенный внутри браузера, не позволяет реализовывать запросы к службам в других доменах. Это ограничение наложено стандартами безопасности для любого клиентского кода. Все дело в том, что разрешение запросов к службам в других доменах, например из JavaScript кода, открывает потенциальную угрозу, которая выражается в возможности организации DoS-атаки (отказ в обслуживании). Ведь современные механизмы обеспечения безопасности (программные и аппаратные firewalls) легко способны заблокировать DoS-атаку с какого-то IP адреса. Но представьте, что злоумышленник внедрил JavaScript код, позволяющий атаковать определенный ресурс, в целый набор популярных сайтов. В этом случае DoS-атака становится распределенной (каждый клиент одного из сайтов начинает генерировать запросы к службе). Такую атаку заблокировать сложнее.

Но не все так печально, как описано выше. Если Вы все-таки хотите разрешить Silverlight-приложению доступ к службе другого домена, который администрируете также Вы, то тут есть два механизма:

- Разместить в корневой директории Web-приложения специальный файл **clientaccesspolicy.xml**, который позволяет дать доступ к службам приложениям из указанного домена;
- Разместить в корневой директории Web-приложения файл **crossdomain.xml**, который должен открывать доступ к домену всем внешним клиентским приложениям;

Пример файла **clientaccesspolicy.xml** находится ниже:

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="SOAPAction ">
        <domain uri="http://contoso.com"/>
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true"/>
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

Тут открывается доступ всем приложениям, загруженным из домена contoso.com.

Второй способ менее эффективен, если говорить о Silverlight, так как делает домен полностью открытым. Тем не менее, приведем пример и файла **crossdomain.xml**:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM
   "http://www.macromedia.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-http-request-headers-from domain="*"
     headers="SOAPAction,Content-Type"/>
</cross-domain-policy>
```

Заключение

Silverlight 4 предоставляет достаточно много механизмов, позволяющих взаимодействовать практически с любыми типами служб. Так, если Silverlight-приложение обращается к службе, возвращающей простые или однотипные данные, то можно воспользоваться классом **WebClient**. Если служба имеет более сложную структуру и возвращает сложные наборы данных, требующих дополнительной обработки на клиенте, то лучше воспользоваться такими классами, как **HttpWebResponse** и **HttpWebRequest**. Кроме этого, Visual Studio позволяет создавать прокси классы, скрывающие от Вас всю черновую работу по взаимодействию с SOAP и WCF службами.

Также нужно помнить, что в целях безопасности все запросы от Silverlight-приложений должны быть направлены только к домену приложения. Хотя удаленный сервер имеет возможность разрешить доступ к службам для отдельных Silverlight-приложений.

Напоследок хочу отметить, что в SDK содержаться дополнительные библиотеки, которые позволяют более эффективно работать с XML-данными RSS, а также JSON.

Глава 8

ГРАФИКА, ТРАНСФОРМАЦИЯ И АНИМАЦИЯ

Графические примитивы

Изучая такие технологии как Win Forms, ASP.NET и др., я бы никогда не останавливался на рисовании прямоугольников, линий и эллипсов. Но в Silverlight-приложениях графические примитивы применяются настолько часто, что игнорировать их глупо. Так, в главе 11 мы познакомимся с возможностью менять внешний вид стандартных элементов управления. При этом, если взять обычную кнопку, то мы обнаружим, что она состоит из целого набора прямоугольников, которые имеют различную заливку.

Начнем наш обзор графики в Silverlight с представления нескольких стандартных элементов: **Rectangle**, **Ellipse**, **Line**, **Polygon**, **Polyline**, **Path**. Все перечисленные графические примитивы являются наследником класса **Shape**, который является прямым наследником **UIElement**. С одной стороны это означает, что графические примитивы могут быть размещены в любом из контейнеров, а с другой — графические примитивы не имеют большинства свойств, которые характерны элементам управления. Несмотря на это, класс Shape предоставляет несколько общих свойств, которые могут быть использованы всеми примитивами. Рассмотрим эти свойства:

- **Stroke** — это свойство задает кисть, которой отрисовывается обводка фигуры;
- **StrokeThickness** — задает ширину обводки;
- **Fill** — это свойство позволяет установить кисть заливки области, которая находится внутри контура фигуры;

Если говорить про контур фигуры, то тут возможны дополнительные свойства, задающие обводку пунктиром или закругление краев обводки:

- **StrokeDashArray** — это свойство позволяет задать массив типа double, который определяет шаблон для обводки фигуры пунктиром. Значения в массиве определяют длину линий и пропусков в обводке примитива;
- **StrokeDashCap** — определяет форму наконечников (углов) обводки (пунктиров). Может принимать одно из четырех значений: **Float**, **Round**, **Squared**, **Triangle**;
- **StrokeDashOffset** — смещение обводки пунктиром;

- **StrokeEndLineCap** — определяет вид наконечника каждого из пунктиров (конец пунктира). Может принимать значения: **Float**, **Round**, **Squared**, **Triangle**;
- **StrokeStartLineCap** — определяет вид наконечника каждого из пунктиров (начало пунктира). Может принимать значения: **Float**, **Round**, **Squared**, **Triangle**;
- **StrokeLineJoin** — определяет тип соединения линий в случае пересечения. Может принимать следующие значения: **Bevel**, **Miter**, **Round**;

Продемонстрируем использование этих свойств на первом графическом примитиве **Rectangle**:

```
<UserControl x:Class="Chapter8_Shape.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <StackPanel x:Name="LayoutRoot" Background="White">

        <Rectangle Width="300" Height="100" Stroke="Black"
           StrokeThickness="10"
           RadiusX="25" RadiusY="25" Margin="5"></Rectangle>

        <Rectangle Width="300" Height="100" Stroke="Black"
           StrokeThickness="10" RadiusX="25" RadiusY="25"
           StrokeDashArray="5,2,3,2" Margin="5"></Rectangle>

        <Rectangle Width="300" Height="100" Stroke="Black"
           StrokeThickness="10" RadiusX="25" RadiusY="25"
           StrokeDashArray="5,2,3,2" StrokeDashCap="Triangle"
           Margin="5"></Rectangle>
    </StackPanel>
</UserControl>
```

В результате работы этого кода на экране появятся следующие «прямоугольники» (рис. 8.1).

Как видно из кода, чтобы создать прямоугольник, достаточно задать свойства **Width** и **Height**, которые определяют длину и ширину прямоугольника. Дополнительные свойства **RadiusX** и **RadiusY** задают углы изгиба вершин прямоугольника. Первый прямоугольник из нашего примера имеет сплошную обводку. Мы установили ширину обводки равную 10, чтобы пример выглядел более наглядно. Второй и третий прямоугольники подключают дополнительное свойство **StrokeDashArray**, задающее длину штрихов в 5 и 3 единиц, а также величину пробела в 2 единицы. Третий прямоугольник задает прорисовку окончаний штрихов в виде треугольников.

Следующий графический примитив Ellipse позволяет нарисовать эллипс или круг. Тут основные свойства Width и Height, которые определяют размер горизонтального и вертикального диаметра эллипса:

```
<Ellipse Width="300" Height="100" StrokeThickness="10"
   Stroke="Black" Fill="Red">
</Ellipse>
```

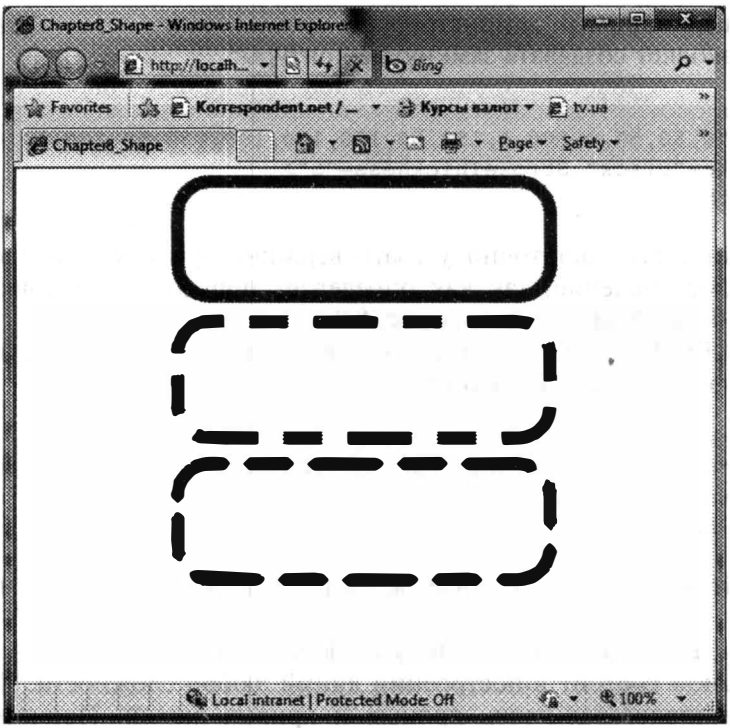


Рис. 8.1. Результат работы приложения

В данном примере мы использовали еще одно свойство **Fill**, которое позволяет выполнить заливку контура. В данном случае мы используем специальную кисть **SolidColorBrush**, о которой речь пойдет позже. Между тем для свойства **Fill** существует конвертор, который преобразует имя цвета в кисть. Результат работы примера показан ниже.

Рис. 8.2. Результат работы приложения

Следующий графический примитив — это **Line**, позволяющий отобразить прямую линию. Этому элементу достаточно принимать координаты начала и окончания линии:

```
<Line X1="0" Y1="0" X2="100" Y2="100" Stroke="Black"></Line>
```

Вопрос, который возникает: относительно чего задаются координаты? Координаты задаются относительно контейнера для графического примитива. Если контейнером является **Canvas**, то проблем с прямой нет, а вот если в качестве контейнера использовать другой элемент, то прямая будет сохранять длину и наклон, но не будет иметь привязки к конкретным координатам.

Следующий пример демонстрирует использование примитива Polygon, который позволяет создавать замкнутые контуры:

```
<Polygon
Points="0,50,50,0,100,0,150,50,150,100,100,150,50,150,0,100"
   Stroke="Black" StrokeThickness="5">
</Polygon>
```

Как видно, тут достаточно указать вершины фигуры. Порядок вершин имеет большое значение, так как определяет порядок прорисовки контура. Результат работы кода показан на рис. 8.3.

Элемент **Polyline** аналогичен предыдущему примитиву, но рисует не замкнутый контур. Вот аналогичный код:

```
<Polyline
Points="0,50,50,0,100,0,150,50,150,100,100,150,50,150,0,100"
   Stroke="Black" StrokeThickness="5">
</Polyline>
```

Результат работы этого примера будет выглядеть немного по-другому (рис. 8.4).

Наконец, перейдем к последнему графическому примитиву — **Path**. Задача этого объекта состоит в построении линий любой сложности. Фактически он строит последовательность прямых и кривых линий. Несмотря на всю сложность объекта, у него только одно основное свойство — это **Data**. Данное свойство может содержать либо геометрические объекты, о которых речь пойдет дальше, либо набор данных, состоящих из специальных команд. Рассмотрим небольшой пример.

```
<Path Stroke="Black" StrokeThickness="5"
   Data="M 10,100 C 10,300 300,-200 300,100">
</Path>
```

Результатом работы этого кода будет кривая, которая отображена на рис. 8.5.

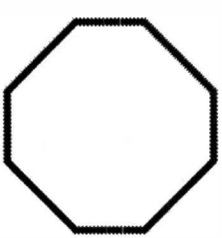


Рис. 8.3. Использование Polygon

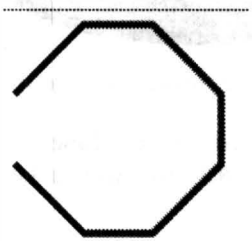


Рис. 8.4. Использование Polyline

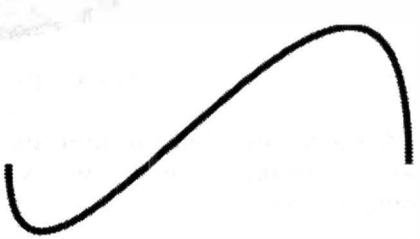


Рис. 8.5. Использование Path

Как видно, команды в свойстве **Data** записывается имя команды, после которой следует набор значений, разделенных запятыми и отделяемый от команды пробелом. Затем через пробел следует другая команда и т. д.

Существует несколько команд, которые можно запомнить и использовать в элементе **Path**:

- **M** — позволяет перейти в заданную точку;
- **L** — рисует линию от текущей точки в заданную;
- **H** — рисует горизонтальную линию заданной длины от текущей точки;
- **V** — рисует вертикальную линию заданной длины от текущей точки;
- **C** — принимает в качестве параметров три точки и позволяет построить кубическую кривую Безье, используя текущую точку и третью точку. Оставшиеся две точки используются как вспомогательные;
- **Q** — принимает в качестве параметров две точки и позволяет построить квадрическую кривую Безье, используя текущую и вторую точки;
- **S** — принимает в качестве параметров две точки и позволяет нарисовать сглаженную кубическую кривую Безье;
- **T** — позволяет построить сглаженную квадрическую кривую Безье;
- **A** — позволяет построить эллиптическую дугу. Принимает пять параметров: размер, угол поворота, один из двух сегментов (0 или 1), направление отсчета угла (0 или 1), конечную точку;
- **Z** — завершает текущий контур, проводя прямую линию между начальной точкой и текущей.

Вот мы и рассмотрели все основные примитивы. Перейдем теперь к кистям, с помощью которых можно определять заливку, как содержимого, так и контура.

Кисти

Кисти представляют собой специальные объекты, которые определяют, каким образом закрашивать контур и содержимое других объектов. Рассмотрим все кисти по порядку.

SolidColorBrush

Выше мы использовали для закраски контура определение цветов (Red, Green и т. д.). На самом деле все используемые нам значения всегда преобразовывались в экземпляр класса **SolidColorBrush**. Это специальный вид кисти, который заполняет контур или внутреннюю часть примитива одним цветом. Фактически, два прямоугольника ниже будут закрашены одним цветом:

```
<Rectangle Width="300" Height="100" Fill="Red"></Rectangle>

<Rectangle Width="300" Height="100">
    <Rectangle.Fill>
        <SolidColorBrush Color="Red"></SolidColorBrush>
    </Rectangle.Fill>
</Rectangle>
```

Естественно, что подобную кисть можно установить и в коде на C#:

```
rect.Fill = new SolidColorBrush(Colors.Red);
```

Поддержка системных цветов

Выше, при создании кистей мы использовали в основном цвета, определенные в типе **Colors**. Однако Silverlight 3 (и 4 соответственно) поддерживает статический класс **SystemColors** с набором цветов, зависимых от локальных настроек пользователя. Это такие цвета как **Menu**, **ActiveCaption** **InactiveWindow** и др. Использование этих цветов возможно применительно к любому из UI элементов:

```
. . . = new SolidColorBrush(SystemColors.ControlDarkColor);
```

LinearGradientBrush

Кисть **LinearGradientBrush** позволяет закрасить области или контур с использованием линейного градиента. По умолчанию линейный градиент рассчитывается от верхнего левого угла к нижнему правому, и позволяет выполнить плавный переход от одного цвета к другому:

```
<Rectangle Width="300" Height="300">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <GradientStop Color="Red" Offset="0"></GradientStop>
            <GradientStop Color="Green" Offset="1"></GradientStop>
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

Этот код генерирует следующий прямоугольник (рис. 8.6).

Направление градиента легко изменить, если задать такие свойства, как **StartPoint** и **EndPoint**. При этом нужно помнить, что градиент привязан к нормированному прямоугольнику со сторонами равными 1.

```
<Rectangle Name="rect" Width="300" Height="300">
    <Rectangle.Fill>
        <LinearGradientBrush StartPoint="0,1" EndPoint="1,0">
            <GradientStop Color="Red" Offset="0"></GradientStop>
```

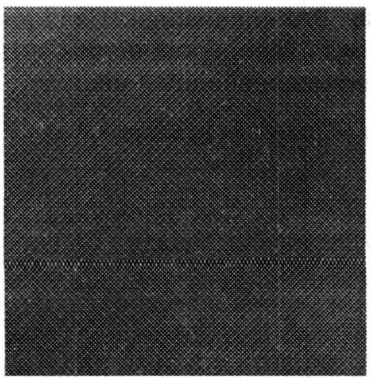


Рис. 8.6. Использование LinearGradient

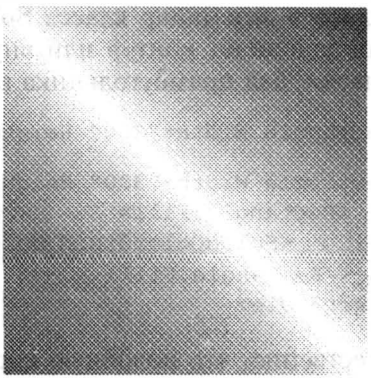


Рис. 8.7. Добавление параметров к LinearGradient

```
            <GradientStop Color="White"
Offset="0.5"></GradientStop>
            <GradientStop Color="Green" Offset="1"></GradientStop>
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

Код выше не только изменил направление градиента, но и установил большое количество интервалов. Таким образом, до диагонали прямоугольника цвет будет меняться с красного на белый, а после диагонали с белого на зеленый. Вот результаты работы приложения рис. 8.7).

RadialGradientBrush

Кисть **RadialGradientBrush** очень похожа на предыдущую кисть, только цвет распространяется в направлении радиуса окружности. **RadialGradientBrush** также привязана к прямоугольнику размерами 1 на 1, но точка отчета для распространения градиента установлена по умолчанию в 0.5, что позволит по умолчанию рисовать градиент из центра фигуры:

```
<Rectangle Width="300" Height="300">
    <Rectangle.Fill>
        <RadialGradientBrush>
            <GradientStop Color="Red" Offset="0"></GradientStop>
            <GradientStop Color="White"
Offset="0.5"></GradientStop>
            <GradientStop Color="Green" Offset="1"></GradientStop>
        </RadialGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

В результате прямоугольник будет выглядеть следующим образом:

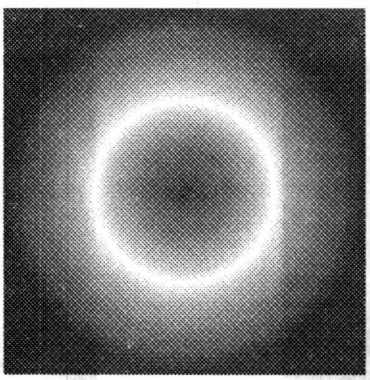


Рис. 8.8. Использование RadialGradientBrush

Если Вы хотите сменить фокус градиента, то просто установите свойство GradientOrigin в пределах прямоугольника со сторонами 1:

```
<Rectangle Name="rect" Width="300" Height="300">
```

```
    <Rectangle.Fill>
        <RadialGradientBrush GradientOrigin="0,0">
            <GradientStop Color="Red" Offset="0"></GradientStop>
            <GradientStop Color="White"
Offset="0.5"></GradientStop>
            <GradientStop Color="Green" Offset="1"></GradientStop>
        </RadialGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

В результате фокус градиента сместится в верхнюю левую вершину прямоугольника (рис. 8.9).

Помимо уже рассмотренных свойств, существуют еще такие как **SpreadMethod**, **RadiusX** и **RadiusY**. Первое свойство определяет характер распространения градиента при подходе к границе нормированного прямоугольника, а вторые два свойства позволяют изменить радиус окружности, по которой распространяется градиент (фактически уменьшить стороны прямоугольника, в который вписана окружность). Ниже показан код, который создает заливку из трех цветов с многократным повторением, а чтобы пример выглядел эффектно, тут также устанавливается меньший радиус для окружности:

```
<Rectangle Name="rect" Width="300" Height="300">
    <Rectangle.Fill>
        <RadialGradientBrush SpreadMethod="Repeat"
            RadiusX="0.2" RadiusY="0.2">
            <GradientStop Color="Red" Offset="0"></GradientStop>
            <GradientStop Color="White"
Offset="0.5"></GradientStop>
            <GradientStop Color="Green" Offset="1"></GradientStop>
        </RadialGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

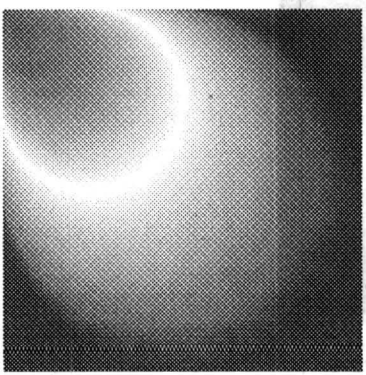


Рис. 8.9. Градиент со смещенным фокусом

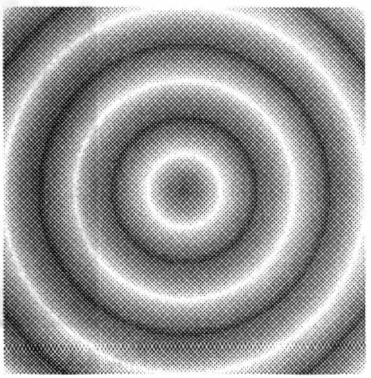


Рис. 8.10. Градиент с измененным радиусом

Результат работы этого кода можно увидеть на рис. 8.10.

ImageBrush и VideoBrush

Наряду с заливками с помощью цвета, можно использовать изображения и видео. Для этого определены две специальные кисти **ImageBrush** и **VideoBrush**.

Кисть **ImageBrush** имеет два основных свойства: **ImageSource** и **Stretch**. Первое свойство задает ссылку на изображение, а второе определяет, как изображение будет заполнять область (по умолчанию заполняет всю область).

Вот пример кода, использующего **ImageBrush** в качестве заливки:

```
<TextBlock Text="Hello" FontSize="170" FontWeight="Bold">
    <TextBlock.Foreground>
        <ImageBrush ImageSource="Tulips.jpg"></ImageBrush>
    </TextBlock.Foreground>
</TextBlock>
```

Результат работы этого кода можно увидеть на следующем рисунке:

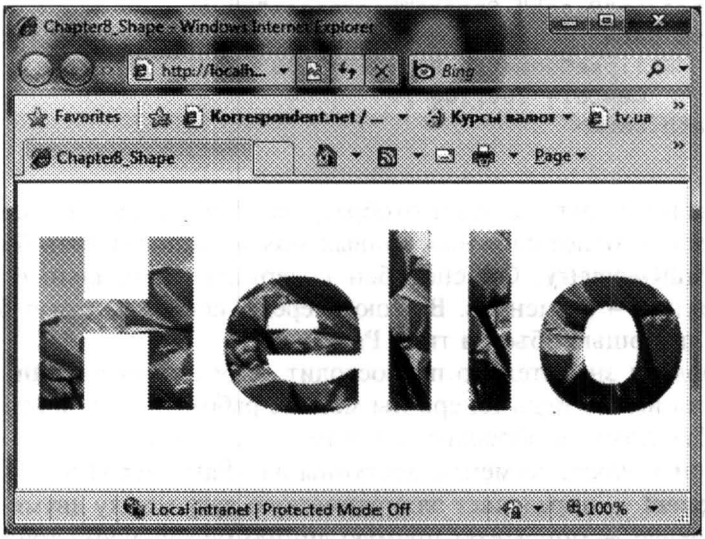


Рис. 8.11. Использование ImageBrush кисти

Аналогично можно использовать и VideoBrush. Правда в качестве источника эта кисть должна получить ссылку на MediaElement.

```
<MediaElement Source="WildLife.wmv" Name="myMedia"
    Visibility="Collapsed"></MediaElement>
<TextBlock Text="Hello" FontSize="170" FontWeight="Bold">
    <TextBlock.Foreground>
        <VideoBrush SourceName="myMedia"></VideoBrush>
    </TextBlock.Foreground>
</TextBlock>
```

В этом примере мы создали медиа элемент, а затем установили свойство **SourceName** в имя созданного элемента. Сам медиа элемент мы сделали невидимым, чтобы не отображать видео полностью.

Использование геометрических объектов

В Silverlight существует целая группа геометрических объектов. Эти объекты описывают всевозможные геометрические фигуры, но в отличие от рассмотренных примитивов, они могут быть использованы только для создания объекта **Path** или для установки свойства **Clip** всем **UIElement** объектам.

Рассматривая геометрические объекты, можно выделить группу простых объектов, а также группу сегментов, которые являются частью составного контура. Рассмотрим группу простых объектов:

- **LineGeometry** — задает линию. Тут можно указать начальную и конечную точку;
- **EllipseGeometry** — задает эллипс. Может принимать радиус по каждой из осей, а также центр эллипса;
- **RectangleGeometry** — задает прямоугольник.

Вот небольшой пример использования простой геометрической фигуры:

```
<Path Stroke="Black" StrokeThickness="2" >
    <Path.Data>
        <EllipseGeometry RadiusX="100" RadiusY="100"
           Center="100,100"/>
    </Path.Data>
</Path>
```

В результате на экране будет отображена обычная окружность.

Для создания более сложных кривых можно использовать более сложный объект — **PathGeometry**. Он способен содержать специальный набор более мелких объектов — сегментов. В свою очередь, сегменты могут быть разбиты на группы с помощью объекта типа **PathFigure**.

Такой подход значительно превосходит язык элемента **Path**, который мы рассматривали выше. Ведь теперь Вы можете работать с сегментами, как с отдельными объектами, и обращаться к ним по имени из кода.

Рассмотрим, какие сегменты доступны для **PathGeometry**:

- **ArcSegment** — определяет эллиптическую дугу между двумя точками;
- **LineSegment** — описывает прямую линию между двумя точками;
- **BezierSegment** — описывает кривую Безье;
- **PolyLineSegment** — позволяет нарисовать ряд прямых линий по массиву точек;
- **PolyBezierSegment** — позволяет задать массив точек, которые используются для построения набора кривых Безье;
- **QuadraticBezierSegment** — квадратическая кривая Безье;
- **PolyQuadraticBezierSegment** — позволяет построить набор квадратических кривых Безье.

Вот небольшой пример, использующий **PathGeometry**:

```
<Path Stroke="Black" StrokeThickness="1" >
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigure StartPoint="10,50">
```

```
            <PathFigure.Segments>
              <BezierSegment Point1="100,0"
                Point2="200,200"
                Point3="300,100"/>
              <LineSegment Point="400,100" />
              <ArcSegment Size="50,50" RotationAngle="45"
                IsLargeArc="True" SweepDirection="Clockwise"
                Point="200,100"/>
            </PathFigure.Segments>
          </PathFigure>
        </PathGeometry.Figures>
      </PathGeometry>
    </Path.Data>
  </Path>
```

Результат работы этого примера выглядит так:

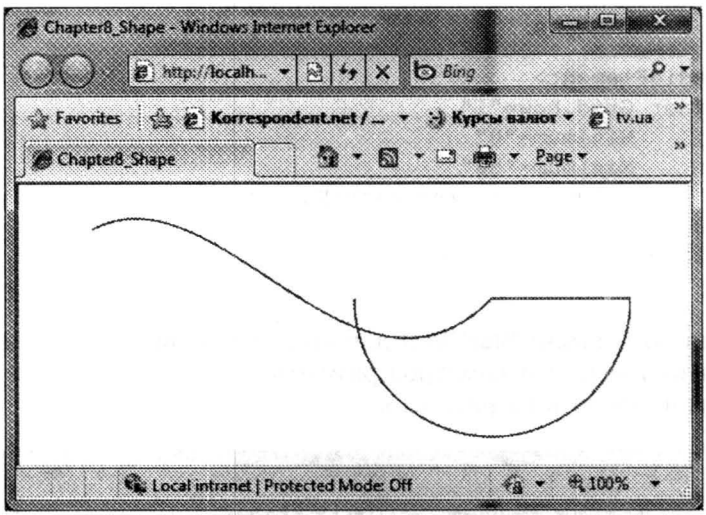


Рис. 8.12. Использование PathGeometry

В завершении отметим, что если вы хотите объединять несколько геометрических объектов, то можете воспользоваться элементов **GeometryGroup**.

Работа с изображениями

После рассмотрения графических примитивов и кистей остановимся на двух интересных темах, раскрывающих возможности работы с изображениями: использование эффектов и взаимодействие с изображением, как с набором точек.

Работа с эффектами

Начиная с Silverlight 3, у разработчика появилась возможность создавать собственные эффекты. Под эффектом мы будем понимать возможность взаи-

модействовать с частью интерфейса как с изображением, применяя к точкам этого изображения некоторые правила. В зависимости от настройки правила, можно достигнуть того или другого эффекта, начиная от смещения точек по спирали (создавая эффект закручивания) и заканчивая дублированием части точек для создания тени с отражением.

Итак, начнем с двух встроенных эффектов — это **BlurEffect** и **DropShadowEffect**. Эти эффекты позволяют размыть заданную область и отобразить тень соответственно. Рассмотрим простой пример:

```
<Grid x:Name="LayoutRoot" Background="White">
    <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <MediaElement Source="7.wmv">
        <MediaElement.Effect>
            <BlurEffect x:Name="blurEf" Radius="0"></BlurEffect>
        </MediaElement.Effect>
    </MediaElement>
    <Slider Grid.Row="1"
            Minimum="0"
            Maximum="50"
            Value="{Binding Path=Radius, ElementName=blurEf,
Mode=TwoWay}">
    </Slider>
</Grid>
```

Тут мы использовали **BlurEffect** для «размытия» видеоизображения, а ползунок — для изменения параметров размытия. Результат работы нашего приложения можно увидеть на рис. 8.13.

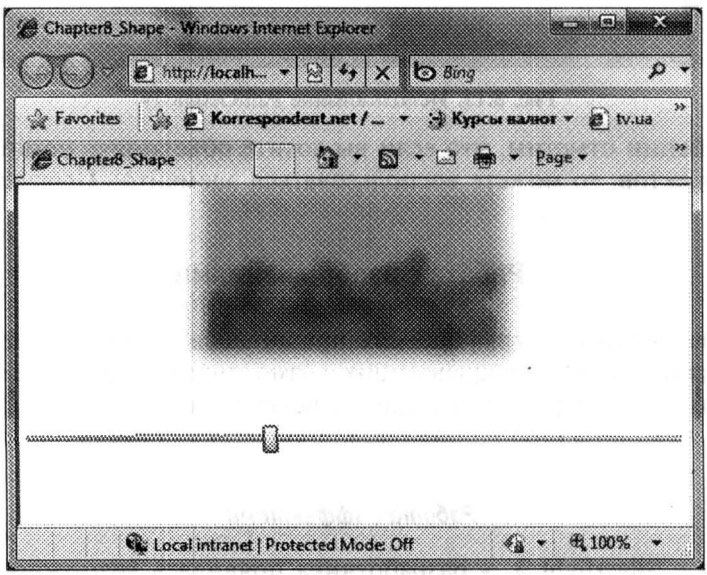


Рис. 8.13. Использование BlurEffect

Обратите внимание на то, что элемент **Effect** может содержать лишь один эффект и не позволяет работать с коллекцией элементов. В то же время эффекты можно применять к любым элементам в Silverlight. Поэтому, если нужно применить несколько эффектов, то просто используйте контейнеры. Следующий пример демонстрирует, как для нашего видео добавить эффект тени:

```
<Grid x:Name="LayoutRoot" Background="White">
    <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <Border Margin="25">
        <Border.Effect>
            <DropShadowEffect x:Name="shadowEf"
              ShadowDepth="0"></DropShadowEffect>
        </Border.Effect>
    <MediaElement Source="WildLife.wmv">
        <MediaElement.Effect>
            <BlurEffect x:Name="blurEf" Radius="0"></BlurEffect>
        </MediaElement.Effect>
    </MediaElement>
    </Border>
    <StackPanel Grid.Row="1">
        <Slider Minimum="0"
            Maximum="50"
            Value="{Binding Radius, ElementName=blurEf,
Mode=TwoWay}">
        </Slider>
        <Slider Minimum="0"
            Maximum="50"
            Value="{Binding ShadowDepth, ElementName=shadowEf,
Mode=TwoWay}">
        </Slider>
    </StackPanel>
</Grid>
```

Вот что мы получим после запуска приложения (рис. 8.14).

Этими двумя эффектами и ограничивается механизм встроенных эффектов, но разработчик может создать свои собственные эффекты. Рассмотрим пример создания собственного эффекта.

Собственные эффекты описывают с помощью специального языка HLSL. Описать эффект на этом языке — самый сложный шаг в создании эффекта. Правда, без труда можно найти уже готовое описание нужного эффекта. Так ниже приводится описание эффекта, создающего волну:

```
sampler2D Input : register(s0);
float4 main(float2 uv : TEXCOORD) : COLOR
{
    float4 Color;
    uv.y = uv.y + (sin((uv.x)*200)*0.01);
    uv.x = uv.x + (cos((uv.y)*200)*0.01);
    Color = tex2D(Input, uv.xy);
    return Color;
```

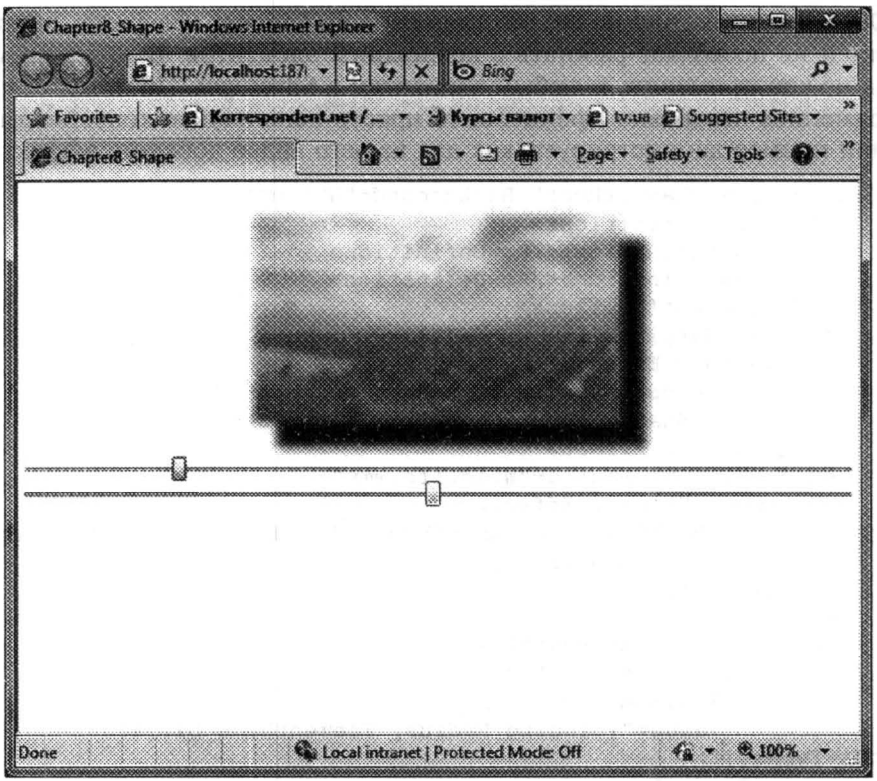


Рис. 8.14. Использование BlurEffect и DropShadowEffect

Обычно описания эффектов хранят в файлах с расширением **.fx**. Чтобы подготовить эффект к использованию в WPF или Silverlight, HLSL код необходимо сначала откомпилировать. Это можно сделать с помощью утилиты **fxc.exe**, которая входит в состав **DirectX SDK**.

Откомпилировав эффект выше и получив файл .ps (от Pixel Shader), добавьте его в проект в виде ресурса.

Остается создать класс, который бы загружал наш эффект и объект которого можно было бы использовать в нашем коде. Вот пример реализации такого класса:

```
public class WaveEffect : ShaderEffect
{
    public WaveEffect()
    {
        PixelShader = _shader;
        UpdateShaderValue(InputProperty);
    }
    public Brush Input
    {
        get { return (Brush)GetValue(InputProperty); }
        set { SetValue(InputProperty, value); }
    }
    public static readonly DependencyProperty InputProperty =
```

```
        ShaderEffect.RegisterPixelShaderSamplerProperty(
                "Input",
                typeof(WaveEffect),
                0);
    private static PixelShader _shader =
        new PixelShader() { UriSource = new Uri(
            "SilverlightApplication51;component/WaveEffect.ps",
            UriKind.Relative) };
}
```

Добавив еще один контейнер в код выше, используем созданный эффект:

```
<Border Margin="25">
            <Border.Effect>
                <local:WaveEffect></local:WaveEffect>
            </Border.Effect>
. . . . . .
```

В результате работы кода получим следующее приложение:

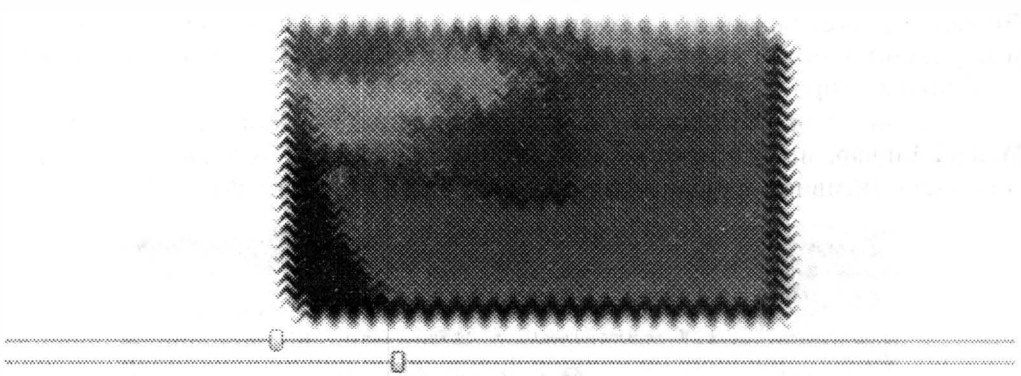


Рис. 8.15. Создание собственного эффекта

Pixel API

Предыдущие версии SilverLight были лишены возможности работать с растровыми изображениями как с набором пикселей. Начиная с SilverLight 3, эта проблема решена с помощью объекта **WritableBitmap**, который позволяет не только генерировать новые изображения, но и использовать в качестве источника любую часть визуального дерева Silverlight. Создадим интерфейс приложения, состоящий из примитивной формы и панели, куда будет записываться сгенерированное изображение:

```
<UserControl x:Class="SilverlightApplication56.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="800" Height="600">
    <Grid x:Name="LayoutRoot" Background="White">
        <Grid.ColumnDefinitions>
```

```
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <StackPanel Grid.Column="0">
            <TextBox Width="100"></TextBox>
            <TextBox Width="100"></TextBox>
            <Button Width="100" Content="Update"
               Click="Button_Click"></Button>
        </StackPanel>
        <StackPanel x:Name="stk1" Grid.Column="1"
VerticalAlignment="Center">
        </StackPanel>
    </Grid>
</UserControl>
```

Чтобы пример был более эффектным, будем делать снимок не просто отдельной панели, а всего интерфейса, включая и панель со снимком. Чтобы реализовать такой код, нам понадобится класс **WritableBitmap**, который содержится в пространстве имен **System.Windows.Media.Imaging** и позволяет создать **Bitmap** контекст заданного размера (длина, ширина изображения). Доступ к изображению может быть осуществлен попиксельно. Тут используется простой индексатор.

Задавая имя **UIElement** и трансформацию в конструкторе класса **WritableBitmap**, мы фактически делаем снимок панели. Для выделения отдельной части активной панели можно использовать метод **Render**.

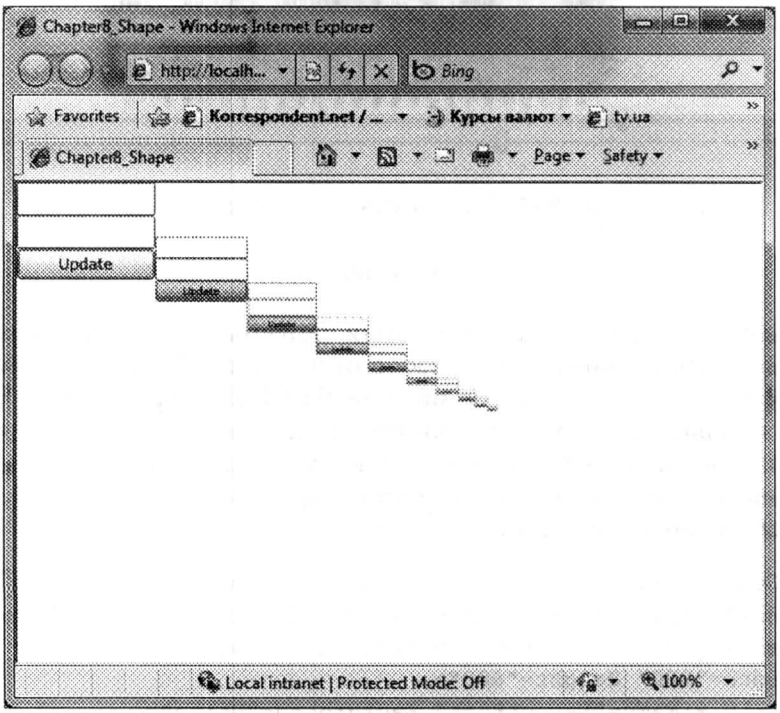


Рис. 8.16. Результат работы приложения

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    WriteableBitmap bit = new WriteableBitmap(
       LayoutRoot, new MatrixTransform());

    Image img = new Image();
    img.Source = bit;
    stk1.Children.Clear();
    stk1.Children.Add(img);
}
```

Вот так будет выглядеть наше приложение после многократного нажатия кнопки Update (рис. 8.16).

Работа с кэшем

Еще один нюанс, который возникает при работе с изображениями, это работа с кэшем (или не работа с ним). Так, для работы с изображениями (отображения изображений) в Silverlight присутствует специальный класс **BitmapImage**. Особенностью этого класса является то, что он не просто способен загрузить изображение, но и сохранить его в кэше браузера. Таким образом, если пользователь заходит второй раз на страницу приложения, то изображение может не загружаться, а выбираться из кэша, что снизит нагрузку на каналы передачи данных. Но, оказывается, это не всегда нужно!

Начиная с Silverlight 3, элемент **BitmapImage** поддерживает дополнительное значение свойства **CreateOptions** — **CreateOptions="IgnoreImageCache"**. Это свойство позволяет **выключить** кэш при загрузке изображений.

Примером, где это действительно нужно, может служить анти-спам элемент управления, который генерирует изображение для ввода пользователем. Таким образом предовращается реализация скрипта, автоматически заполняющего форму. Обычно такой элемент управления присутствует на страницах регистрации. Если пользователь ввел данные, представленные анти-спам элементом неправильно, то элемент должен сгенерировать новое содержимое и предоставить пользователю следующую попытку. Но при включенном кэше пользователь не сможет увидеть новую версию изображения, и система регистрации не будет работать.

Трансформация

Основные виды трансформаций

В Silverlight все визуальные элементы, которые наследуются от **UIElement**, поддерживают трансформации. Под трансформацией мы будем понимать преобразование относительно прямоугольной декартовой системы координат. В Silverlight существуют следующие виды трансформаций:

- **RotateTransform** — данный вид трансформации поворачивает заданный элемент на указанный угол и вокруг указанной точки;

- **ScaleTransform** — эта трансформация предназначена для сжатия или расширения элемента в заданное количество раз. В качестве параметров следует указать **ScaleX** и **ScaleY**, которые определяют, во сколько раз нужно сжать или расширить элемент по каждой из осей. Если значение этих свойств от 0 до 1, то элемент сжимается, а если больше 1, то элемент расширяется;
- **SkewTransform** — позволяет наклонить заданный элемент по осям X и Y. В качестве параметра принимает углы наклона **AngleX** и **AngleY**;
- **TranslateTransform** — позволяет переместить элемент на указанный промежуток по осям X и Y;
- **MatrixTransform** — позволяет задать матрицу трансформации для преобразования элемента.

Чтобы установить трансформацию у одного из элементов, достаточно создать объект соответствующей трансформации и заполнить этим объектом свойство элемента **RenderTransform**. Если Вы хотите использовать несколько трансформаций одновременно, то для заполнения **RenderTransform** используется **TransformGroup**, позволяющий объединить несколько видов трансформации. Рассмотрим пример, который объединяет все стандартные виды трансформации, при этом пользователь может менять их свойства с помощью ползунков. Вот код этого примера:

```
<StackPanel x:Name="LayoutRoot" Background="White">
    <Rectangle Width="200" Height="100" Fill="Red">
        <Rectangle.RenderTransform>
            <TransformGroup>
              <RotateTransform Angle=
                "{Binding Value, ElementName=rotateSlider,
Mode=OneWay}"
                CenterX="100" CenterY="50">
              </RotateTransform>

              <ScaleTransform  CenterX="100" CenterY="50"
                ScaleX=
                "{Binding Value, ElementName=scaleXSlider,
Mode=OneWay}"
                ScaleY=
                "{Binding Value, ElementName=scaleYSlider,
Mode=OneWay}">
              </ScaleTransform>

              <SkewTransform CenterX="100" CenterY="50"
                AngleX=
                "{Binding Value, ElementName=skewXSlider,
Mode=OneWay}"
                AngleY=
                "{Binding Value, ElementName=skewYSlider,
Mode=OneWay}">
              </SkewTransform>

              <TranslateTransform X=
                "{Binding Value, ElementName=translateXSlider,
Mode=OneWay}"
```

```
            Y=
            "{Binding Value, ElementName=translateYSlide
=OneWay}">
          </TranslateTransform>
        </TransformGroup>
    </Rectangle.RenderTransform>
</Rectangle>
<Slider Width="200" Name="rotateSlider" Minimum="0"
   Maximum="360">
</Slider>
<Slider Width="200" Name="scaleXSlider" Minimum="0"
   Maximum="2" Value="1">
</Slider>
<Slider Width="200" Name="scaleYSlider" Minimum="0"
   Maximum="2" Value="1">
</Slider>
<Slider Width="200" Name="skewXSlider" Minimum="-180"
   Maximum="180" Value="0">
</Slider>
<Slider Width="200" Name="skewYSlider" Minimum="-180"
   Maximum="180" Value="0">
</Slider>
<Slider Width="200" Name="translateXSlider"
   Minimum="-100" Maximum="100" Value="0">
</Slider>
<Slider Width="200" Name="translateYSlider"
   Minimum="-100" Maximum="100" Value="0">
</Slider>
ackPanel>
```

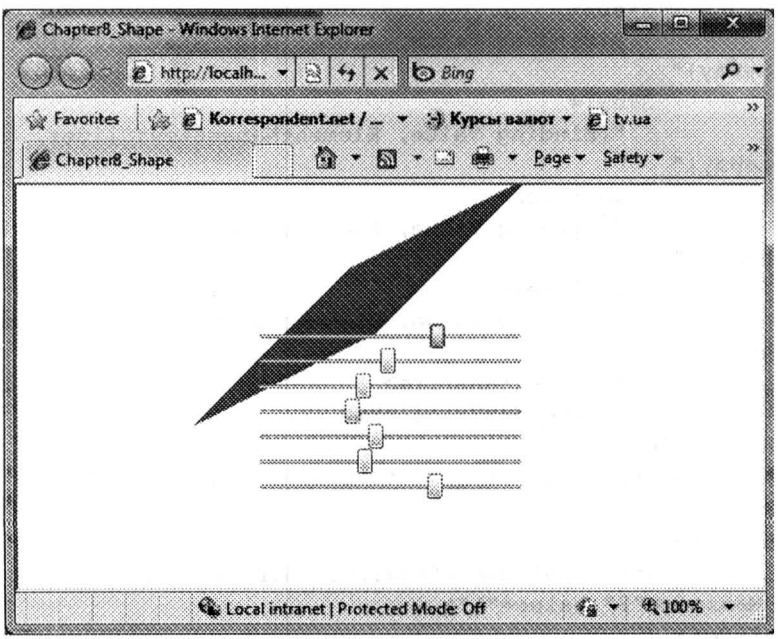


Рис. 8.17. Результат работы приложения

Результат работы приложения показан на рис. 8.17.

Обратите внимание на то, что трансформация не влияет на компоновку элементов, то есть элементы компоновки не учитывают трансформацию.

CompositeTransform в Silverlight 4

В Silverlight 4 появился дополнительный тип трансформации **CompositeTransform**, объединяющий все основные трансформации. Используя этот тип трансформации, разработчик тратит меньше времени на реализацию программного взаимодействия с интерфейсом. Хотя данный тип трансформации можно использовать только в том случае, если Вы хотите реализовать комбинацию трансформаций, привязанных к одному центру. Перепишем пример выше, используя новый тип трансформации:

```
<StackPanel x:Name="LayoutRoot" Background="White">
    <Rectangle Width="200" Height="100" Fill="Red">
        <Rectangle.RenderTransform>
            <CompositeTransform CenterX="100" CenterY="50"
              ScaleX=
                "{Binding Value, ElementName=scaleXSlider,
Mode=OneWay}"
              ScaleY=
                "{Binding Value, ElementName=scaleYSlider,
Mode=OneWay}"
              Rotation=
                "{Binding Value, ElementName=rotateSlider,
Mode=OneWay}"
              SkewX=
                "{Binding Value, ElementName=skewXSlider,
Mode=OneWay}"
              SkewY=
                "{Binding Value, ElementName=skewYSlider,
Mode=OneWay}"
              TranslateX=
                "{Binding Value, ElementName=translateXSlider,
Mode=OneWay}"
              TranslateY=
                "{Binding Value, ElementName=translateYSlider,
Mode=OneWay}">
            </CompositeTransform>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Slider Width="200" Name="rotateSlider" Minimum="0"
       Maximum="360">
    </Slider>
    <Slider Width="200" Name="scaleXSlider" Minimum="0"
       Maximum="2" Value="1">
    </Slider>
    <Slider Width="200" Name="scaleYSlider" Minimum="0"
```

```
        Maximum="2" Value="1">
    </Slider>
    <Slider Width="200" Name="skewXSlider" Minimum="-180"
        Maximum="180" Value="0">
    </Slider>
    <Slider Width="200" Name="skewYSlider" Minimum="-180"
        Maximum="180" Value="0">
    </Slider>
    <Slider Width="200" Name="translateXSlider"
        Minimum="-100" Maximum="100" Value="0">
    </Slider>
    <Slider Width="200" Name="translateYSlider"
        Minimum="-100" Maximum="100" Value="0">
    </Slider>
</StackPanel>
```

Как видно, код стал выглядеть более приятно, а разработчик теперь взаимодействует с одним объектом вместо четырех.

Трехмерные проекции

Начиная с Silverlight 3, разработчикам стал доступен новый элемент **PlaneProjection**, который позволяет реализовывать наиболее распространенные задачи по работе в трехмерном пространстве. Элемент довольно простой. С одной стороны его можно использовать по отношению к любому элементу или даже некоторым контейнерам (**Border**), а с другой — тут довольно простые свойства, позволяющие задать углы поворота и расположить объект в трехмерном пространстве. Вот пример использования объекта **PlaneProjection** по отношению к **MediaElement**, отображающему видео:

```
<StackPanel x:Name="LayoutRoot" Background="White">
    <MediaElement Source="WildLife.wmv" Width="400" Height="300">
        <MediaElement.Projection>
            <PlaneProjection
             RotationX=
              "{Binding Value, ElementName=rotateXSlider,
Mode=OneWay}"
             RotationY=
              "{Binding Value, ElementName=rotateYSlider,
Mode=OneWay}"
             RotationZ=
               "{Binding Value, ElementName=rotateZSlider,
Mode=OneWay}">
            </PlaneProjection>
        </MediaElement.Projection>
    </MediaElement>
    <Slider Width="400" Minimum="0" Maximum="360"
        Name="rotateXSlider">
    </Slider>
```

```
        <Slider Width="400" Minimum="0" Maximum="360"
            Name="rotateYSlider">
        </Slider>
        <Slider Width="400" Minimum="0" Maximum="360"
            Name="rotateZSlider">
        </Slider>
    </StackPanel>
```

Тут использовались свойства **RotationX**, **RotationY**, **RotationZ**, позволяющие развернуть элемент по каждой из осей в трехмерном пространстве на заданное количество градусов. В результате работы этого кода на экране можно увидеть следующее приложение:

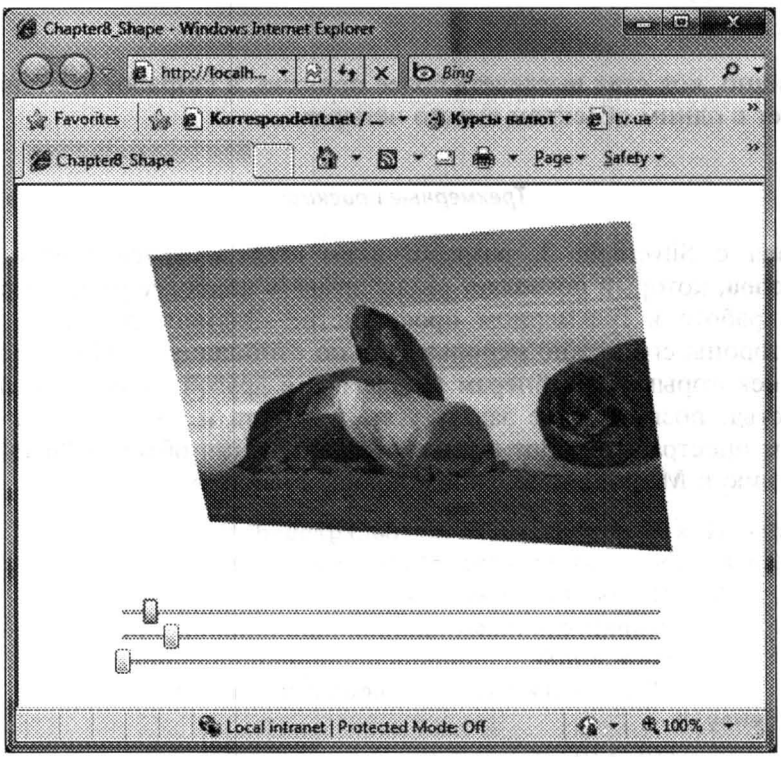


Рис. 8.18.

Обратите внимание, что в отличие от трансформаций, проекции имеют свой контейнер, которые не пересекается с **RenderTransform**.

Дополнительно в элементе **PlaneProjection** существуют атрибуты, позволяющие задать начало координат в трехмерном пространстве, а также смещение элемента по каждой из осей. При этом смещение можно отсчитывать относительно текущего контейнера или всего окна. Вот эти свойства: **CenterOfRotationX**, **CenterOfRotationY**, **CenterOfRotationZ**, **LocalOffsetX**, **LocalOffsetY**, **LocalOffsetZ**, **GlobalOffsetX. GlobalOffsetY. GlobalOffsetZ**.

Введение в анимацию

Общие типы анимации

Silverlight позволяет использовать анимацию для изменения свойств любого из объектов, реализуемых от **UIElement**. Анимация представляет собой мощный механизм, который позволяет создавать красивые эффекты, практически не используя код на C#.

Прежде чем перейти к созданию примеров, использующих анимацию, рассмотрим доступные в Silverlight типы анимации.

DoubleAnimation. Анимация **DoubleAnimation** используется для создания анимации любых свойств типа **double**. Основными свойствами данного типа анимации являются **From** и **To**, которые задают начальное и конечное значение анимируемого свойства. Если начальное значение свойства неизвестно, то можно воспользоваться еще одним свойством **By**, которое позволяет задать относительное значение для анимируемого свойства;

ColorAnimation. Тип анимации **ColorAnimation** используется для анимации свойств, которые имеют тип **Color**. На тип анимируемого свойства нужно обратить особое внимание. Так, если Вы решили анимировать такие свойства как **Fill** или **Stroke** для графических примитивов, то нужно отталкиваться не от этих свойств, а от свойств кистей. Основными свойствами данного типа анимации являются те же, что и в предыдущем случае, только вместо значений типа **double** задается цвет;

PointAnimation. Анимация **PointAnimation** позволяет анимировать точку. Основные свойства тут те же, что и в случае с **DoubleAnimation**, только в качестве значений задаются координаты точек;

Независимо от типа анимации, все объекты обладают такими свойствами:

- **Duration** — это свойство задает время продолжительности анимации в формате ЧЧ:ММ:СС (00:00:03 — 3 секунды). Все перечисленные виды анимации позволяют изменять анимируемые свойства равномерно, разбивая приращение по заданному интервалу времени;
- **BeginTime** — если планируется запустить анимацию не сразу, а только через некоторый промежуток времени, то этот промежуток можно определить с помощью свойства **BeginTime**, указав время начала анимации;
- **SpeedRatio** — это свойство позволяет ускорить анимацию в n раз. Может быть использовано для реализации различных проигрывателей анимации, предоставляя механизм ускорения или замедления процесса;
- **AutoReverse** — это свойство принимает булевское значение и позволяет определить необходимость выполнить анимацию в обратном порядке. При этом если продолжительность анимации 5 секунд, то с установленным свойством **AutoReverse** в **true**, общая продолжительность анимации составит 10 секунд;
- **RepeatBehavior** — последнее свойство, которое позволяет установить возможность повторения анимации. Тут возможны следующие значения:
  - ✓ **Время в секундах** — после прохождения промежутка времени, заданного здесь, анимация будет запускаться повторно;
  - ✓ **Forever** — запускает анимацию повторно сразу после ее окончания;

- ✓ **Nx** — тут **N** необходимо заменить на число, определяющее количество повторений анимации. Буква **x** была добавлена, чтобы не путать это значение со временем в секундах.

Поскольку в одном объекте могут анимироваться сразу несколько свойств, объекты анимации не могут быть «беспризорными». Чтобы объединить нужное количество объектов анимации, используется специальный объект типа **StoryBoard**. В задачи этого объекта входит контроль и запуск анимации. Кроме этого он использует два зависимых свойства **StoryBoard.TargetName** и **StoryBoard.TargetPropetry** для определения имени объекта, по отношению к которому направлена анимация, и имени свойства, которое будет анимироваться.

Вот пример определения объекта **StoryBoard**, который содержит лишь одну анимацию:

```
<Storyboard>
   <DoubleAnimation
      Storyboard.TargetName="myButton"
      Storyboard.TargetProperty="(Canvas.Left)"
      To="200" Duration="0:0:5" AutoReverse="True" />
</Storyboard>
```

Разобравшись с типами анимации, рассмотрим каким образом можно запускать анимацию.

Запуск анимации

Для запуска анимации существует два механизма:

- Запуск анимации с помощью XAML в момент загрузки объекта;
- Запуск анимации из кода на C#;

Чтобы реализовать запуск анимации из XAML, достаточно определить элемент **EventTrigger**, который разместить в коллекции **Triggers** нужного нам элемента управления. Триггер позволяет реагировать на какое-то событие, так **EventTrigger** позволяет реагировать на событие, связанное с элементом управления. Несмотря на наличие коллекции **Triggers** у каждого из визуальных элементов управления, в Silverlight существует только **EventTrigger**, да и то, он способен реагировать лишь на событие **Loaded**. Рассмотрим небольшой пример заполнения триггера нашей анимацией:

```
<Button x:Name="myButton" Canvas.Top="0" Canvas.Left="0"
Width="100"
 Height="50" Content="Hello">
    <Button.Triggers>
        <EventTrigger RoutedEvent="UserControl.Loaded">
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimation
                     Storyboard.TargetName="myButton"
                     Storyboard.TargetProperty="(Canvas.Left)"
                     To="200" Duration="0:0:5" AutoReverse="True"
/>
                </Storyboard>
            </BeginStoryboard>
```

```
        </EventTrigger>
    </Button.Triggers>
</Button>
```

Второй способ, который является более универсальным, — это размещение объекта **StoryBoard** в ресурсах, связанных с элементом или приложением (о ресурсах речь пойдет в главе 10) с последующим запуском из управляемого кода. Этот способ наиболее универсальный и гибкий, так как позволяет запустить анимацию в момент любого события. Чтобы управлять объектом **StoryBoard** из кода, существует несколько простых методов:

- **Begin**;
- **Stop**;
- **Pause**.

Кроме этих методов существует очень полезное событие Completed, которое позволяет определить момент завершения анимации.

Анимация с помощью ключевых кадров

Стандартные типы анимации позволяют изменить свойства объектов с течением времени, используя равномерное приращение свойств. Но, если Вы хотите изменять свойства не по линейному закону, то типы анимации, рассмотренные ранее, не подходят. Вместо них в Silverlight доступны аналогичные типы анимации, но с возможностью разбивать весь процесс на отдельные фрагменты. Вот список типов, описывающих такой класс анимации:

- **DoubleAnimationUsingKeyFrames**;
- **PointAnimationUsingKeyFrames**;
- **ColorAnimationUsingKeyFrames**.

Как видно к названию стандартных типов анимации был добавлен суффикс **UsingKeyFrames**.

Рассмотрим пример использования **DoubleAnimationUsingKeyFrames**:

```
<Storyboard>
   <DoubleAnimationUsingKeyFrames Duration="0:0:5"
      AutoReverse="True" Storyboard.TargetName="myButton"
      Storyboard.TargetProperty="(Canvas.Left)">

      <LinearDoubleKeyFrame KeyTime="0:0:2"
          Value="70"></LinearDoubleKeyFrame>
      <LinearDoubleKeyFrame KeyTime="0:0:5"
          Value="100"></LinearDoubleKeyFrame>

    </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

Как видно из кода, линейное приращение задается с помощью объектов **LinearDoubleKeyFrame**. В данном случае эти объекты задают перемещение кнопки на 70 единиц за первые две секунды и на 30 единиц за следующие три секунды.

Кроме **LinearDoubleKeyFrame** мы могли бы использовать **DiscreteDoubleKeyFrame** или **SplineDoubleKeyFrame**. Первый из объектов задает дискретное приращение свойства, то есть наш объект начнет двигаться прыжка-

ми. Второй из объектов анимации задает изменение свойства по математическому закону, используя кривую Безье.

Аналогичные механизмы изменения свойств присутствуют и в двух других типах анимации.

В рассмотренном классе объектов анимации существует еще один тип анимации — **ObjectAnimationUsingKeyFrame**, способный изменять любые свойства у любого объекта. Тут поддерживается лишь дискретный набор кадров, требующий задать состояние объекта в желаемые промежутки времени, без возможности автоматического приращения свойств (еще бы, тип свойства может ведь не укладываться в поддерживаемые).

Простая анимация

Выше мы рассмотрели анимацию с помощью ключевых кадров. Подобная анимация позволяет изменять свойства практически по любому закону, но очень часто требует большого количества кода и усилий для описания простейшей функции. Например, чтобы выполнить анимацию для мячика, падающего на поверхность, использовать один объект анимации было недостаточно. Так, очень сомнительно, что мячик упадет на поверхность и не подскачет вверх. Если же программист хотел реализовать прыгающий мячик, то приходилось реализовывать каждый скачок в виде отдельного кадра, что значительно увеличивало код и сложность приложения.

Начиная с третьей версии, Silverlight поддерживает несколько полезных функций для решения ряда распространенных задач, прилагая минимум усилий. Ниже показан код, который использует функцию, позволяющую достичь эффекта прыгающего мячика:

```
<UserControl x:Class="SilverlightApplication52.MainPage"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300" Loaded="UserControl_Loaded">
    <UserControl.Resources>
        <Storyboard x:Name="sb1">
            <DoubleAnimation From="0" To="250"
                             Storyboard.TargetName="el1"
                          Storyboard.TargetProperty="(Canvas.Top)"
                             Duration="0:0:5">
                <DoubleAnimation.EasingFunction>
                    <BounceEase Bounces="10"
                       EasingMode="EaseOut" Bounciness="2">
                     </BounceEase>
                </DoubleAnimation.EasingFunction>
            </DoubleAnimation>
        </Storyboard>
    </UserControl.Resources>
    <Canvas x:Name="LayoutRoot" Background="White">
        <Ellipse Fill="Blue" Width="50" Height="50"
x:Name="el1"></Ellipse>
    </Canvas>
</UserControl>
```

Как Вы видите, тут была использована специальная функция **BounceEase**, которая описывает функцию с затухающей амлитудой, позволяющую нам реализовать «прыжки» мячика.

На текущий момент существует 11 простых функций анимации. Вот полный список: **ExponentialEase**, **PowerEase**, **QuadraticEase**, **BackEase**, **BounceEase**, **CircleEase**, **CubicEase**, **ElasticEase**, **QuarticEase**, **QuinticEase**, **SineEase**.

Каждая из функций имеет свои параметры, позволяющие задать коэффициенты. Кроме того, с помощью специального свойства **EasingMode** можно с легкостью получить зеркальное отображение функции.

Если для создания простой анимации использовать Expression Blend, то разработчик получает в свое распоряжение подсказки, которые делают использование анимации действительно простой задачей:

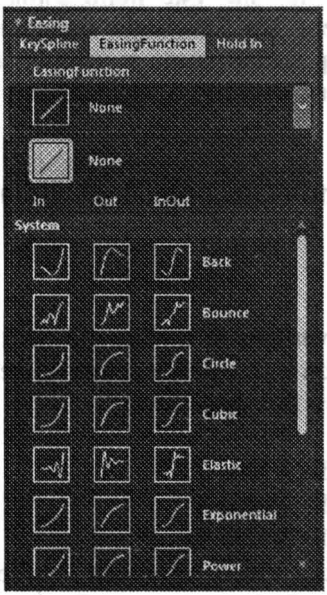


Рис. 8.19. Выбор простых функций в Expression Blend

Заключение

Комбинируя графические примитивы, кисти и стандартные элементы управления, можно добиться совершенно уникальных интерфейсов. При этом использование анимации и трансформации может добавить некоторую изюминку Вашему интерфейсу. Трехмерные проекции можно использовать для разработки новых элементов навигации (например, карусель). О том, как использовать материал этой главы для создания собственных элементов управления, мы будем говорить в главе 11.

Глава 9

РАБОТА С АУДИО И ВИДЕО

Пришло время познакомиться с возможностями отображения видео в Silverlight. В примерах выше, мы уже использовали **MediaElement**, который способен отображать видео. Но Silverlight предоставляет не просто элемент управления, а целый набор механизмов и утилит, которые мы и рассмотрим. Естественно, что начнем мы именно с **MediaElement**.

Использование MediaElement

Общие сведения

Начнем с того, что Silverlight 4, в отличие от первой версии, уже поддерживает целый ряд форматов для отображения видео и проигрывания аудио в элементе **MediaElement**. Вот эти форматы:

- Raw Video;
- WMV1: Windows Media Video 7;
- WMV2: Windows Media Video 8;
- WMV3: Windows Media Video 9;
- WMVA: Windows Media Video Advanced Profile, non-VC-1;
- WVC1: Windows Media Video Advanced Profile, VC-1;
- H264 (ITU-T H.264 / ISO MPEG-4 AVC).

Как видно, к различным форматам Windows Media был добавлен MPEG-4. Последний пользуется достаточно большой популярностью при распространении видео.

Если Ваше видео в формате, который не указан выше, то его необходимо преобразовать в один из поддерживаемых форматов с помощью одной из утилит, например Expression Encoder, о которой речь пойдет дальше.

Рассмотрим, какие форматы поддерживаются для аудио:

- WAV;
- WMA7: Windows Media Audio 7;
- WMA8: Windows Media Audio 8;
- WMA9: Windows Media Audio 9;
- WMA10: Windows Media Audio 10;
- MP3: ISO/MPEG-1 Layer 3;
- AAC.

И, наконец, если говорить о протоколах, которые поддерживает **MediaElement**, то это **http**, **https** и **mms**. Последний из протоколов использует Windows Media Service для трансляции потокового видео. В зависимости от указанного протокола, **MediaElement** пытается выполнить или прогрессивную, или потоковую загрузку. Если Вы указываете **http** или **https**, то сначала производится попытка выполнить прогрессивную загрузку, а в случае неудачи — потоковую. Если вы указываете **mms**, то сначала будет попытка выполнить потоковую загрузку.

Если говорить о прогрессивной загрузке, то тут есть два варианта:

- разместить видео на сервере — при большом количестве видеороликов этот способ наиболее оптимален, так как позволяет загрузить видео только по запросу. Кроме того, благодаря специальным модулям в Internet Information Services 7, Web-сервере от Microsoft, можно значительно оптимизировать прогрессивную загрузку;
- разместить видео в виде ресурсов Silverlight-приложения — данный способ оптимален, если размер ваших видеофайлов не дает большой прирост к размеру пакета. Также существует ряд сценариев, которые могут потребовать установку приложения на локальный компьютер пользователя для работы в разъединенном окружении. Чтобы разместить видео в качестве ресурсов Silverlight-приложения, достаточно добавить файл в проект и установить свойство (в окне свойств) **Build Action** в **Resource**.

Если говорить о потоковой загрузке видео, то тут доступны следующие возможности:

- трансляция «живого» видео — речь идет о трансляции видео в режиме реального времени. Само видео может поступать либо со спутника, либо с записывающего устройства. Для реализации этого механизма требуется выделить компьютер, который будет заниматься кодированием видео в цифровой формат, поддерживаемый Silverlight. Еще недавно кодированием видео занимался продукт Windows Encoder, но теперь он в прошлом. На смену Windows Encoder пришел Microsoft Expression Encoder, который и занимается обработкой входящего сигнала (равно как и другими преобразованиями). Обработанное видео передается на сервер, где установлен Windows Media Service, осуществляющий трансляцию в Web, используя протокол **mms**;
- потоковая трансляция записанного видео — Windows Media Service можно использовать и для трансляции уже записанного видео. По большому счету Silverlight не видит разницы между «живым» и записанным видео. Silverlight-приложение просто получает поток данных по протоколу **mms**;
- трансляция в формате Smooth Streaming — Smooth Streaming представляет собой новую технологию, которая позволяет отобразить видео на клиенте в том качестве, которое зависит от пропускной способности его канала и способности процессора обработать входящие данные. Идея состоит в том, чтобы хранить на сервере сразу несколько вариантов кодировки видео, которое адаптировано для различных каналов связи и имеет разное качество. Благодаря специальному модулю в Internet Information Services 7, определяется, какое качество может быть использовано для отображения видео в следующие несколько секунд. Выбирается соответ-

ствующий пакет из нужного варианта кодировки и отправляется пользователю. Если в данный момент времени канал пользователя способен пропустить большой пакет, то видео передается в хорошем качестве, а если нет, то качество видео начинает ухудшаться. Данный механизм позволяет организовывать трансляцию видео в хорошем качестве, позволяя пользователю смотреть видео без «тормозов»;

- трансляция «живого» видео в формате Smooth Streaming — благодаря специальному модулю в Internet Information Services 7, возможно применить технологию Smooth Streaming и для живого видео. Однако, на практике это можно реализовать лишь с большими вложениями средств. Ведь Microsoft Expression Encoder не способен выполнять кодирование «живого» видео в формат Smooth (только записанного). Фактически, чтобы получить несколько потоков, Вам необходимо запустить несколько экземпляров Expression Encoder, каждый из которых должен владеть своим экземпляром видеокарты. Отсюда следует, что нужно использовать несколько серверов, либо один мощный сервер, который имеет несколько видеокарт для приема сигнала. Но, даже если у Вас и будет такое оборудование, Вы столкнетесь с проблемой синхронизации видеопотоков, возвращаемых каждым из декодеров. Единственное решение — это использовать специальное оборудование, специально разработанное для кодирования живого видео в формат Smooth. Данное оборудование существует и даже использовалось при трансляции олимпийских игр в Пекине и Ванкувере. Но стоимость одного такого сервера составляет около 10-15 тысяч долларов.

Итак, разобравшись с форматами, протоколами и вариантами трансляции, перейдем к возможностям элемента управления **MediaElement**. Пока будем говорить только о прогрессивной загрузке видео, а к потоковому видео перейдем в следующих разделах.

Если посмотреть на иерархию классов, то **MediaElement** является прямым наследником **FrameworkElement**. Это означает, что он обладает большинством свойств, характерных для других элементов управления. Но несколько свойств характерны только для этого элемента. Рассмотрим эти свойства:

- **Source** — данное свойство позволяет установить путь к видеофайлу (потоку), который должен быть отображен в **MediaElement**. Кроме этого свойства существует метод **SetSource**, о котором мы поговорим ниже;
- **AutoPlay** — свойство позволяет указать, нужно ли проигрывать указанное видео или аудио сразу после загрузки Silverlight-приложения. Имейте в виду, что по умолчанию это свойство установлено в **true**. Между тем, если Ваше видео или аудио не передается вместе с Silverlight-приложением, то лучше установить значение в **false**. Пользователя очень раздражает, когда канал используется для загрузки больших объемов информации без его ведома. Часто пользователь открывает сразу несколько закладок в браузере, где Ваше приложение — лишь одна из закладок. И первое, что приходит ему в голову, когда он слышит какие-то звуки из своего компьютера, это быстро найти и закрыть закладку, издающую эти звуки☺
- **IsMuted** — позволяет отключить звук, если свойство установлено в **true**.

- **Stretch** — это свойство позволяет задать, каким образом видео будет заполнять **MediaElement**. Тут возможны следующие значения: **None**, **Fill**, **Uniform**, **UniformToFill**. Давайте поговорим об этом свойстве в следующем абзаце;
- **Volume** — задает текущий уровень громкости. Значение громкости варьируется от 0 до 1. По умолчанию значение установлено в 0.5. Поэтому, если Ваш интерфейс не предоставляет механизма регулирования громкости, то установите **Volume** в 1. В противном случае получите много отзывов о том, что слишком тихо;
- **Balance** — это свойство принимает значения в диапазоне от −1 до 1. По умолчанию это свойство установлено в 0, что гарантирует одинаковый баланс между правым и левым аудиовыходами. Если значение установлено в −1, то весь звук будет направлен на левую колонку, а если +1, то на правую;
- **Position** — данное свойство содержит текущую позицию в видеофайле, отображаемом пользователю. Чтобы установить или получить свойство Position, необходимо использовать тип **TimeSpan**, который задает время в часах, минутах и секундах;
- **CanSeek** — это свойство позволяет определить, возможно ли использовать свойство **Position** для перемещения по видео. Ведь если **MediaElement** принимает потоковое видео, то ни о каком перемещении не может быть и речи;
- **CanPause** — определяет, можно ли приостановить воспроизведение контента. Если передается потоковое видео, то возвращает значение **false**;
- **CurrentState** — свойство определяет состояние медиа элемента и может содержать одно из значений перечислимого типа **MediaElementState**: **Buffering**, **Opening**, **Playing**, **Closed**, **Paused**, **Stopped**;
- **DownloadProgress** — определяет процент загруженного контента с сервера. Это свойство применимо при прогрессивной загрузке, когда файл с контентом загружается с сервера;
- **DownloadProgressOffset** — определяет смещение (позицию), с которого начинается загрузка фрагмента видео. Данное свойство устанавливается в том случае, если пользователь изменил текущую позицию на фрагмент, который еще не был загружен;
- **DroppedFramesPerSecond** — показывает количество фреймов в секунду, которые были выброшены из потока в связи с невозможностью их отобразить. Используется для потокового видео;
- **Markers** — содержит коллекцию маркеров, которые присутствуют в видео. О маркерах мы будем говорить в следующем разделе;
- **NaturalDuration** — это свойство определяет общую продолжительность видео или аудио фрагмента. Свойство задается с помощью экземпляра класса **TimeSpan**;
- **NaturalVideoWidth** — это свойство, а также свойство **NaturalVideoHeight**, задают исходную длину и ширину видео;
- **NaturalVideoHeight** — возвращает исходную ширину видео;
- **RenderedFramesPerSecond** — содержит количество отображаемых фреймов в секунду;

Вернемся к свойству **Stretch**. Создавая **MediaElement**, всегда можно указать его явные размеры, либо разместить его внутри элемента компоновки. В любом случае, отображаемое видео всегда будет подстраиваться к размеру **MediaElement**, с сохранением его пропорций. На самом деле, такое поведение возможно установленному по умолчанию свойству **Stretch** в значение **Uniform**. Именно свойство **Uniform** обеспечивает сохранение пропорций. В качестве примера рассмотрим небольшой код, содержащий **MediaElement**, принимающий в качестве **Source** файл из стандартной поставки Windows 7:

```
<UserControl x:Class="Chapter9_Media.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="400">
    <Grid x:Name="LayoutRoot" Background="Gray">
        <MediaElement Width="400" Height="400"
           Source="WildLife.wmv"></MediaElement>
    </Grid>
</UserControl>
```

Тут мы специально установили явные размеры родительского элемента и серый фон. Запустив это приложение, Вы сможете увидеть такую картину (рис. 9.1).

Как видите, несмотря на то, что данное изображение достаточно большого разрешения, оно было ужато до размеров медиа элемента с сохранением пропорций. При этом в MediaElement могут быть незаполненные участки.

Установим значение **Stretch** в **None** (рис. 9.2).

Как видно, в этом случае видео отображается с исходным разрешением, то есть без какого-либо сжатия, но отображается лишь та часть, которая смогла поместиться в медиа элемент (верхний левый угол видео).

Следующее значение, которое мы рассмотрим, — это **Fill**. Установим **Stretch** в значение **Fill** (рис. 9.3).

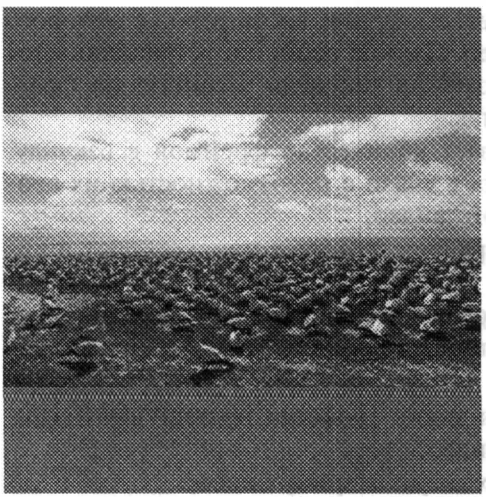


Рис. 9.1. Результат работы приложения (Stretch=Uniform)

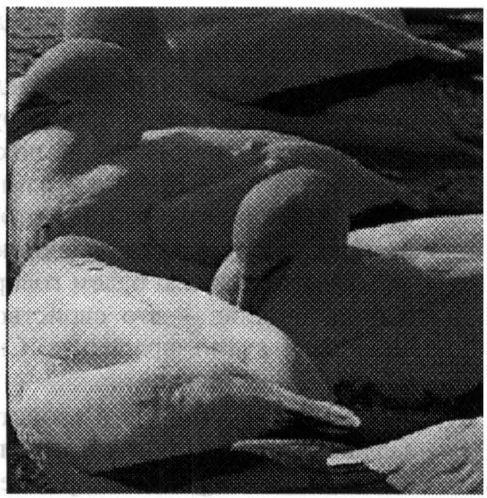


Рис. 9.2. Результат работы приложения (Stretch=None)

Рис. 9.3. Результат работы приложения (Stretch=Fill)

Рис. 9.4. Результат работы приложения (Stretch=UniformToFill)

В этом случае видео полностью заполняет область медиа элемента. В зависимости от пропорций медиа элемента видео может выглядеть сжатым или растянутым.

Последнее значение свойства **Stretch** — это **UniformToFill**. Рассмотрим результат работы этого свойства (рис. 9.4).

Как видно, видео полностью заполнило **MediaElement**, но пропорции были сохранены. Видео было ужато до таких размеров, чтобы отобразить максимально возможный участок в данном медиа элементе. Остальные участки отсекаются.

В дополнение к рассмотренным свойствам можно выделить свойство **Clip**. Именно его мы использовали во второй главе, отображая видео на фоне некоторой надписи. Свойство может содержать любую геометрию, создавая причудливые формы, заполненные видео.

Разобравшись со свойствами **MediaElement**, рассмотрим основные методы, позволяющие управлять видео:

- **SetSource** — метод **SetSource** позволяет установить свойство **Source** медиа элемента, когда имеется экземпляр типа **Stream** или **MediaStreamSource**. В первом случае метод используется, когда загрузка осуществляется с помощью класса **WebClient**, который позволяет получить загруженный файл в виде потока. Во втором случае, **MediaStreamSource** является базовой площадкой для разработки собственного декодера. Так, используя **MediaStreamSource** в качестве базового класса, разработчик может преобразовать любой поток с данными в формат, пригодный для отображения медиа элементом;
- **Play** — запускает видео или аудио для проигрывания. Если видео передается в виде потока, то просто начинает выбирать данные из потока;
- **Pause** — останавливает проигрывание контента. Если видео передается в виде потока, то вызов метода будет проигнорирован;
- **Stop** — останавливает воспроизведение.

Ну и, наконец, самое важное, — это события, которые связаны с **MediaElement**. Ведь именно реакция на события позволит сделать Ваш интерфейс наиболее динамичным и привлекательным для пользователя. Рассмотрим все характерные события для медиа элемента:

- **BufferingProgressChanged** — генерируется при изменении свойства **BufferingChanged**, то есть при буферизации очередного фрагмента видео;
- **CurrentStateChanged** — генерируется при изменении состояния медиа элемента;
- **DownloadProgressChanged** — генерируется при изменении процента загрузки видео с сервера;
- **MarkerReached** — генерируется в тот момент, когда был достигнут один из маркеров, установленных внутри видео. О маркерах мы поговорим в следующем разделе;
- **MediaEnded** — событие происходит в тот момент, когда медиа элемент прекратил проигрывать видео или аудио;
- **MediaFailed** — это важное событие, так как именно оно позволяет реагировать на ошибки, связанные с чтением медиа контента, в управляемом коде. Если обработчика на это событие не найдено, то ошибка будет передана в JavaScript, а приложение прекратит свою работу;
- **MediaOpened** — происходит, когда медиа поток был успешно открыт и была получена начальная информация о нем.

Рассмотрев такое количество свойств, методов и событий, можно сделать вывод, что **MediaElement** обладает всем необходимым для реализации интерфейса любой сложности. Давайте реализуем небольшой пример, демонстрирующий некоторые свойства, события и методы медиа элемента.

Реализуем интерфейс нашего приложения следующим образом:

```
<UserControl x:Class="Chapter9_Media.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="450">
    <StackPanel x:Name="LayoutRoot">
        <MediaElement Name="myMedia" Height="300"
            Source="WildLife.wmv" Stretch="Uniform"
            Margin="5" AutoPlay="False"
            MediaOpened="myMedia_MediaOpened"
            MediaEnded="myMedia_MediaEnded">
        </MediaElement>
        <StackPanel Orientation="Horizontal">
            <TextBlock Name="durationText"
               Text="Duration: " Margin="5">
            </TextBlock>
            <TextBlock
               Text=
               "{Binding Position, ElementName=myMedia,
Mode=OneWay}"
               Margin="5">
            </TextBlock>
            <TextBlock Text="/" Margin="5"></TextBlock>
            <TextBlock Text="" Name="secondsText" Margin="5">
```

```
            </TextBlock>
        </StackPanel>
        <Slider Name="positionSlider" Minimum="0"
            Margin="5" IsEnabled="False"
            Value=
          "{Binding Position.TotalSeconds, ElementName=myMedia,
Mode=OneWay}"
        >
        </Slider>
        <StackPanel Orientation="Horizontal" >
            <Button Content="Play" Name="playButton" Margin="5"
               Width="100" IsEnabled="False"
               Click="playButton_Click"></Button>
            <Button Content="Pause" Name="pauseButton"
               IsEnabled="False" Margin="5" Width="100"
               Click="pauseButton_Click" ></Button>
            <Button Content="Stop" Name="stopButton"
               Margin="5" Width="100" IsEnabled="False"
               Click="stopButton_Click" ></Button>
            <CheckBox Content="Mute" Margin="5" IsEnabled="False"
               Name="muteBox" IsChecked=
            "{Binding IsMuted, ElementName=myMedia, Mode=TwoWay}"
        >
        </CheckBox>
        </StackPanel>
        <StackPanel Orientation="Horizontal" >
            <TextBlock Text="Volume:" Margin="5"></TextBlock>
            <Slider Name="volumeSlider" Minimum="0" Maximum="1"
               Width="200" IsEnabled="False"
               Value=
               "{Binding Volume, ElementName=myMedia,
Mode=TwoWay}">
            </Slider>
        </StackPanel>
    </StackPanel>
</UserControl>
```

Тут мы создали медиа элемент и множество элементов управления, которые позволят отображать текущую позицию, изменять громкость, останавливать видео и др.

Вот как должен выглядеть наш интерфейс (рис. 9.5).

Большинство элементов мы установили с помощью механизма связывания с данными, но какой-то код нам придется написать. Вот как будут выглядеть наши обработчики событий:

```
private void myMedia_MediaOpened(object sender, RoutedEventArgs e)
{
    positionSlider.IsEnabled = true;
    volumeSlider.IsEnabled = true;
    playButton.IsEnabled = true;
    stopButton.IsEnabled = true;

    positionSlider.Maximum =
```

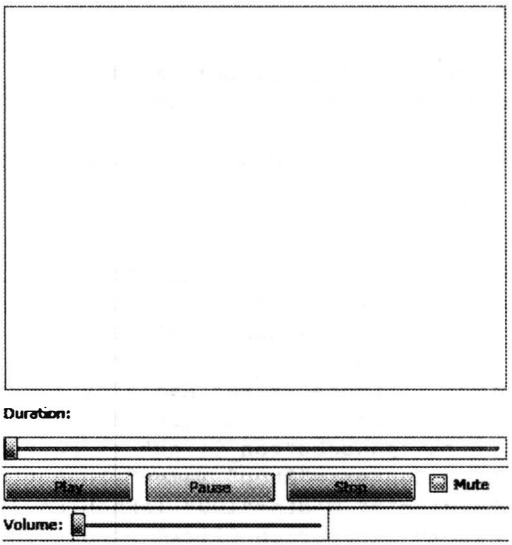


Рис. 9.5. Внешний вид интерфейса

```
        myMedia.NaturalDuration.TimeSpan.TotalSeconds;
    secondsText.Text =
        myMedia.NaturalDuration.ToString();
}

private void playButton_Click(object sender, RoutedEventArgs e)
{
    playButton.IsEnabled = false;
    pauseButton.IsEnabled = true;

    myMedia.Play();
}

private void pauseButton_Click(object sender, RoutedEventArgs e
{
    pauseButton.IsEnabled = false;
    playButton.IsEnabled = true;

    myMedia.Pause();
}

private void stopButton_Click(object sender, RoutedEventArgs e)
{
    pauseButton.IsEnabled = false;
    playButton.IsEnabled = true;

    myMedia.Stop();
}

private void myMedia_MediaEnded(object sender, RoutedEventArgs
{
    pauseButton.IsEnabled = false;
    playButton.IsEnabled = true;
```

То, что мы делаем в этих обработчиках событий, — это вызываем соответствующие методы (**Stop**, **Start**, **Pause**) и активируем кнопки, которые должны быть доступны исходя из логики интерфейса.

Как видно, реализовать полноценный медиа плейер не очень сложно. Больше времени уйдет на дизайн интерфейса. Естественно, что если вы хотите реализовать универсальный плейер, который будет проигрывать и потоковое видео, то придётся добавить дополнительные проверки. Хотя идея от этого не изменится.

Перейдем еще к одной особенности медиа элемента — маркерам.

Использование маркеров

Маркеры представляют собой специальные метки, привязанные к конкретной точке на временной шкале видео. Эти метки способны содержать данные (текст) и могут быть созданы либо с помощью Microsoft Expression Encoder (тогда они сохраняются прямо в видео файле), либо динамически (такие метки сохраняются лишь в памяти).

Сценариев использования маркеров много. Например, маркеры могут быть использованы для создания обучающих уроков, когда в процессе отображения видео необходимо пройти промежуточные тесты или запустить анимацию.

Еще один хороший пример — это управление субтитрами. Фактически, чтобы ассоциировать субтитры с видео, достаточно создать коллекцию маркеров и при достижении очередного маркера (событие **MarkerReached**), отобразить субтитр на экран. Покажем использование маркеров на примере сценария с субтитрами.

Создадим простое приложение, содержащее медиа элемент и поле для отображения субтитров:

```
<UserControl x:Class="Chapter9_Markers.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="400">

    <StackPanel x:Name="LayoutRoot" Background="White">
        <MediaElement Width="400" Height="300" Name="myMedia"
           Source="Wildlife.wmv" MediaOpened="myMedia_MediaOpened"
           MarkerReached="myMedia_MarkerReached">
        </MediaElement>
        <TextBlock Text="" HorizontalAlignment="Center"
           Name="subtitleText" FontSize="16" FontWeight="Bold" >
        </TextBlock>
    </StackPanel>
</UserControl>
```

Было бы хорошо выбирать субтитры из XML или текстового файла. Но чтобы не писать много кода, я просто создам несколько экземпляров класса Subtitle, который я определил самостоятельно. Экземпляры этого класса будут содержать время и сам текст субтитров.

Заполнение коллекции маркеров должно произойти после успешной загрузки видео, а при достижении очередного маркера мы просто выдаем текст хранящийся в нем. Вот как выглядит код:

```
public partial class MainPage : UserControl
{
    Subtitle[] subtitles = new Subtitle [3];

    public MainPage()
    {
        InitializeComponent();

        Subtitle s=new Subtitle();
        s.text = "Побежали лошадки";
        s.time = new TimeSpan(0, 0, 0);
        subtitles[0] = s;

        s = new Subtitle();
        s.text = "Полетели птички";
        s.time = new TimeSpan(0, 0, 5);
        subtitles[1] = s;

        s = new Subtitle();
        s.text = "Суслики на пляже";
        s.time = new TimeSpan(0, 0, 7);
        subtitles[2] = s;
    }

    private void myMedia_MediaOpened(object sender, RoutedEventArgs e)
    {
        TimelineMarker t;
        foreach (Subtitle s in subtitles)
        {
            t = new TimelineMarker();
            t.Time = s.time;
            t.Text = s.text;
            myMedia.Markers.Add(t);
        }
    }

    private void myMedia_MarkerReached(object sender,
       TimelineMarkerRoutedEventArgs e)
    {
        subtitleText.Text = e.Marker.Text;
    }
}


class Subtitle
{
    public TimeSpan time;
    public String text;
```

Результат работы приложения показан на рисунке.

Рис. 9.6. Результат работы приложения

Таким образом, работа с маркерами не представляет сложностей. Вместе с тем, маркеры являются очень мощным инструментом при работе с видео.

Поддержка GPU

Начиная с третьей версии, Silverlight теперь включает поддержку GPU (прорисовку с привлечением возможностей процессора графической видеокарты).

Оптимизация через GPU в Silverlight делает только первые шаги и не предоставляет пока много возможностей. На графический процессор можно рассчитывать только в двух случаях: при отображении видео и при отображении растровых изображений. При этом речь идет не столько о простом отображении картинки или видео, как об использовании ряда эффектов. Фактически можно выделить три типа эффектов, которые поддерживаются в Silverlight:

а) трансформации изображения;

б) урезание прямоугольной части изображения;

с) смешивание пикселей в области медиа-элемента или рисунка (отображение каких-то рекламных вставок, панели инструментов, управление прозрачностью).

Пока GPU поддерживается только на Windows платформе. Совершенно не важно, работает ваше приложение в окне браузера или в полноэкранном режиме. Если говорить о Mac, то тут поддержка осуществляется только в полноэкранном режиме. В любом случае видеокарта должна поддерживать DirectX9.0C или выше.

Для включения оптимизации с помощью GPU необходимо:

а) задать эту опцию при создании плагина с помощью параметра **EnableGPUAcceleration**:

```
<param name="EnableGPUAcceleration" value="true" />
```

б) включить оптимизацию для конкретного элемента управления. Это делается с помощью свойства **CacheMode**, которое должно быть установлено в значение **BitmapCache**:

```
<MediaElement Source="a.wmv" Width="500" Height="300"
   CacheMode="BitmapCache">
```

При этом свойство автоматически устанавливается и для всех дочерних элементов дерева.

При тестировании приложений можно отображать все области экрана, для которых работает оптимизация с помощью GPU. Для этого используется параметр плагина **EnableCacheVisualization**:

```
<param name="EnableCacheVisualization" value="true" />
```

Возможности Internet Information Services 7

Если Вы все-таки рещились размещать Silverlight-приложения под управлением Internet Information Services, то сможете получить дополнительные преимущества при размещении видео на этом Web-сервере. В частности для видео с прогрессивной загрузкой существуют два интересных модуля — это **Web Playlists** и **Bit Rate Throttling**. Но прежде чем говорить об этих двух модулях, поговорим о специальном приложении **Web Platform Installer**.

Запуск Web Platform Installer

Web Platform Installer представляет собой специальную утилиту, которая позволит настроить Ваш Web-сервер без особых хлопот. Фактически, чтобы установить любой из дополнительных модулей к Internet Information Services 7, развернуть одно из приложений, адаптированных Microsoft для использования на Windows платформе или включить какие-то компоненты Web-сервера, достаточно установить **Web Platform Installer**.

Чтобы установить **Web Platform Installer**, обратитесь к сайту http://iis.net Тут находится громадное количество ресурсов, посвященных работе с IIS 7, а также все дополнительные расширения и информация о всевозможных проектах. Обычно, **Web Platform Installer**, можно легко установить с главной страницы этого сайта.

После установки **Web Platform Installer**, утилита определяет текущую конфигурацию системы и предоставляет набор опций по расширению ее возможностей. Вот как выглядит основное окно утилиты:

В конфигурации моей системы утилита отображает четыре отдельные вкладки. На первой вкладке представлены модули и приложения, которые недавно были включены в пакет инсталляции **Web Platform Installer**. Вторая вкладка позволяет получить доступ к настройкам Web-сервера, включая возможность инсталляции дополнительных модулей и утилит. Третья вкладка наиболее интересная, так как тут представлено несколько десятков приложений на

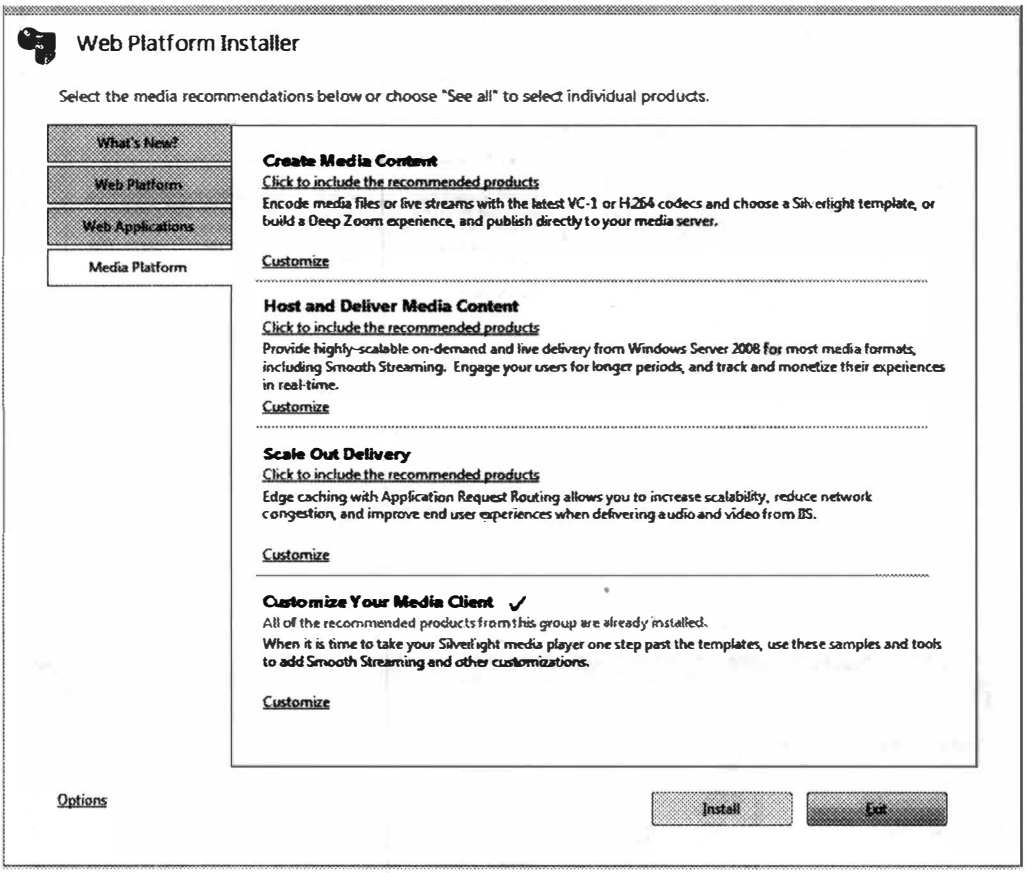


Рис. 9.7. Web Platform Installer

любой вкус. Многие из этих приложений давно завоевали популярность на платформе LAMP, а сейчас были мигрированы на Windows-платформу.

В этом разделе нас будет интересовать последняя закладка — **Media Platform**. Эта закладка позволяет развернуть дополнительные модули, оптимизирующие трансляцию видео. Так, в разделе **Host and Deliver Media Content** можно обнаружить пакет **IIS Media Services 3.0**. Установите этот пакет, он нам понадобится для дальнейшей работы.

После установки этого пакета, в консоли управления IIS 7 должны появиться следующие четыре модуля: **Web Playlists**, **Bit Rate Throttling**, **Smooth Streaming Presentations**, **Live Smooth Streaming Publishing Points**.

Тут первые два модуля связаны с прогрессивной загрузкой, а вторые — с потоковым видео. Ниже мы рассмотрим все четыре модуля.

Еще одной особенностью **Web Platform Installer** есть то, что утилита интегрируется отдельным модулем в консоль управления IIS 7, который доступен как на уровне Web-сервера, так и на уровне отдельного Web-приложения. Это позволяет разворачивать дополнительные компоненты и приложения владельцу сайта, не привлекая администратора системы.

Перейдем к описанию установленных модулей.

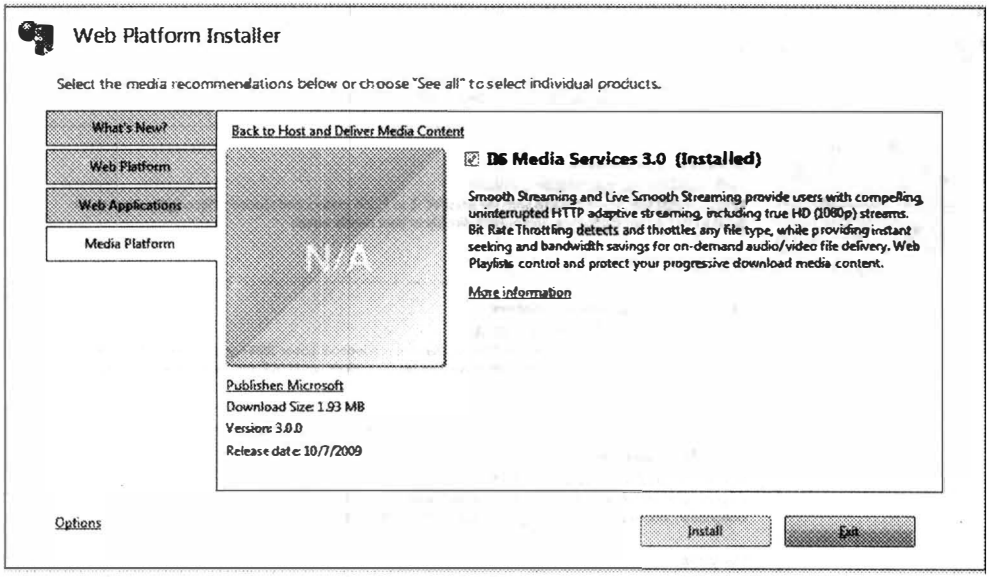


Рис. 9.8. Установка IIS Media Services 3.0

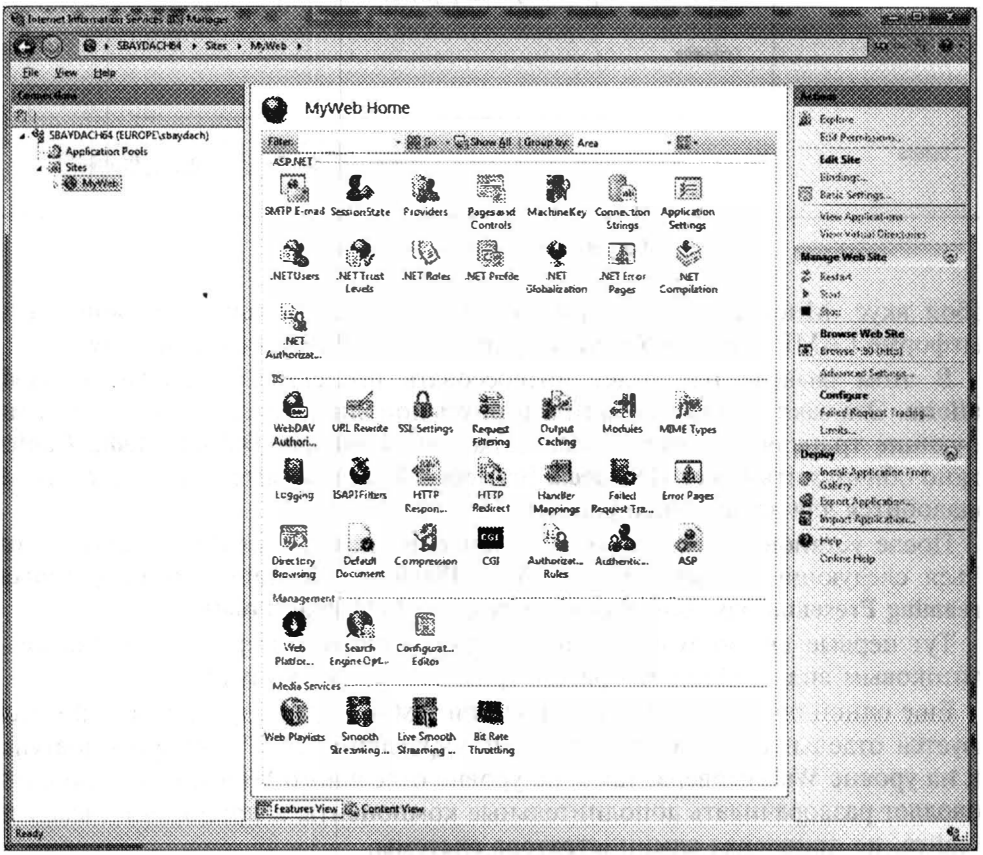


Рис. 9.9. Модули IIS 7

Создание списков

Начнем рассмотрение модулей IIS 7 с **Web Playlists**. Задача этого модуля состоит в объединении нескольких видеофрагментов для отображения их как единого целого. При этом каждый компонент списка можно настраивать, запрещая «прокрутку» конкретного видеофрагмента (вперед, назад или любую прокрутку). Иными словами, если перед трансляцией основного видео Вы хотите разместить рекламу, то **Web Playlists** — это то, что нужно. Достаточно создать новый список, который будет включать рекламный ролик (без возможности прокрутки), а также основной ролик (тут можно дать пользователю перемещаться в любой момент видео).

Вот как выглядит консоль, позволяющая создавать списки:

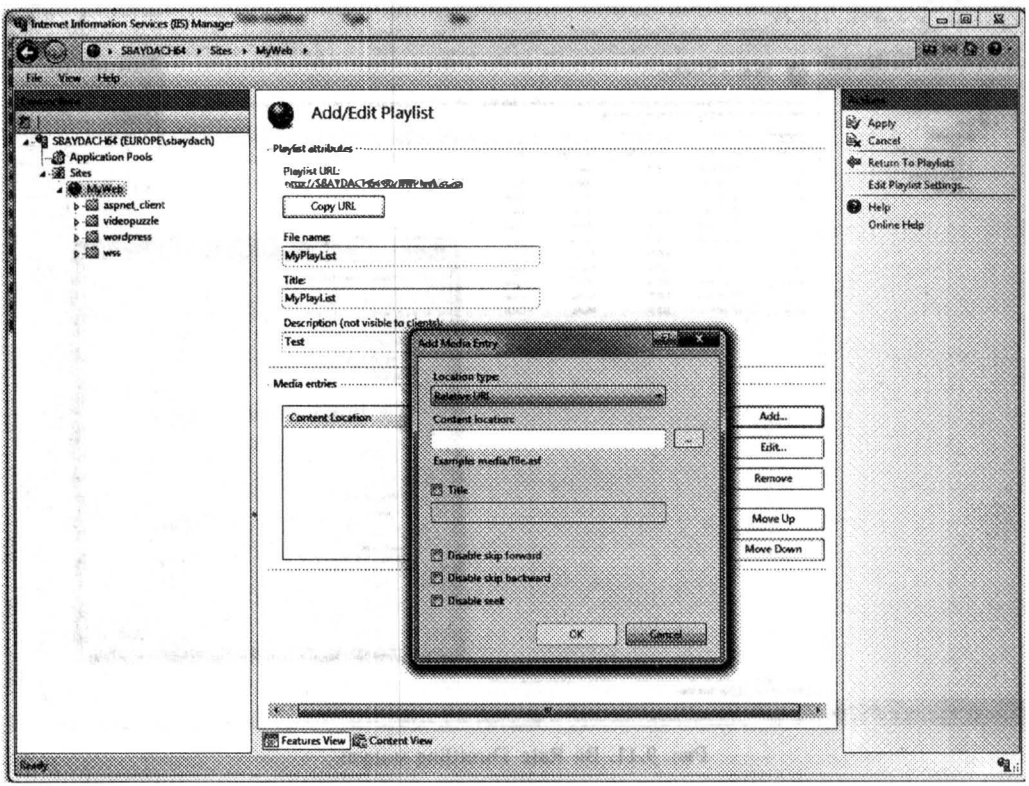


Рис. 9.10. Web Playlists модуль

Чтобы отобразить видео в заданном порядке, **MediaElement** должен принимать не ссылку на видеофайл, а ссылку на файл списка.

Возможности Bit Rate Throttling

К сожалению, размещая видео на странице, мы не знаем, будет ли пользователь смотреть это видео, а если и будет, то будет ли смотреть полностью Между тем, полная загрузка видео на сторону клиента не выгодна ни пользователю, ни провайдеру видео. Ведь и у пользователя, и у провайдера каналь

связи ограничены. По этой причине IIS 7 обеспечивает администраторов IIS 7 модулем, который позволяет загрузить лишь фрагмент видео, инициируя остальную загрузку лишь в случае необходимости. Этот модуль имеет название **Bit Rate Throttling**.

Идея работы **Bit Rate Throttling** состоит в том, чтобы для указанного типа файла (задается расширение), загрузить лишь определенное количество секунд видео с заданным качеством (можно обеспечить и потерю фреймов). Но если пользователь начинает просмотр видео, то модуль «отдает» остальную порцию. Такой подход позволяет снизить нагрузки на каналы и использовать их более эффективно.

Вот как выглядит консоль нашего модуля:

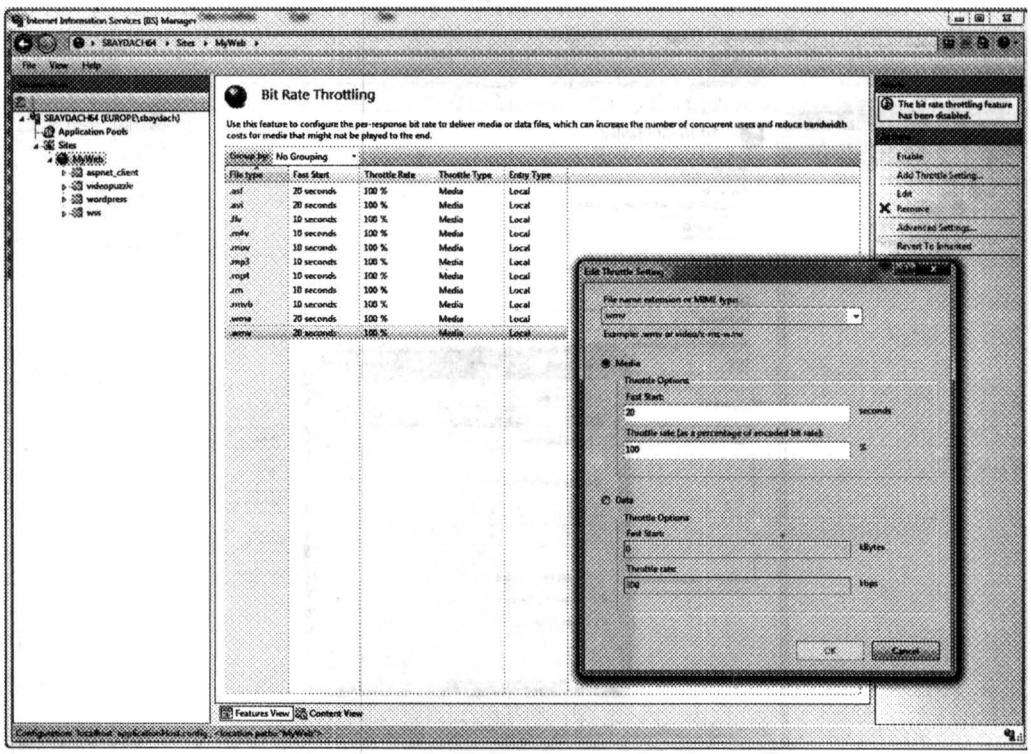


Рис. 9.11. Bit Rate Throttling модуль

С помощью этой консоли администратор может задать тип файла и размер первого пакета.

Кстати, данный модуль применим ко всем типам файлов, а не только к медиа файлам.

Использование Smooth Streaming

Путешествуя по сети, очень часто можно столкнуться с отображением видео на различных сайтах: пресс-конференции, трансляции футбольных матчей, материалы прощедших событий. Однако если проанализировать качество

материала, представленного на сайтах, то можно сделать вывод, что владельцы сайтов стремятся покрыть максимальное количество аудитории, предоставляя видео в плохом качестве (обычно рассчитывая на каналы 256 кбит). Крайне редко можно встретить возможность переключаться между видео в плохом качестве и видео с лучшим разрешением. Таким образом, если у пользователя есть широкий канал и мощный компьютер, он все равно будет смотреть видео только с плохим разрешением.

Технология **Smooth Streaming** позволяет решить проблему с трансляцией видео в хорошем качестве. При этом пользователи с узким каналом или не очень мощным компьютером могут продолжать принимать видео с плохим качеством, а пользователи с более широкими каналами смогут принимать видео в том качестве, которое будет приемлемо для их канала и мощности компьютера.

Идея **Smooth Streaming** состоит в преобразовании видео в специальный формат (фактически это набор файлов), который содержит несколько вариаций кодировки видео. Каждая из вариаций рассчитана на определенную ширину канала и определяет свой уровень качества видео. При отображении такого видео на стороне клиента, приложению передается фрагмент видео из той вариации, которая в данный момент наиболее подходит для клиента. В связи с тем, что загрузка канала клиента (как и загрузка процессора) может меняться, то передача фрагментов происходит небольшими порциями (по умолчанию 2 секунды), а настройка на оптимальное качество происходит во время всей трансляции видео и меняется в режиме реального времени.

Чтобы преобразовать видео в формат **Smooth Streaming** необходимо воспользоваться **Expression Encoder**. Для этого импортируйте видео, которое хотите отображать в формате Smooth, а затем на вкладке **Encode** выберите **IIS Smooth Streaming** в поле **Output Format**. В результате на вкладке Video Вы увидите несколько предопределенных треков, которые будут созданы при преобразовании видео. Вы можете добавить новые треки или удалить существующие (рис. 9.12).

В результате будет создано несколько файлов, которые следует разместить в одну из директорий Web-приложения. Среди созданных файлов, главный файл имеет расширение **.ism**, а остальные содержат различные варианты кодировок для нашего видео. Именно **.ism** файл и нужно передавать **MediaElement** в качестве параметров.

Чтобы Smooth Streaming работал, на Web-сервере должен быть установлен модуль **Smooth Streaming Presentations**, об инсталляции которого мы говорили выше. Консоль этого модуля различается в зависимости от места доступа. Так, если доступ к модулю осуществлен на уровне сервера, то тут есть возможность задать настройки для модуля на уровне всего сервера (рис. 9.13).

Если доступ осуществляется на уровне каталога, содержащего видео в формате **Smooth**, то консоль модуля позволяет управлять потоком, просматривать и удалять ненужные треки (рис. 9.14).

При реализации **Smooth Streaming** мне знакома только одна проблема — это организация процесса автоматического кодирования видео при загрузке на сервер. Действительно, при создании сложного решения Вы не будете осуществлять кодирование каждого файла самостоятельно. Поэтому придется разработать свою службу со сложной логикой. Это вполне реализуемо, так как

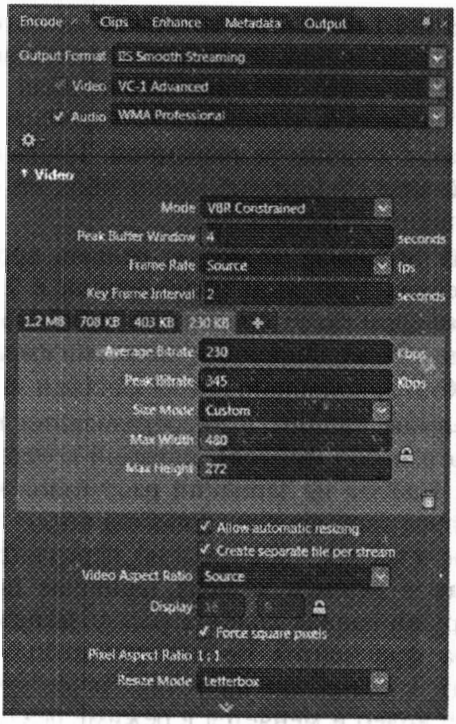


Рис. 9.12. Использование Expression Encoder

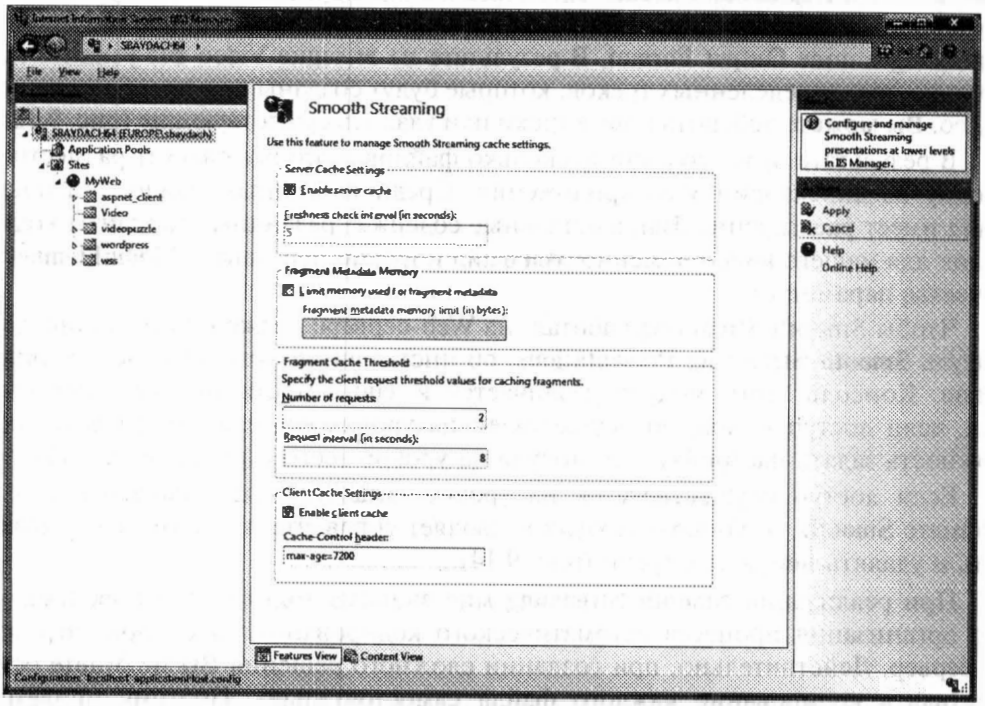


Рис. 9.13. Настройка модуля на уровне сервера

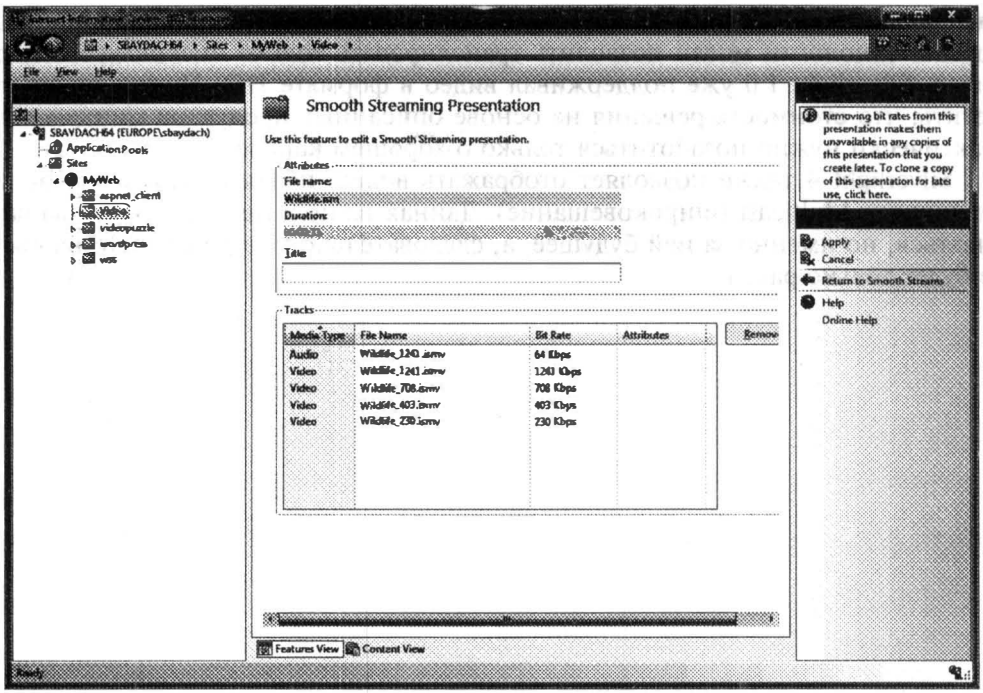


Рис. 9.14. Управление кодировками

Expression Encoder предоставляет SDK для разработчика, позволяя организовывать преобразования программно (без вызова графического интерфейса).

О проблемах с «живым» видео и его преобразованиях в формат Smooth я уже писал выше.

Защита видео с помощью DRM

Последнее о чем мне хотелось бы рассказать — это возможность передачи защищенного видео. Технология заключается в развертывании специального сервера **PlayReady**, который позволяет провести аутентификацию пользователя и выдать ему сертификат на декодирование видео. Отличие этого механизма от **https** состоит в скорости работы, ведь механизм рассчитан только на передачу видео, но не на прием данных. Основная проблема состоит в том, что технология платная и обойдется в несколько тысяч долларов. Поэтому остановиться на DRM и PlayReady у меня возможности нет. Но если передача защищенного видео остро стоит перед Вами, то можете обратиться к документации сервера PlayReady по следующей ссылке: http://go.microsoft.com/fwlink/?LinkID=125124.

Заключение

На текущий момент технология Silverlight, совместно с Internet Information Services, Expression Encoder и Media Services, представляет наиболее полный пакет средств для трансляции видео любой сложности. Именно

работа с видео и обеспечила Silverlight большую популярность. В то время как другие технологии могли позволить транслировать только низкокачественное видео, Silverlight 1.0 уже поддерживал видео в формате HD. Нужно также отметить, что стоимость решения на основе описанных продуктов минимальна. Фактически нужно позаботиться только о хорошем канале.

Silverlight 4 также позволяет отображать видео, передаваемое с помощью технологии Multicast (широковещание). Данная технология только начала развиваться, но именно за ней будущее, а, следовательно, доля Silverlight на рынке будет только расти.

Глава 10

РЕСУРСЫ И СТИЛИ

Ресурсы

В Silverlight ресурсы можно разбить на два типа:

- **Ресурсы приложения** — это стандартный набор ресурсов, который содержит изображения, иконки, текстовые файлы и другие внешние, по отношению к приложению, объекты. Использование подобного рода ресурсов позволяет упаковать все необходимые компоненты в сборку приложения или в отдельную сборку, что не будет требовать от разработчика действий по проверки наличия тех или иных файлов;
- **Ресурсы объектов** — это специальный тип ресурсов, характерный для Silverlight- и WPF-приложений. Данный тип ресурсов связан с конкретным объектом в дереве элементов Silverlight или с объектом приложения. Фактически, ресурсы объектов, — это не что иное, как возможность заполнить специальное свойство **Resources** у объектов, порожденных от **FrameworkElement** или **Application**. При этом свойство **Resources** ссылается на объект типа **ResourceDictionary**, содержащий словарь ресурсов в формате <ключ, значение>;

Ресурсы приложения

Если говорить о ресурсах приложения, то чтобы добавить некоторый файл к подобному набору ресурсов, достаточно добавить файл к проекту, перейти к свойствам файла и установить свойство **Build Action** в значение **Resource**.

При компиляции приложения, все ресурсы компилируются (фактически, просто складируются) в отдельный файл **<имя сборки приложения.g.resources>**, а затем добавляются в общую с приложением сборку.

Как ни странно, но стандартная утилита **ildasm.exe** не способна отобразить ресурсы приложения. Поэтому я скачал ознакомительную версию платного продукта Red Gate's .NET Reflector. Этот продукт способен работать с любыми типами сборок, включая сборки Silverlight 4. Скачать этот продукт можно по ссылке http://www.red-gate.com/products/reflector/. Давайте посмотрим результат работы этого продукта на одном из приложений, содержащих видео и изображение.

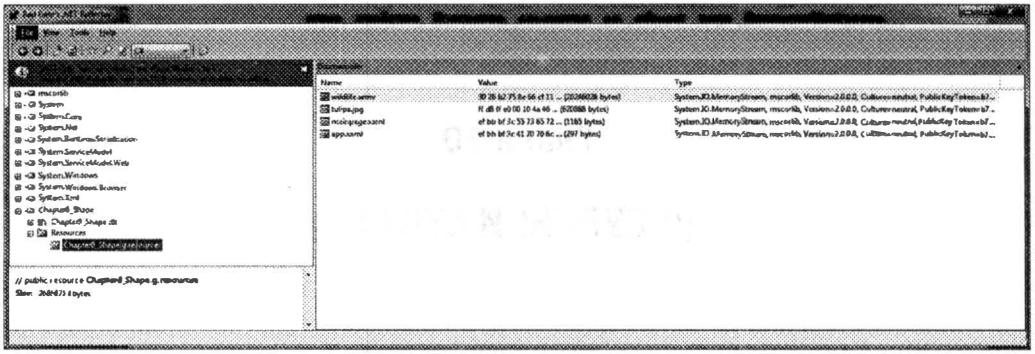


Рис. 10.1. Просмотр ресурсов внутри сборки

Обратите внимание, что в ресурсах содержатся и **.xaml** файлы нашего приложения. Это еще раз подтверждает то, что XAML подгружается в момент выполнения приложения.

Теоретически, чтобы извлечь ресурсы из сборки, можно воспользоваться статическим методом **GetResourceStream** класса **Application**:

```
StreamResourceInfo res =
   Application.GetResourceStream(
      new Uri("Wildlife.wmv",UriKind.Relative));
```

Метод **GetResourceStream** позволяет вернуть объект типа **StreamResourceInfo**, который содержит информацию о типе ресурса, а также возвращает ссылку на поток типа **UnmanagedMemoryStream**, который можно использовать для считывания информации из ресурса.

На практике большинство объектов, используемых в Silverlight, замечательно умеет работать с ресурсами самостоятельно. Например, таким элементам как **Image** и **MediaElement** достаточно указать имя файла, который находится в ресурсах. Вся остальная работа будет сделана за Вас.

В заключении отметим, что если ресурсы находятся во внешней сборке, то к ним все равно достаточно легко получить доступ, задав обычный **Uri**. Просто в этом случае формат **Uri** более громоздкий:

```
new Uri("pack://application:,,,/MyLibrary;component/Wildlife.wmv",
   UriKind.Relative)
```

В данном случае мы получаем доступ к ресурсу, который находится в сборке MyLibrary. Данный Uri можно записать и проще:

```
new Uri("MyLibrary;component/Wildlife.wmv",
   UriKind.Relative)
```

Ресурсы объектов

Перейдем к использованию ресурсов объектов. Как мы уже отметили выше, каждый элемент включает специальное свойство **Resources**, способное содержать словарь ресурсов. Данный словарь может быть доступен как из кода. так и из XAML.

Если мы используем ресурсы в XAML (а именно такой подход наиболее эффективен), то доступ к ресурсу может быть осуществлён лишь по ключу, вне зависимости от того, в каком из родительских элементов был объявлен ресурс. Если же ресурс был объявлен в одном из дочерних элементов, то он не может быть использован родительскими элементами.

Рассмотрим небольшой пример:

```
<UserControl x:Class="Chapter10_Style.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <UserControl.Resources>
        <LinearGradientBrush x:Key="myBrush">
            <GradientStop Color="Red" Offset="0"></GradientStop>
            <GradientStop Color="Green" Offset="1"></GradientStop>
        </LinearGradientBrush>
    </UserControl.Resources>

    <StackPanel x:Name="LayoutRoot" Background="White">
        <Button Width="100" Height="50"
           Background="{StaticResource myBrush}"
           Content="Button 1" Margin="5">
        </Button>
        <Button Width="100" Height="50"
           Background="{StaticResource myBrush}"
           Content="Button 2" Margin="5">
        </Button>
    </StackPanel>
</UserControl>
```

Тут мы определили в качестве ресурса градиентную кисть, указав в качестве ключа (он же и имя) myBrush. Данный ключ будет использоваться для доступа к ресурсу. Как Вы смогли заметить, доступ к ресурсу определяется с помощью расширения разметки **StaticResource**. Тут достаточно указать имя ключа в качестве параметра.

В отличие от WPF, в Silverlight не существует динамических ресурсов (**DynamicResource**).

В примере выше мы разместили ресурсы внутрь элемента **UserControl**, который является контейнером для всего нашего интерфейса. Это обеспечивает видимость ресурсов для любого дочернего элемента. Если мы хотим ограничить область видимости, то ресурсы можно разместить в любом из дочерних элементов, например в **StackPanel**:

```
<UserControl x:Class="Chapter10_Style.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <StackPanel x:Name="LayoutRoot" Background="White">
        <StackPanel.Resources>
            <LinearGradientBrush x:Key="myBrush">
                <GradientStop Color="Red"
Offset="0"></GradientStop>
```

```
                <GradientStop Color="Green"
Offset="1"></GradientStop>
            </LinearGradientBrush>
        </StackPanel.Resources>
        <Button Width="100" Height="50"
           Background="{StaticResource myBrush}"
           Content="Button 1" Margin="5">
        </Button>
        <Button Width="100" Height="50"
           Background="{StaticResource myBrush}"
           Content="Button 2" Margin="5">
        </Button>
    </StackPanel>
</UserControl>
```

Как Вы можете заметить, для конечного элемента управления нет никакой разницы, где находятся ресурсы. Главное, чтобы они находились в области видимости элемента, а доступ к ним осуществляется по одинаковой схеме.

Наконец, если Вы хотите обеспечить ресурсам глобальную область видимости, то можете разместить их на уровне всего приложения. Такой подход очень эффективен при разработке сложных приложений, то есть приложений, обладающих несколькими окнами и механизмом навигации между ними.

Вот пример определения этих же ресурсов внутри файла приложения:

```
<Application
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   x:Class="Chapter10_Style.App">
    <Application.Resources>
        <LinearGradientBrush x:Key="myBrush">
            <GradientStop Color="Red" Offset="0"></GradientStop>
            <GradientStop Color="Green" Offset="1"></GradientStop>
        </LinearGradientBrush>
    </Application.Resources>
</Application>
```

Выделение ресурсов объектов в отдельные файлы

Начиная с Silverlight 3, ресурсы возможно хранить в отдельных файлах и собирать в нужном месте с помощью элемента **ResourceDictionary**. Вынесем кисть из предыдущего примера в отдельный файл RD1.xaml:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <LinearGradientBrush x:Key="myBrush">
        <GradientStop Color="Red" Offset="0"></GradientStop>
        <GradientStop Color="Green" Offset="1"></GradientStop>
    </LinearGradientBrush>
</ResourceDictionary>
```

Для подключения внешнего файла к интерфейсу приложения используется все тот же **ResourceDictionary**:

```
<Application
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   x:Class="Chapter10_Style.App">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="RD1.xaml" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

Если у вас несколько файлов — не проблема. Перечислите их все.

Если Вы хотите подключить файл с ресурсами в ресурсы одного из элементов, то это делается аналогично.

Стили

Понятие стилей

Очень тесно с ресурсами связаны такие элементы как стили. Стиль — это специальный механизм, который позволяет собрать в одном месте все общие свойства какой-то определенной группы элементов управления. Очевидно, что если Вы создаете интерфейс, то все Ваши кнопки, поля редактирования, надписи и т. д. выполнены в едином стиле. Вряд ли кнопки в Вашем приложении раскрашены в разные цвета, а надписи имеют разный шрифт.

Рассмотрим предыдущий пример, установив кнопкам дополнительные свойства:

```
<UserControl x:Class="Chapter10_Style.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <StackPanel x:Name="LayoutRoot" Background="White">
        <Button Width="100" Height="50"
           Background="{StaticResource myBrush}"
           Content="Button 1" Margin="5" FontFamily="Arial"
           FontSize="12" FontWeight="Bold"
           Foreground="Blue" BorderThickness="3">
        </Button>

        <Button Width="100" Height="50"
           Background="{StaticResource myBrush}"
           Content="Button 2" Margin="5" FontFamily="Arial"
           FontSize="12" FontWeight="Bold"
           Foreground="Blue" BorderThickness="3">
        </Button>
```

```
        <Button Width="100" Height="50"
           Background="{StaticResource myBrush}"
           Content="Button 3" Margin="5" FontFamily="Arial"
           FontSize="12" FontWeight="Bold"
           Foreground="Blue" BorderThickness="3">
        </Button>

        <Button Width="100" Height="50"
           Background="{StaticResource myBrush}"
           Content="Button 4" Margin="5" FontFamily="Arial"
           FontSize="12" FontWeight="Bold"
           Foreground="Blue" BorderThickness="3">
        </Button>

    </StackPanel>
</UserControl>
```

Как Вы видите, создав всего четыре кнопки, мы продублировали множество строк кода. Это не только значительно увеличивает размер файла, но и ведет к возникновению ошибок. Ведь указав один атрибут неверно, можно испортить внешний вид всего интерфейса.

Чтобы выделить общие свойства в отдельный стиль, в Silverlight используется специальный элемент **Style**. Поскольку этот элемент создается для использования в различных элементах управления, его место в ресурсах объектов.

Продемонстрируем работу элемента **Style** для нашего примера:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <LinearGradientBrush x:Key="myBrush">
        <GradientStop Color="Red" Offset="0"></GradientStop>
        <GradientStop Color="Green" Offset="l"></GradientStop>
    </LinearGradientBrush>

    <Style x:Key="buttonStyle" TargetType="Button">
        <Setter Property="Background"
           Value="{StaticResource myBrush}"></Setter>
        <Setter Property="Margin" Value="5"></Setter>
        <Setter Property="FontFamily" Value="Arial"></Setter>
        <Setter Property="FontSize" Value="12"></Setter>
        <Setter Property="FontWeight" Value="Bold"></Setter>
        <Setter Property="Foreground" Value="Blue"></Setter>
        <Setter Property="BorderThickness" Value="3"></Setter>
    </Style>
</ResourceDictionary>
```

Как Вы видите, чтобы задать стиль, необходимо указать ключ и тип элементов, к которым может быть применим стиль. Далее, с помощью набора элементов **Setter** мы перечислили все свойства и их значения. Обратите вни-

мание, что сюда мы включили и свойство **Background**, сославшись на ранее созданную кисть. Код XAML нашего интерфейса будет выглядеть следующим образом:

```
<UserControl x:Class="Chapter10_Style.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <StackPanel x:Name="LayoutRoot" Background="White">
        <Button Width="100" Height="50"
           Style="{StaticResource buttonStyle}" Content="Button 1">
        </Button>

        <Button Width="100" Height="50"
           Style="{StaticResource buttonStyle}" Content="Button 2">
        </Button>

        <Button Width="100" Height="50"
           Style="{StaticResource buttonStyle}" Content="Button 3">
        </Button>

        <Button Width="100" Height="50"
           Style="{StaticResource buttonStyle}" Content="Button 4">
        </Button>
    </StackPanel>
</UserControl>
```

Если же использовать возможности Silverlight 4, то можно определить стиль по умолчанию, и совсем не указывать атрибут **Style** для наших кнопок.

Чтобы создать стиль по умолчанию, достаточно удалить атрибут **Key** из определения стиля. А в описании интерфейса атрибут **Style** удалить вовсе.

```
<UserControl x:Class="Chapter10_Style.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <StackPanel x:Name="LayoutRoot" Background="White">
        <Button Width="100" Height="50" Content="Button 1">
</Button>
        <Button Width="100" Height="50" Content="Button 1">
</Button>
        <Button Width="100" Height="50" Content="Button 1">
</Button>
        <Button Width="100" Height="50" Content="Button 1">
</Button>
    </StackPanel>
</UserControl>
```

Замечание. Почему-то стили по умолчанию не поддерживаются дизайнером Visual Studio. Поэтому Вы увидите элементы без установленного стиля. Но после запуска приложения все будет в порядке.

Динамическая установка стилей

Во второй версии Silverlight свойство **Style** можно было установить только один раз. При попытке изменить стиль, генерировалось исключение. При такой реализации можно и не думать об изменении обверток для интерфейса или о персональном интерфейсе для каждого пользователя.

Начиная с третьей версии SilverLight, поддерживается многократная установку стилей.

Рассмотрим простой пример, где определим два стиля и возможность выбора одного из них в процессе работы приложения:

```
<UserControl x:Class="SilverlightApplication59.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <UserControl.Resources>
        <Style x:Name="btnStyleRus" TargetType="Button">
            <Setter Property="Foreground" Value="Blue"></Setter>
            <Setter Property="Width" Value="100"></Setter>
            <Setter Property="Content" Value="Кнопка"></Setter>
        </Style>
        <Style x:Name="btnStyle" TargetType="Button">
            <Setter Property="Foreground" Value="Green"></Setter>
            <Setter Property="Width" Value="100"></Setter>
            <Setter Property="Content" Value="Button"></Setter>
        </Style>
    </UserControl.Resources>
    <StackPanel x:Name="LayoutRoot" Background="White">
        <Button x:Name="myButton"></Button>
        <ComboBox Width="100" x:Name="cmb">
                <ComboBoxItem Content="English"></ComboBoxItem>
                <ComboBoxItem Content="Russian"></ComboBoxItem>
        </ComboBox>
    </StackPanel>
</UserControl>
```

А вот и код приложения:

```
public MainPage()
{
    InitializeComponent();
    cmb.SelectionChanged += ComboBox_SelectionChanged;
    cmb.SelectedIndex = 0;
}
private void ComboBox_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    switch (((ComboBox)sender).SelectedIndex)
    {
        case 0:
            myButton.Style = btnStyle;
            break;
```

```
            case 1:
                myButton.Style = btnStyleRus;
                break;
        }
    }
```

В результате мы получили интерфейс, который при выборе языка из выпадающего списка позволяет менять надпись на кнопке и ее цвет.

BasedOn стили

Как и WPF, Silverlight поддерживает BasedOn-стили. Это позволяет задать общие стили применительно к базовым классам для общей группы элементов управления (или для конкретного элемента управления) и определить дополнительные стили путем наследования. Ниже пример BasedOn-стиля:

```
<Style x:Key="baseStyle" TargetType="Button" >
    <Setter Property="Width" Value="100"></Setter>
</Style>

<Style x:Key="btnStyleRus" BasedOn="{StaticResource baseStyle}"
TargetType="Button">
    <Setter Property="Foreground" Value="Blue"></Setter>
    <Setter Property="Content" Value="Кнопка"></Setter>
</Style>
<Style x:Key="btnStyle" BasedOn="{StaticResource baseStyle}"
TargetType="Button">
    <Setter Property="Foreground" Value="Green"></Setter>
    <Setter Property="Content" Value="Button"></Setter>
</Style>
```

Заключение

Ресурсы представляют собой мощный механизм, позволяющий не только работать с бинарными данными различного типа, но и выделять различные элементы интерфейса (файлы с изображениями, стили, надписи) для последующей локализации. Благодаря возможности выделения ресурсов в отдельные сборки, их можно однозначно разделить на группы, характерные для определенного языка и страны. Такой подход позволяет упростить работу по локализации, сосредоточившись только на работе с ресурсами без перекомпиляции основного приложения. Кроме того, ресурсы предоставляют единственный механизм, позволяющий хранить объекты и отдельные свойства для многократного использования во всем коде. Так, использование стилей позволяет реализовать интерфейс приложения в определенной цветовой гамме и с одинаковым внешним видом всех элементов. При этом стиль достаточно описать в одном месте и просто применить для каждого из элементов, а также можно применять новую возможность Silverlight 4 создавать стили по умолчанию.

Глава 11

СОЗДАНИЕ ШАБЛОНОВ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ

Понятие шаблона

Создавая приложения для Windows с использованием Win API или Windows Forms, разработчик всегда сталкивался с одинаковым набором элементов, которые можно было модифицировать, изменив цвет фона или шрифт текста, отображаемого в элементе. Если разработчик хотел получить новое представление уже знакомого элемента (например, круглую кнопку), то приходилось разрабатывать собственный элемент управления с нуля. Многие компании выпускали целые библиотеки стилизованных элементов управления, которые позволяли сделать Ваш интерфейс отличающимся от стандартных «серых» приложений.

В WPF, а впоследствии и в Silverlight, ситуация со стилизацией элементов управления изменилась кардинальным образом. Так, большинство элементов разработано таким образом, чтобы отделить логику работы элемента от его представления. При этом представление элемента всегда можно изменить по Вашему желанию. Например, логика кнопки совершенно не связана с ее прямоугольным представлением.

Подобное разделение стало возможно благодаря использованию XAML в Silverlight. XAML представляет собой настолько мощный механизм, что позволяет описывать не только внешнее представление элемента, с помощью контейнеров и различных примитивов, но и активировать анимацию. Фактически все, что должен делать код, — это управлять свойствами элемента управления и перехватывать события. При этом свойства элемента управления могут иметь привязку к любому из элементов в его представлении, а реакция на события позволяет коду изменять состояние элемента, что дает возможность осуществлять переход к другому визуальному представлению элемента в XAML (кнопка нажата -> кнопка отпущена).

И, наконец, поскольку представление элемента отделено от логики, то логично было бы иметь механизм, который позволил бы поменять данное представление. В Silverlight такой механизм есть. Так, визуальное представление элемента управления можно задать, используя простое свойство **Template**. Тут содержится объект типа **ControlTemplate**, который задает представление по умолчанию. но изменить его можно в любой момент.

Разбор шаблона для элемента Button

Составляющие элемента управления

Чтобы разобраться со структурой шаблона элемента управления, воспользуемся Expression Blend 4. Дело в том, что эта утилита умеет копировать встроенные в элемент шаблоны, позволяя редактировать их в виде ресурсов и использовать для всех аналогичных элементов управления.

Запустим Expression Blend 4 и поместим простую кнопку на рабочую поверхность и инициируем команду контекстного меню **Edit Template->Edit a Copy**:

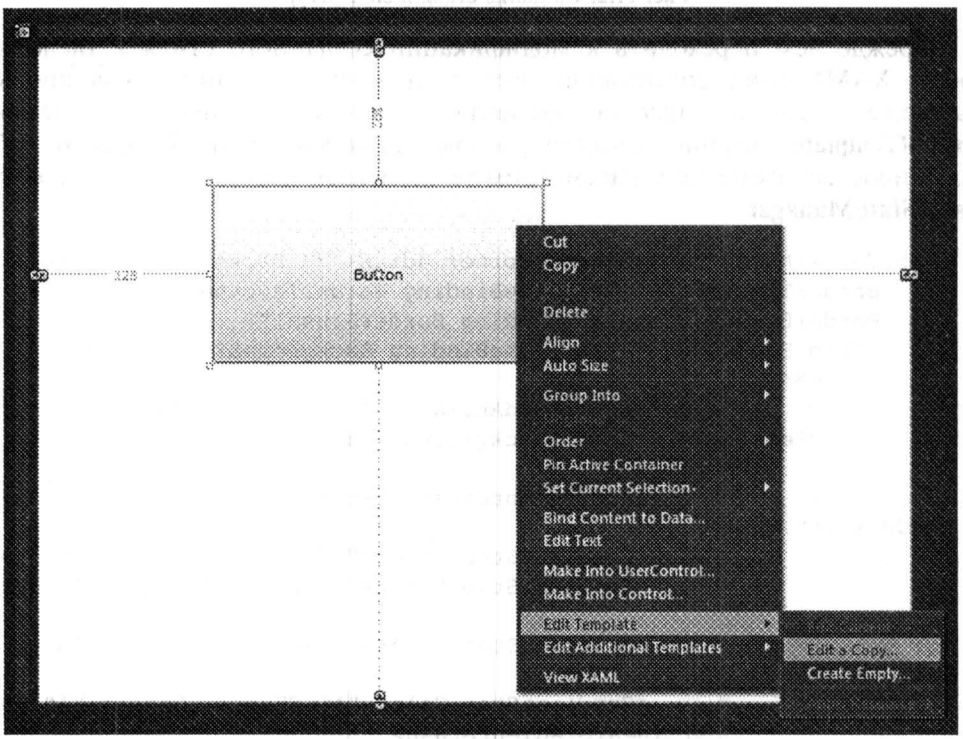


Рис. 11.1. Создание шаблона в Expression Blend

Эта команда позволяет извлечь предопределенный шаблон в ресурсы приложения и перейти в режим редактирования копии шаблона для создания новой обвертки элемента управления. Обращаю Ваше внимание, что Blend встраивает шаблон элемента внутри стиля, задавая свойство **Template**. Между тем шаблон элемента может находиться вне стиля и привязываться напрямую в свойство **Template** конкретного элемента. Естественно, что стили более удобны, так как позволяют применить новый шаблон ко всем однотипным элементам в интерфейсе без дополнительного кода. При этом стиль позволяет задать дополнительные свойства элемента.

Итак, выберите тип ресурса, куда Вы хотите разместить шаблон, и нажмите OK (рис. 11.2).

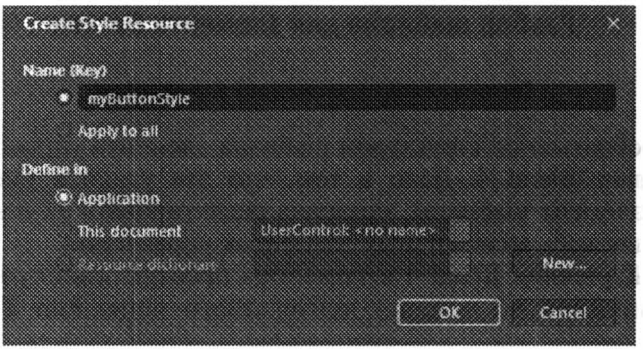


Рис. 11.2. Создание стиля в виде ресурса

Прежде чем переходить к модификации созданного стиля в Blend, откройте XAML файл, содержащий этот стиль в текстовом представлении. Как Вы можете заметить, шаблон элемента управления начинается с элемента **ControlTemplate**, внутри которого располагается **VisualStateManager** и набор более простых элементов. Рассмотрим те элементы, которые следуют сразу за **VisualStateManager**:

```
<Border x:Name="Background" CornerRadius="3" Background="White"
    BorderThickness="{TemplateBinding BorderThickness}"
    BorderBrush="{TemplateBinding BorderBrush}">
    <Grid Background="{TemplateBinding Background}"  Margin="1">
        <Border Opacity="0"
            x:Name="BackgroundAnimation" Background="#FF448DCA" />
        <Rectangle x:Name="BackgroundGradient" >
            <Rectangle.Fill>
                <LinearGradientBrush StartPoint=".7,0"
EndPoint=".7,1">
                    <GradientStop Color="#FFFFFFFF" Offset="0" />
                    <GradientStop Color="#F9FFFFFF" Offset="0.375"
/>
                    <GradientStop Color="#E5FFFFFF" Offset="0.625"
/>
                    <GradientStop Color="#C6FFFFFF" Offset="1" />
                </LinearGradientBrush>
            </Rectangle.Fill>
        </Rectangle>
    </Grid>
</Border>
<ContentPresenter
        x:Name="contentPresenter"
        Content="{TemplateBinding Content}"
        ContentTemplate="{TemplateBinding ContentTemplate}"
        VerticalAlignment="{TemplateBinding
VerticalContentAlignment}"
        HorizontalAlignment="{TemplateBinding
HorizontalContentAlignment}"
        Margin="{TemplateBinding Padding}"/>
<Rectangle x:Name="DisabledVisualElement" RadiusX="3" RadiusY="3"
```

```
    Fill="#FFFFFFFF" Opacity="0" IsHitTestVisible="false" />
<Rectangle x:Name="FocusVisualElement" RadiusX="2" RadiusY="2"
Margin="1"
    Stroke="#FF6DBDD1" StrokeThickness="1" Opacity="0"
    IsHitTestVisible="false" />
```

Весь этот набор элементов и представляет собой визуальное описание кнопки. Как Вы видите, основными составляющими кнопки являются прямоугольники, которые используются для внешней окантовки, а также для выделения кнопки в фокусе и неактивном состоянии. Кроме прямоугольников тут присутствует элемент **ContentPresenter**, который способен содержать любой контент. Если такой элемент присутствует, то свойство Content элемента автоматически выполняет привязку к нему. Но в данном случае этот элемент не является обязательным (как и остальные).

Если изучить кнопку более детально, то можно обнаружить, что логика элемента никоим образом не связана с представлением. Любой элемент можно удалить или заменить другим. Однако, это верно не для всех элементов. Так, многие элементы требуют наличия определенного набора элементов управления с заданными именами и соответствующих типов. Как правило, это более сложные элементы (например, **ComboBox**), которые составляются из не просто из прямоугольников. В этом случае шаблоны могут быть вложенными, опускаясь на уровень ниже и задавая свое представление для более мелких составляющих.

Элементы управления, обязательные для создания нового шаблона называются Составляющими (**Parts**). Изучая XAML, Вы не сможете найти ни одного атрибута, которые описывают требование к наличию того или иного элемента в шаблоне. Эти требования описываются непосредственно в коде с помощью атрибутов класса **TemplatePart**:

```
[TemplatePart(Name = "TextElement", Type = typeof(TextBlock))]
[TemplatePart(Name = "UpButtonElement", Type =
typeof(RepeatButton))]
[TemplatePart(Name = "DownButtonElement", Type =
typeof(RepeatButton))]
public class MyControl: Control
```

Состояния и переходы

Итак, разобравшись с **Составляющими** элемента управления, углубимся в анализ поведения кнопки. Дело в том, что кнопка меняет свое внешнее представление в зависимости от поведения пользователя и движения мыши. Чтобы определить поведение кнопки в каждом из ее состояний, необходимо:

- Описать возможные состояния;
- Модифицировать текущее состояние внутри логики элемента управления;
- Описать изменения для каждого из состояний в XAML;
- Определить дополнительные характеристики перехода из одного состояния в другое.

Итак, чтобы описать возможные состояния внутри логики элемента управления используется атрибут класса **TemplateVisualState**. Для описания элемента управления Button используется целых шесть подобных атрибутов:

```
[TemplateVisualState(Name = "Normal", GroupName = "CommonStates")]
[TemplateVisualState(Name = "MouseOver", GroupName =
"CommonStates")]
[TemplateVisualState(Name = "Pressed", GroupName = "CommonStates")]
[TemplateVisualState(Name = "Disabled", GroupName =
"CommonStates")]
[TemplateVisualState(Name = "Unfocused", GroupName =
"FocusStates")]
[TemplateVisualState(Name = "Focused", GroupName = "FocusStates")]
public class Button : ButtonBase
{
}
```

Из синтаксиса несложно догадаться, что атрибут принимает два параметра: имя состояния и имя группы. Группы вводятся для того, чтобы ограничить возможность пересечения состояний. Так, кнопка не может быть нажата и отпущена одновременно. Фактически, кнопка может находиться только в одном состоянии из каждой группы.

Чтобы обеспечить переход из состояния в состояние, внутри реализации логики элемента управления достаточно воспользоваться статическим методом **GoToState** класса **VisualStateManager**:

```
private void UpdateStates(bool useTransitions)
{
    if (isFocused)
    {
        VisualStateManager.GoToState(this, "Focused",
           useTransitions);
    }
    else
    {
        VisualStateManager.GoToState(this, "Unfocused",
           useTransitions);
    }
}
```

Тут метод принимает имя свойства и задает, может ли использоваться дополнительная анимация при переходе из одного состояния в другое.

Теперь можно перейти к XAML. Тут и выполняется вся работа по изменению визуального представления элемента из одного состояния в другое. Это делается с помощью элемента **VisualStateManager**, который содержит описание всех состояний (**VisualState**), разбитых на группы (**VisualStateGroup**):

```
<vsm:VisualStateManager.VisualStateGroups>
      <vsm:VisualStateGroup x:Name="CommonStates">
          <vsm:VisualState x:Name="Normal"/>
          <vsm:VisualState x:Name="MouseOver">
              <Storyboard>
```

```
                    <DoubleAnimation Duration="0"
                       Storyboard.TargetName="BackgroundAnimation"
                       Storyboard.TargetProperty="Opacity" To="1"/>
                    <ColorAnimation Duration="0"
                       Storyboard.TargetName="BackgroundGradient"
                       Storyboard.TargetProperty=

"(Rectangle.Fill).(GradientBrush.GradientStops)[1].(GradientStop.Co
lor)"
                       To="#F2FFFFFF"/>
                    <ColorAnimation Duration="0"
                       Storyboard.TargetName="BackgroundGradient"
                       Storyboard.TargetProperty=

"(Rectangle.Fill).(GradientBrush.GradientStops)[2].(GradientStop.Co
lor)"
                       To="#CCFFFFFF"/>
                    <ColorAnimation Duration="0"
                       Storyboard.TargetName="BackgroundGradient"
                       Storyboard.TargetProperty=

"(Rectangle.Fill).(GradientBrush.GradientStops)[3].(GradientStop.Co
lor)"
                       To="#7FFFFFFF"/>
                </Storyboard>
            </vsm:VisualState>
. . . . .
```

Если посмотреть на исходный код кнопки, то все выглядит достаточно просто. В каждом из состояний запускается серия анимации, изменяющих свойства элементов управления — составляющих кнопки.

Наверное, Вы обратили внимание, что каждая анимация имеет свойство **Duration**, установленное в 0. То есть все изменения происходят мгновенно. Чтобы сделать анимацию более плавной, пользуются специальным элементом **VisualTransition**, который и позволяет задать время выполнения всего **StoryBoard**.

```
<VisualTransition From="Normal" GeneratedDuration="0:0:3"
To="MouseOver"/>
```

Таким образом, создание шаблона — задача тривиальная. Между тем, Expression Blend позволяет еще больше упростить эту задачу, предлагая специальный режим редактирования шаблона. В этом режиме разработчик получает доступ к таким окнам как **States** и **Parts**, а также ко всему дереву элемента управления. Окно Parts позволяет отобразить все обязательные элементы, что позволяет отлеживать ошибки создания шаблона и не требует изучать документацию по каждому из элементов (а какие же там обязательные составляющие). Окно **States** отображает все состояния, разбитые по группам, с возможностью создавать переходы и редактировать анимацию в каждом из состояний.

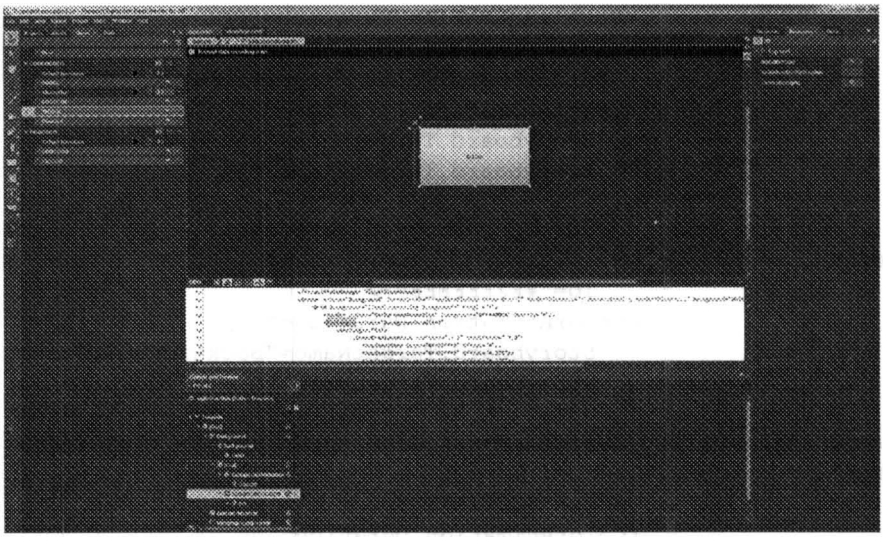


Рис. 11.3. Использование Expression Blend

Заключение

Разделение логики работы элемента и его представления впервые появилось в WPF, а затем перекочевало и в Silverlight, и является одной из основополагающих возможностей. Следует только немного перестроить свое понимание стандартных интерфейсов, возникшее еще на заре Windows-программирования, и Вы сможете получать абсолютно новые интерфейсы, приложив для их создания минимум усилий. При этом стоит прогнозировать, что работа дизайнера станет более востребованной. Так, работая в нескольких компаниях по разработке программного обеспечения, я помню только одного дизайнера. Разрабатывая стандартные «серые» интерфейсы мы не прибегали к их услугам, но теперь ситуация меняется кардинально. Дизайнер нужен как для подбора цветовой гаммы интерфейса приложения, так и для создания новых шаблонов. Чтобы убедиться в этом, вернитесь к главе 8, и, обладая всеми знаниями градиентной заливки, попробуйте создать объемную кнопку овальной формы (выпуклую вверх или приподнятую по краям и выпуклую вниз) — после этого начинайте искать дизайнера!

Глава 12

ОТЛАДКА ПРИЛОЖЕНИЙ И ТЕСТИРОВАНИЕ

Отладка с помощью Visual Studio 2010

Visual Studio 2010 позволяет отлаживать Silverlight-приложения подобно другим типам приложений. Если Вы планируете приступить к отладке Silverlight-приложения, то достаточно запустить его с помощью команды **Debug->Start Debugging** (F5). Расставляя в желаемых местах точки останова, Вы сможете наблюдать за ходом выполнения приложения, имея доступ ко всем известным окнам: **Autos, Locals, Watch**.

Однако, отладка Silverlight-приложения может потребовать отлаживать и JavaScript. Visual Studio позволяет отлаживать JavaScript без всяких проблем. Но, если Вы поставите точку останова в JavaScript во время отладки, то ничего не произойдет. Это связано с тем, что Visual Studio не позволяет отлаживать сразу два типа приложений (Silverlight и JavaScript). А Silverlight-проект настроен таким образом, что включает отладку для Silverlight по умолчанию.

Чтобы включить отладку JavaScript в Вашем приложении, в Visual Studio доступно два способа.

Способ 1. Для включения отладки JavaScript (на самом деле выключения отладки Silverlight) достаточно просто открыть свойства проекта, где на вкладке Web, снять флаг Silverlight (раздел Debuggers). Это простое действие позволит отлаживать JavaScript, но точки останова для Silverlight работать перестанут. Описанное выше действие нужно сделать для Web-приложения, которое содержит встраиваемый Silverlight-компонент (рис. 12.1).

Способ 2. Выбрать тип приложения, которое Вы планируете отлаживать, можно при подключении к процессу. Для этого запустите Silverlight-приложение НЕ в режиме отладки, а обычным способом (Start Without Debugging (Ctrl+F5)). Когда приложение откроется в браузере, выполните команду **Debug->Attach to Process...** (рис. 12.2).

В появившемся окне достаточно легко найти экземпляр Вашего браузера, выполняющего Silverlight-приложение. Нужный Вам процесс содержит значения Script и Silverlight в колонке Type. Чтобы переключатся между этими двумя типами приложения, нажмите кнопку Select. В появившемся окне выберите нужный тип (рис. 12.3).

Описанные способы можно применять не только при создании простых Web-приложений, но и при создании сложных проектов на базе SharePoint 2010

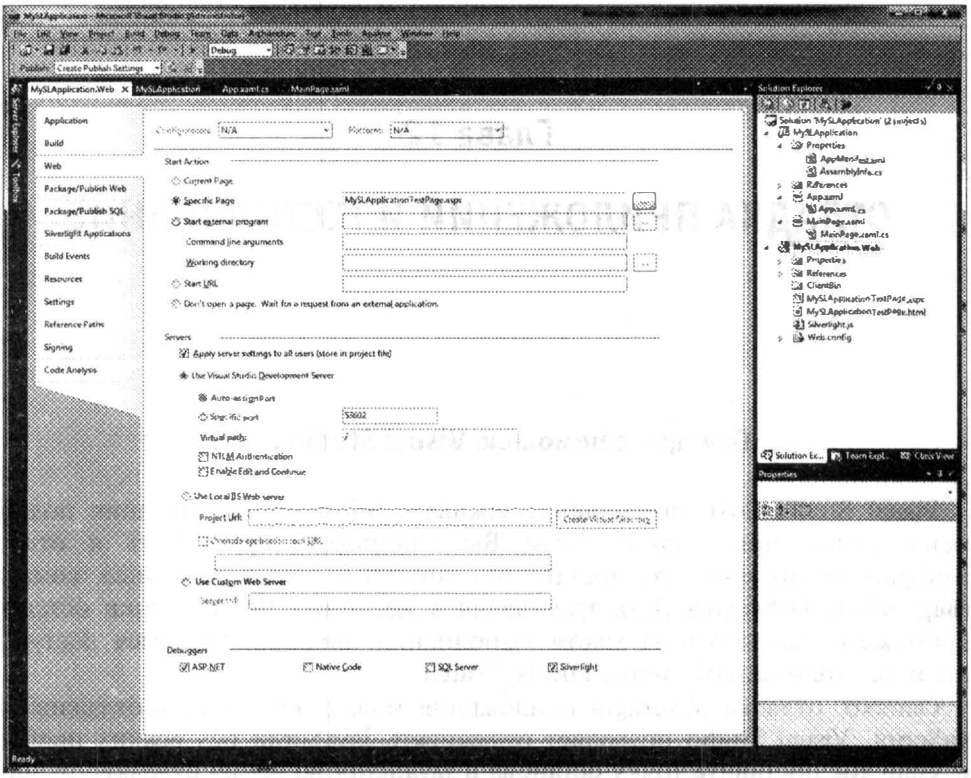


Рис. 12.1. Установка отладчика по умолчанию

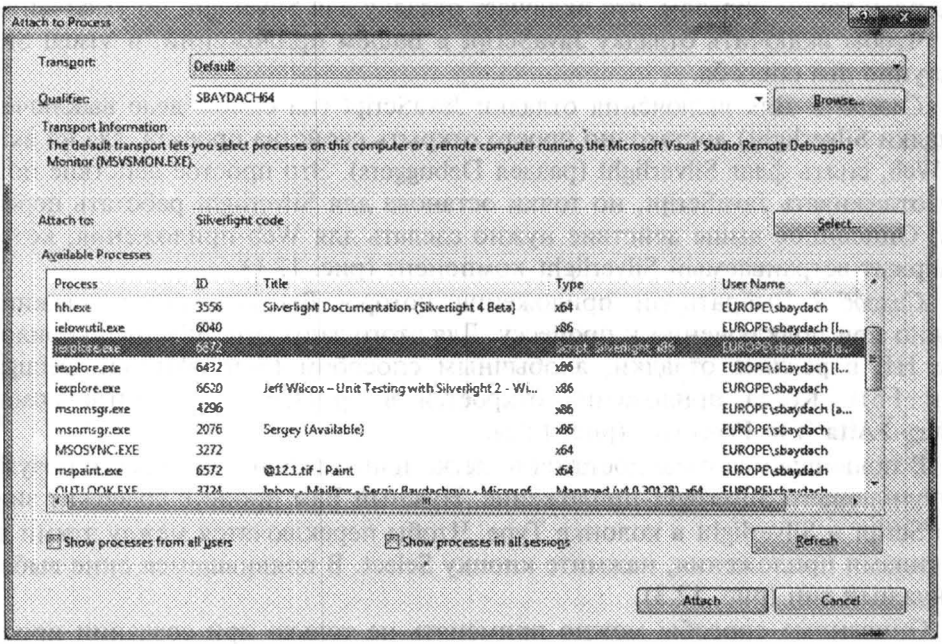


Рис. 12.2. Выбор процесса

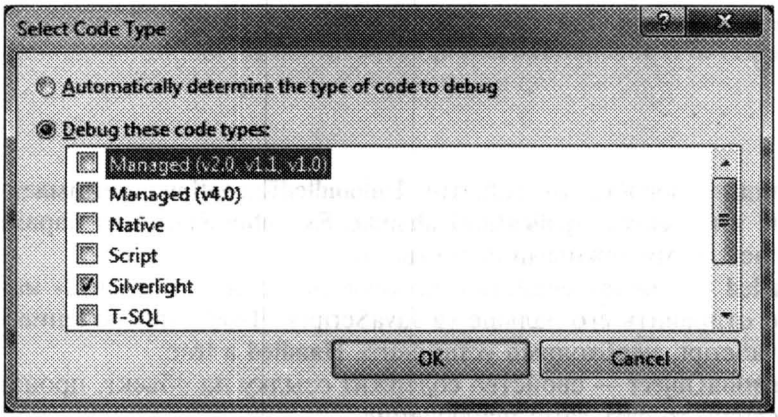


Рис. 12.3. Выбор типа отлаживаемого приложения

или другой платформы, когда ошибку может вызывать не только Silverlight-приложение, но и механизмы интеграции с хостом.

Обработка ошибок в Silverlight

Ошибки в Silverlight-приложениях можно условно разделить на два типа: ошибки, которые генерируются управляемым кодом в потоке интерфейса, и ошибки, которые генерирует ядро самого встраиваемого Silverlight-компонента. В соответствии с этим разделением используются два разных подхода при обработке ошибок. Рассмотрим оба подхода.

Обработка ошибок в управляемом коде

При обработке ошибок в управляемом коде все выглядит достаточно просто. Как и в любом другом .NET приложении, Вы просто заключаете код, генерирующий исключения, в блок **try...catch**. Единственная проблема состоит в том, что разработчик обычно останавливается на обработке наиболее явных исключений, не обрабатывая редкие или почти невозможные ситуации. А это означает, что даже если ошибка и незначительна, но не была обработана, то она будет переброшена в JavaScript, что приведет к сбою всего Silverlight-компонента. По этой причине, разработчик имеет возможность реализовать обработчик специального события **UnhandledException**, которое описано в классе **Application**. Ниже показан код, который генерируется Visual Studio при создании Silverlight-приложения:

```
public App()
{
    this.Startup += this.Application_Startup;
    this.Exit += this.Application_Exit;
    this.UnhandledException += this.Application_UnhandledException;

    InitializeComponent();
```

```
private void Application_UnhandledException(object sender,
    ApplicationUnhandledExceptionEventArgs e)
{
    . . . . .
}
```

Как видно, обработчик события **UnhandledException** принимает два параметра. Нас интересует **ApplicationUnhandledExceptionEventArgs** параметр, который обладает двумя важными свойствами:

- **Handled** — важное свойство, которое позволяет «погасить» исключение, либо отправить его дальше (в JavaScript). Чтобы исключение не попало в JavaScript, необходимо установить **Handled** в true;
- **ExceptionObject** — свойство содержит ссылку на объект, производный от **Exception**, то есть само исключение.

Таким образом, перехват всех исключений, генерируемых управляемым кодом, не представляет труда.

Обработка ошибок в JavaScript

В случае если разработчик не определил обработчик события **UnhandledException** и не перехватил исключение в коде, то ошибка передается в JavaScript. Все ошибки, генерируемые ядром Silverlight-компонента (обычно неуправляемый код), также передаются в JavaScript. При этом если в JavaScript нет никакой реакции на ошибку, то вместо Silverlight-приложения пользователь увидит серый прямоугольник. Если Вы хотите все-таки обработать ошибку, то для этого можно воспользоваться параметром управляемого компонента **onerror**. Вот как может выглядеть элемент **object** с заданным параметром:

```
<object data="data:application/x-silverlight-2,"
   type="application/x-silverlight-2" width="100%" height="100%">
  <param name="source" value="ClientBin/MySLApplication.xap"/>
  <param name="onError" value="onSilverlightError" />
  <param name="background" value="white" />
  <param name="minRuntimeVersion" value="4.0.50204.0" />
  <param name="autoUpgrade" value="true" />
</object>
```

В данном случае мы перенаправляем все ошибки в JavaScript метод onSilverlightError. Сигнатура этого метода должна выглядеть следующим образом:

```
function onSilverlightError(sender, args)
```

Тут Вы можете увидеть два параметра — это sender, который ссылается на встраиваемый Silverlight-компонент (и не имеет ничего общего с объектом, который выбросил исключение или сгенерировал ошибку), и параметр args, который представляет собой объект типа **ErrorEventArgs**, или объект производного от него класса.

Обрабатывая объект типа **ErrorEventArgs,** нужно обратить внимание на следующие свойства:

- **errorMessage** — тут содержится сообщение об ошибке, которое Вы вряд ли захотите отобразить пользователю. Хотя при разработке и тестировании приложения это сообщение может быть очень полезно;
- **errorType** — это свойство содержит тип ошибки. Несмотря на то, что это свойство способно принимать одно из девяти значений, особое внимание нужно обратить на **RuntimeError** и **ParseError**. Наличие этих значений говорит о том, что обработчик **onerror** принимает вторым параметром не **ErrorEventArgs**, а **RuntimeErrorEventArgs** и **ParseErrorEventArgs** соответственно. Следовательно, Вы сможете получить дополнительные параметры, используя свойства дочерних объектов;
- **errorCode** — это свойство содержит код ошибки, который обычно не применим для управляемого кода.

При обработке ошибок в JavaScript следует помнить о том, что если Вы создаете Silverlight-компонент, используя Silverlight.js, то указывать null в соответствующем параметре метода **createObject** противопоказано. В этом случае ошибка не будет поглощена. Вместо этого будет использован обработчик, описанный в файле Silverlight.js, который отобразит пользователю пугающее сообщение с информацией о сбое.

Асинхронный вызов методов

Отдельно стоит сказать об обработке ошибок в управляемом коде. Дело в том, что код, выполняющийся в другом потоке, «не умеет» бросать исключения в интерфейсный поток. Что самое интересное, исключения в подобных методах встречаются чаще, чем в других. Ведь асинхронный вызов методов обычно используется при доступе к серверным ресурсам. А при взаимодействии с сервером не всегда можно полагаться на формат данных и на стабильность канала связи. Это один из немногих случаев, когда возможность наличия исключений не зависит от качества Вашего кода.

Чтобы сделать возможным обработку ошибок для асинхронных вызовов, часто прибегают к реализации методов с дополнительным параметром, который содержит ссылку на метод, вызывающийся в случае ошибки. Ниже пример кода, пытающегося получить доступ к SharePoint 2010 через Client API.

```
ClientContext context = new ClientContext("http://sbaydach64");

Web myWeb=context.Web;

List myListSP = myWeb.Lists.GetByTitle(curList);

items = myListSP.GetItems(
    CamlQuery.CreateAllItemsQuery());

context.Load(items);
context.ExecuteQueryAsync(client_Completed, client_Error);
```

В этом коде метод **ExecuteQueryAsync** принимает два параметра: обработчик, вызывающийся при успешном выполнении запроса и обработчик, кото-

рый вызывается в случае ошибки. Следует отметить, что обработчик, реагирующий на ошибку, также вызывается в параллельном потоке, который не имеет доступа к интерфейсу. Поэтому, если Вы планируете сгенерировать исключение, то нужно инициировать вызов метода в интерфейсном потоке, передав ему параметры ошибки.

При всем этом, если Вы не задали обработчик, реагирующий на ошибку, то ошибка будет передана в JavaScript, где метод, указанный в **onerror**, успешно ее перехватит и сможет обработать.

Тестирование Silverlight-приложений

В заключение главы хотелось бы остановиться на тестировании Silverlight-приложений. Естественно, что речь идет о тестах, которые создает сам разработчик, — это Unit тесты. Вопросы создания тест кейсов, тест планов и др. не интересуют разработчиков, а тестер не будет читать эту книгу. О разработке же через тестирование, разработчики говорят всегда. На эту тему написано много книг и существует множество полезных утилит и библиотек, облегчающих работу и создание Unit тестов. Не исключением является и Visual Studio. Так, для создания тестов, тут присутствует специальный шаблон проектов, а также множество видов тестов, включая Unit тесты. Но все это работает и интегрируется с Visual Studio только для управляемого кода, работающего на «взрослом» .NET Framework. Если говорить о Silverlight-приложениях, то использовать встроенные в Visual Studio механизмы нельзя. Ведь Silverlight-приложение работает в ограниченном, безопасном контексте, а это означает, что в том же контексте должен работать и тест. Последнее исключает взаимодействие с Visual Studio и использование встроенных утилит.

Несмотря на существующие проблемы, сообщество разработчиков при поддержке Microsoft решило проблему создания тестов и для Silverlight-приложений. Для этого была разработана специальная сборка **Microsoft.Silverlight.Testing.dll**, которая позволяет создавать не просто новые тесты (формально это позволяет другая сборка), но и отображать результаты работы тестов внутри Silverlight-интерфейса, интегрируя отдельное Silverlight-приложение в ту же страницу, что и тестируемое приложение. Кроме этого, специально для Silverlight была разработана сборка **Microsoft.VisualStudio.QualityTools.dll**. Эта сборка содержит практически все классы, характерные для одноименной сборки в «большом» .NET Framework. Поэтому если Вы раньше разрабатывали Unit тесты, то разработка тестов для Silverlight будет Вам знакома.

Выше я сказал, что механизмы для создания тестов были реализованы сообществом разработчиков. Это означает, что представленные сборки, а также шаблон проекта для Visual Studio, можно найти на сайте **codeplex.com**. Фактически, сборки и шаблон для Visual Studio входят в инсталляцию **Silverlight Toolkit**, который мы рассматривали в предыдущих главах. Поэтому, если у Вас уже есть **Silverlight Toolkit**, то можно приступить к созданию тестов, а если нет, то его можно скачать по адресу http://silverlight.codeplex.com.

Итак, перейдем к созданию Unit тестов. Для этого в существующее решение (Вы же пишете эти тесты для одного из приложений) необходимо добавить проект на основе шаблона **Silverlight Unit Test Application**.

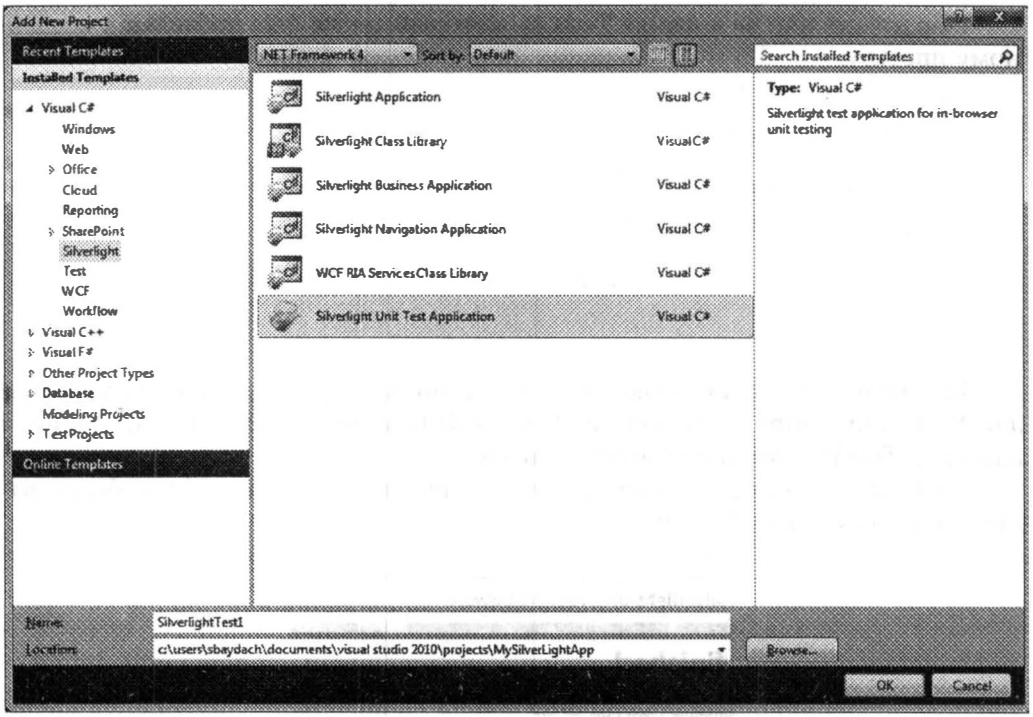


Рис. 12.4. Создание проекта для работы с тестами

Если **Silverlight Toolkit** был установлен правильно, то этот тип проекта должен появиться в разделе Silverlight. Test Project для наших целей решительно не подходит.

При создании проекта не стоит добавлять еще одно Web-приложение, которое будет использоваться в качестве хоста. Тестовую страницу мы будем создавать самостоятельно.

В созданном проекте было добавлено несколько файлов, одним из которых есть App.xaml.cs. Откройте этот файл. Фактически его структура соответствует классической реализации класса приложения, за исключением метода Startup.

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    RootVisual = UnitTestSystem.CreateTestPage();
}
```

Как видно, тут вместо создания элемента, наследуемого от **UserControl**, вызывается статический метод **CreateTestPage**, который содержится в классе **UnitTestSystem**. Задача этого метода состоит в создании тестовой страницы для разрабатываемого Silverlight-приложения, включая вставку приложения, отображающего результаты тестирования. Метод реализован в двух вариантах. Вторая сигнатура метода описывает параметр типа **UnitTestSettings**, объект которого позволяет сортировать и фильтровать все доступные тесты.

Обратимся теперь к файлу Tests.cs, который также был добавлен к созданному проекту.

```
[TestClass]
public class Tests
{
    [TestMethod]
    public void TestMethod1()
    {
        Assert.Inconclusive();
    }
}
```

Как видно, тут используются те же атрибуты, что и при создании тестов для WPF или Forms приложений. Так, **TestClass** определяет наличие тестов в классе, а **TestMethod** задает очередной тест.

Если Вы запустите созданное приложение, то результат работы будет выглядеть следующим образом:

Рис. 12.5. Результат работы приложения

Как видно, было создано приложение с достаточно информативным интерфейсом, а все тесты (в нашем случае один) были запущены.

Остается добавить ссылку на тестируемое приложение и можно приступать к созданию реальных тестов. Чтобы это сделать, щелкните правой кнопкой на проекте с тестами и вызовите Add Reference.

Если Вы правильно разделили логику и представление, то написание тестов для логики не должно составить труда. Единственное, с чем Вы можете столкнуться, — это недоступность некоторых методов из-за модификатора доступа **private**. Поэтому, рекомендую сразу же **private** изменить на **internal**. При этом чтобы к **internal** методам в тестируемом проекте появился доступ добавьте в тестируемый проект атрибут:

```
[assembly: InternalsVisibleTo("MyTest")]
```

Этот атрибут можно разместить в AssemblyInfo.cs файле.

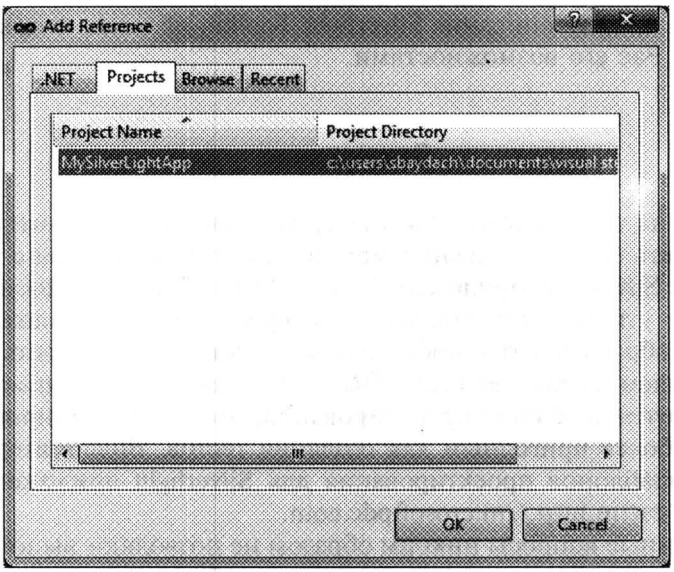


Рис. 12.6. Добавление ссылки на тестируемый проект

Если Вы планируете тестировать интерфейс, то тут также необходимо выполнить ряд действий. Во-первых, **Silverlight Toolkit** не предназначен для тестирования целого приложения. Если Вы хотите протестировать конкретный элемент, порожденный от **UserControl**, то его нужно явно создать в начале работы теста. Это означает, что класс, наследуемый от Application тестируемого приложения, никак не используется. Поэтому если в App.xaml.cs или App.xaml есть какие-то объявления (чаще всего это ресурсы), то их нужно перенести в файлы, ассоциированные с тестируемым элементом. Во-вторых, чтобы Toolkit смог найти тест, создающий и тестирующий отдельный элемент интерфейса, класс, описывающий тест, нужно породить от **SilverlightTest**. Вот как может выглядеть подобный класс:

```
[TestClass]
public class UITest : SilverlightTest
{
    MySilverLightApp.MainPage _page;

    [TestInitialize]
    public void PreparePage()
    {
        _page = new MySilverLightApp.MainPage();

        this.TestPanel.Children.Add(_page);
    }
}
```

Тут мы создали экземпляр нашего элемента и добавили его на панель, которая предназначена для отображения тестируемых элементов. При этом наш метод мы пометили атрибутом **TestInitialize**. Это означает, что он будет вызываться всякий раз, когда понадобится инициализировать тесты.

На этом я закончу описание Silverlight Toolkit, полагая, что, как минимум, заинтересовал Вас его возможностями.

Заключение

Несмотря на то, что тема этой главы достойна целой книги, мы постарались рассмотреть наиболее важные моменты, которые связаны с тестированием и отладкой Silverlight-приложений. Так, Visual Studio предоставляет достаточно мощные утилиты для отладки, а модель Silverlight-приложений такова что позволяет обрабатывать ошибки любых типов. Если говорить о тестировании, то в будущем, вероятнее всего, Вы столкнетесь с необходимостью применять те или другие шаблоны проектирования, чтобы унифицировать Ваш код и сделать его более пригодным для создания тестов. Видеоматериалы по использованию шаблонов проектирования для Silverlight-приложений Вы сможете найти на сайте http://microsoftpdc.com.

Рассмотренные вопросы никоим образом не позволяют выполнить профилирование приложения и осуществить поиск наиболее критичных мест с точки зрения производительности. Подобные утилиты мы рассмотрим в главе 14.

Глава 13

СОЗДАНИЕ СЛОЖНЫХ ПРИЛОЖЕНИЙ

Разработка приложений, работающих вне браузера

Начиная с третьей версии Silverlight, программист получил мощную возможность, позволяющую создавать приложения, поддерживающие работу «вне браузера». Подобные приложения могут быть развернуты в кэш пользователя и запущены в отключенном от сети режиме.

Для того чтобы позволить пользователю установить приложение в кэш, необходимо выполнить два шага еще на этапе разработки:

- Вызвать метод **Install** у объекта **Application**. При этом данный метод должен быть вызван только из обработчика события, инициируемого пользователем. Например, событие может быть связано с нажатием на кнопку:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Application.Current.Install();
}
```

- Добавить несколько элементов в манифест приложения. Эти элементы необходимы в первую очередь для того, чтобы предоставить дополнительные параметры, которые нужны при развертывании приложения на стороне клиента (надписи, иконки). Вот пример манифеста:

```
<Deployment
xmlns="http://schemas.microsoft.com/client/2007/deployment"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml1">
    <Deployment.Parts>
    </Deployment.Parts>
    <Deployment.ApplicationIdentity>
        <ApplicationIdentity
            ShortName="Offline App"
            Title="My Title">
            <ApplicationIdentity.Blurb>
              This is my offline App
            </ApplicationIdentity.Blurb>
        </ApplicationIdentity>
    </Deployment.ApplicationIdentity>
</Deployment>
```

Если Вы хотите также описать и иконки, то добавьте следующий элемент в раздел **ApplicationIdentity**:

```
<ApplicationIdentity.Icons>
   <Icon Size="16x16">icons/16x16.png</Icon>
   <Icon Size="32x32">icons/32x32.png</Icon>
   <Icon Size="48x48">icons/48x48.png</Icon>
   <Icon Size="128x128">icons/128x128.png</Icon>
</ApplicationIdentity.Icons>
```

Если иконки не указаны явно, то подставляется иконка по умолчанию.

Благо при использовании Visual Studio 2010 Вам нет необходимости править манифест приложения. Достаточно войти в настройки проекта и отметить на вкладке **Silverlight** параметр **Enable running application out of the browser**. После этого можно перейти к окну, в котором настраиваются параметры приложения, работающего вне браузера:

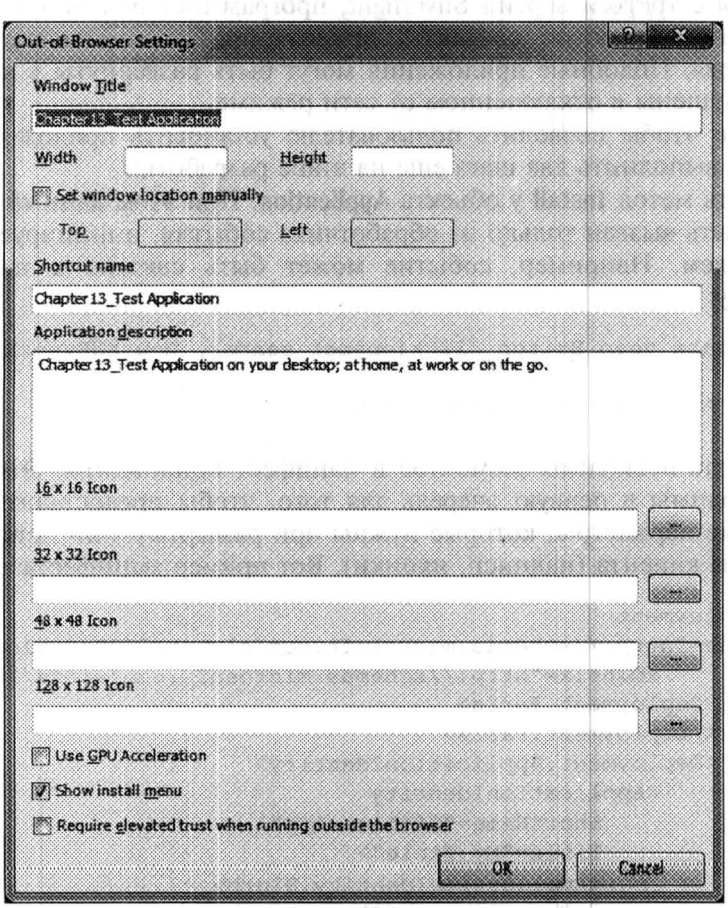


Рис. 13.1. Настройка приложения

Запустив приложение, способное работать вне браузера, пользователь получает две возможности установки приложения на локальный компьютер.

- Инициировав контекстное меню и выполнив соответствующую команду:

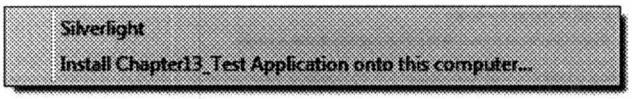


Рис. 13.2. Установка приложения из контекстного меню

- Инициировав событие, вызывающее метод **Install**.

Независимо от способа установки, пользователь должен будет подтвердить установку приложения. Это возможно с помощью следующего окна, которое появляется всякий раз, как пользователь собирается установить приложение на локальный компьютер:

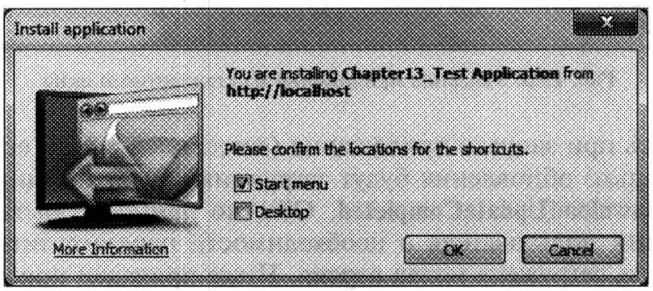


Рис. 13.3. Установка приложения на локальный компьютер

Итак, пользователь должен выбрать, где он хочет видеть иконку приложения, и нажать **OK** для установки приложения в кэш. Нужно отметить, что для установки приложения в кэш пользователь может НЕ обладать административными правами.

Аналогично, чтобы удалить приложение с локального компьютера, пользователь может выполнить одно из двух:

- Инициировать команду **Remove** из контекстного меню — при этом вовсе не важно, какая из копий приложения сейчас запущена (локальная или в браузере):

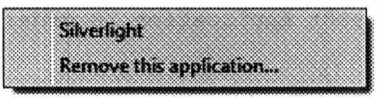


Рис. 13.4. Удаление приложения с помощью контекстного меню

- Воспользоваться стандартным окном Windows для удаления программ (рис. 13.5).

Несомненно, чтобы эффективно управлять приложением, работающим вне браузера, программист имеет возможность инициировать процедуру обновления приложения. Для этого существует простой метод **CheckAndDownloadUpdateAsync**, который можно вызвать, используя текущий объект **Application**. Метод инициирует загрузку необходимых обновлений, но лишь в том случае, если клиент имеет нужную версию Silverlight. Данный метод очень

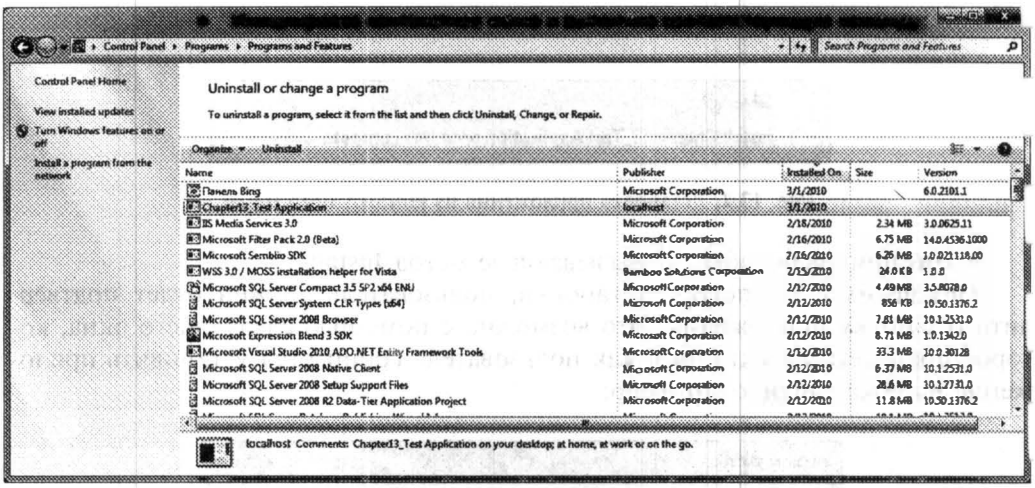


Рис. 13.5. Удаление приложения из стандартного окна

удобно вызывать при запуске приложения (естественно, если есть связь с Интернет). Как только обновления будут загружены, будет инициировано событие **CheckAndDownloadUpdateCompleted**. Обычно данное событие используют для нотификации пользователя о необходимости перезагрузить приложение, чтобы могла быть загружена новая версия. Ниже приведен пример кода, который выполняет проверку обновлений:

```
private void updateButton_Click(object sender, RoutedEventArgs e)
{
    App.CheckAndDownloadUpdateAsync();
}

private void App_CheckAndDownloadUpdateCompleted(object sender,
    CheckAndDownloadUpdateCompletedEventArgs e)
{
    if (e.UpdateAvailable)
    {
        MessageBox.Show(
           "Restart your Application in order to launch updated
version");
    }
    else if (e.Error != null &&
        e.Error is PlatformNotSupportedException)
    {
        MessageBox.Show("An application update is available, " +
            "but it requires a new version of Silverlight. ");
    }
}
```

Чтобы проверить, установлено ли Ваше приложение на локальный компьютер пользователя и запущено ли оно вне браузера, Вам могут помочь следующие два свойства объекта **Application**.

1) **InstallState** — позволяет определить, установлено ли приложение на локальный компьютер пользователя, при этом не важно какую версию приложения пользователь запустил в данный момент;

2) **IsRunningOutOfBrowser** — позволяет определить, запущен ли данный экземпляр приложения вне браузера.

В дополнении к этим свойствам Вам может пригодиться событие **InstallStateChanged** объекта **Application**, инициирующееся при установке или удалении приложения с компьютера пользователя.

Наконец, чтобы определить наличие подключения к сети, разработчик обладает специальным статическим методом **GetIsNetworkAvailable**, который содержится в классе **NetworkInterface**. А для реакции на изменение состояния сети можно определить обработчик статического события **NetworkAddressChanged**, которое описано в классе **NetworkChange**.

Isolated Storage

Начиная со второй версии Silverlight, разработчик получил возможность сохранять состояние приложения, настройки пользователя и практически любые данные на компьютере пользователя, используя специальное изолированное хранилище, расположенное в файловой системе пользователя. Фактически существует два класса для этой цели:

- **IsolatedStorageFile** — с помощью этого класса разработчик может работать с изолированным хранилищем как с виртуальной файловой системой, создавая файлы и папки. Физически структура изолированного хранилища содержится в одном файле (иногда в нескольких), который хранит как информацию о виртуальной файловой системе, так и сами данные;
- **IsolatedStorageSettings** — чтобы не заставлять разработчика каждый раз создавать какие-то файлы для хранения элементарных настроек, которые можно записать в формате ключ-значение, Silverlight предлагает специальный класс, который облегчает жизнь разработчику.

Рассмотрим оба класса и начнем с **IsolatedStorageSettings**.

IsolatedStorageSettings

Фактически **IsolatedStorageSettings** является вспомогательным классом для **IsolatedStorageFile** и выполняет функции надстройки над последним. Чтобы приступить к работе с **IsolatedStorageSettings**, необходимо создать соответствующий объект, ссылающийся на изолированное хранилище одного из двух типов:

- Хранилище уровня приложения — к данному хранилищу имеет доступ только приложение, инициировавшее в него запись;
- Хранилище уровня домена — к данному типу изолированного хранилища имеют доступ все приложения «родного» домена;

Поскольку для каждого из типов хранилища можно управлять только одним объектом **IsolatedStorageSettings**, то конструктором этот класс не облада-

ет. Вместо конструктора используются два статических свойства: **ApplicationSettings** и **SiteSettings**, которые и позволяют создать или вернуть ссылку на уже готовые объекты, связанные с изолированными хранилищами приложения или домена соответственно.

Получив ссылку на готовый объект, с ним можно обращаться как с обычной коллекцией типа **Dictionary**. Тут есть возможность и обращаться к значению по ключу, и удалять запись с ключом командой Remove, и добавлять новую запись с помощью команды Add и др.

IsolatedStorageFile

Более универсальный механизм для хранения данных приложения — это **IsolatedStorageFile**. Как и в предыдущем случае, чтобы создать объект этого класса, нужно обратиться к статическим членам класса. На этот раз это два метода:

- **GetUserStoreForApplication** — возвращает ссылку на объект **IsolatedStorageFile**, который ассоциирован с конкретным приложением;
- **GetUserStoreForSite** — возвращает ссылку на объект **IsolatedStorageFile**, который ассоциирован с доменом приложения.

Поскольку речь идет уже не о хранении небольшого количества параметров, занимающих немного места на диске, а, возможно, о хранении изображений, видео или других типах файлов, следующее, что необходимо проверить, — это наличие места в изолированном хранилище. По умолчанию изолированному хранилищу выделяется 1 Мб пространства. Чтобы определить количество свободного или занятого пространства в данный момент времени, используют следующие три свойства:

- **AvailableFreeSpace** — доступное свободное пространство;
- **Quota** — общее доступное пространство в рамках данного изолированного хранилища;
- **UsedSpace** — используемое пространство на данный момент времени в рамках заданной квоты.

Используя перечисленные выше свойства и метод **IncreaseQuotaTo**, можно попытаться выделить дополнительное пространство для хранения данных приложения. Ниже показан пример кода, использующего этот метод и некоторые из описанных свойств:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    try
    {
        using (var store =
            IsolatedStorageFile.GetUserStoreForApplication())
        {
            Int64 spaceToAdd = 90000000;
            Int64 curAvail = store.AvailableFreeSpace;

            if (curAvail < spaceToAdd)
            {
                if (!store.IncreaseQuotaTo(store.Quota + spaceToAdd))
```

```
                {
                    //пользователь не согласен
                }
                else
                {
                    //пользователь разрешил операцию
                }
            }
        }
    }
    catch (IsolatedStorageException)
    {

    }
}
```

Как Вы смогли заметить, метод **IncreaseQuotaTo** инициирует вызов диалогового окна, предупреждающего пользователя об увеличении изолированного хранилища. Пользователь может как согласиться, так и отказаться от данной операции.

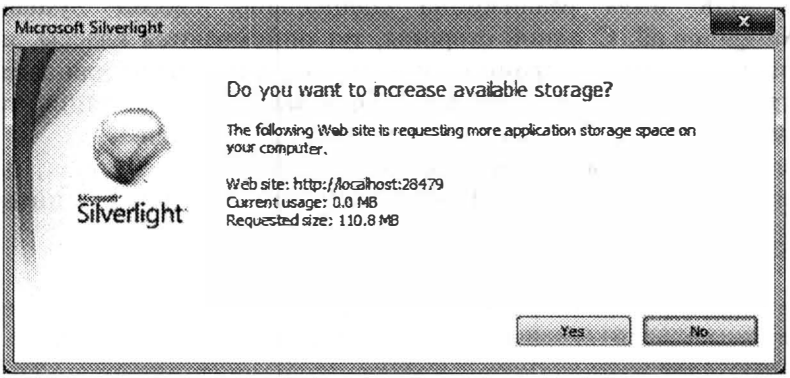


Рис. 13.6. Запрос пользователю на выделение дополнительного места

Получив ссылку на объект, связанный с изолированным хранилищем, и убедившись в наличии необходимого места, можно приступать к работе с виртуальной файловой системой. Для этого существует целый набор простых методов:

- **CopyFile** — создает копию файла внутри изолированного хранилища;
- **CreateFile** — создает новый файл внутри виртуального файлового пространства, возвращая ссылку на **IsolatedStorageFileStream**, который может использоваться для запуска в файл;
- **CreateDirectory** — создает директорию внутри изолированного хранилища;
- **DeleteDirectory** — удаляет директорию внутри изолированного хранилища;
- **DeleteFile** — удаляет файл;
- **DirectoryExists** — возвращает **true**, если директория существует;
- **FileExists** — возвращает **true**, если файл существует;
- **GetDirectoryNames** — возвращает список директорий;
- **GetFileNames** — возвращает список файлов;

- **MoveFile** — перемещает файл внутри виртуального файлового пространства;
- **MoveDirectory** — перемещает директорию внутри виртуального файлового пространства;
- **OpenFile** — открывает файл и возвращает ссылку на объект **IsolatedStorageFileStream**, который можно использовать как для записи так и для чтения данных;
- **Remove** — очищает изолированное хранилище от всех файлов и директорий.

Навигация в Silverlight-приложениях

Разрабатывая сложные Silverlight-приложения, разработчик рано или поздно, сталкивается с невозможностью поместить всю функциональность внутри одного окна приложения. Так, часто требуется реализовать интерфейс на несколько страниц с поддержкой механизма навигации между ними. Поэтому чтобы обеспечить разработчиков необходимыми инструментами, в Silverlight 3 SDK была реализована специальная сборка **System.Windows.Controls.Navigation.dll**. С одной стороны, тут описываются классы, позволяющие создавать отдельные страницы и выделять контейнеры для загрузки созданных страниц, а с другой — тут реализован механизм навигации, позволяющий перемещаться между страницами внутри приложения.

Чтобы познакомиться с механизмом навигации в Silverlight, попробуем реализовать небольшой пример. Начнем с реализации простого интерфейса:

```
<UserControl x:Class="Chapter13_Test.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:nav=

"clr-namespace:System.Windows.Controls;assembly=System.Windows.Cont
rols.Navigation">
    <StackPanel Name="LayoutRoot" Background="White">
        <StackPanel Orientation="Horizontal"
HorizontalAlignment="Right">
            <HyperlinkButton Content="Home" FontSize="16"
Margin="5"
                Tag="/Views/Home.xaml" Click="Navigate_Clicked">
            </HyperlinkButton>
            <HyperlinkButton Content="Employees" FontSize="16"
Margin="5"
                Tag="/Views/Employees.xaml"
Click="Navigate_Clicked">
            </HyperlinkButton>
            <HyperlinkButton Content="About" FontSize="16" Margin="5"
                Tag="/Views/About.xaml" Click="Navigate_Clicked">
            </HyperlinkButton>
        </StackPanel>
```

```
        <StackPanel HorizontalAlignment="Center" Width="500">
            <nav:Frame Name="navFrame"
HorizontalAlignment="Stretch"
                Source="/Views/Home.xaml">
            </nav:Frame>
        </StackPanel>

    </StackPanel>
</UserControl>
```

Представленный XAML файл описывает примитивный интерфейс, содержащий три элемента типа **HyperlinkButton**, которые мы будем использовать для навигации, и элемент типа **Frame**. Именно класс **Frame** описывает элемент управления, определяющий область, доступную для загрузки внутренних страниц. Чтобы использовать этот класс, нам пришлось подключить сборку **System.Windows.Controls.Navigation.dll**.

Как видно из кода, элементы **HyperlinkButton** и **Frame** ссылаются на некоторые XAML страницы. Это внутренние страницы, которые мы будем использовать для навигации между различными частями приложения. Создайте директорию Views и добавьте туда 4 страницы со следующими именами: **Home.xaml**, **Employees.xaml**, **About.xaml**, **Employee.xaml**. Чтобы иметь возможность отображать внутренние страницы с помощью объекта класса **Frame**, все они должны быть порождены от класса **Page**. Чтобы добиться этого, создавайте страницы на базе специального шаблона Silverlight Page (рис. 13.7).

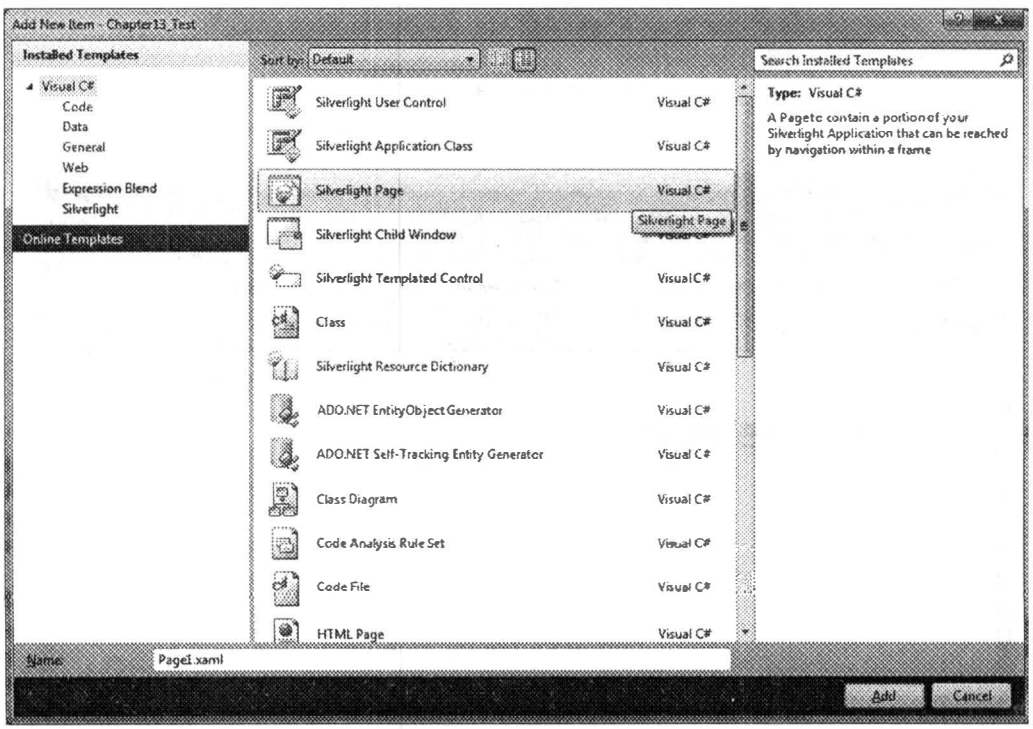


Рис. 13.7. Создание страницы навигации

Рассмотрим исходный код одной из созданных страниц:

```
<navigation:Page x:Class="Chapter13_Test.Views.Employee"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:navigation=
"clr-namespace:System.Windows.Controls;assembly=System.Windows.
Controls.Navigation"
          Title="Employee">
    <StackPanel x:Name="LayoutRoot">

    </StackPanel>
</navigation:Page>
```

Как видно, в этом шаблоне вместо **UserControl** используется **Page**, что по зволяет загрузить страницу в элемент **Frame**.

Добавьте в каждую из страниц элемент **TextBlock**, чтобы продемонстриро вать возможности навигации (как-то визуально увидеть сам переход):

```
<TextBlock Text="Employees Page" FontSize="32"></TextBlock>
```

Наконец, воспользовавшись свойством **Tag** объектов **HyperlinkButton**, реа лизуем обработчик события **Click** для этих объектов:

```
private void Navigate_Clicked(object sender, RoutedEventArgs e)
{
    HyperlinkButton but = sender as HyperlinkButton;
    navFrame.Navigate(new
Uri(but.Tag.ToString(),UriKind.Relative));
}
```

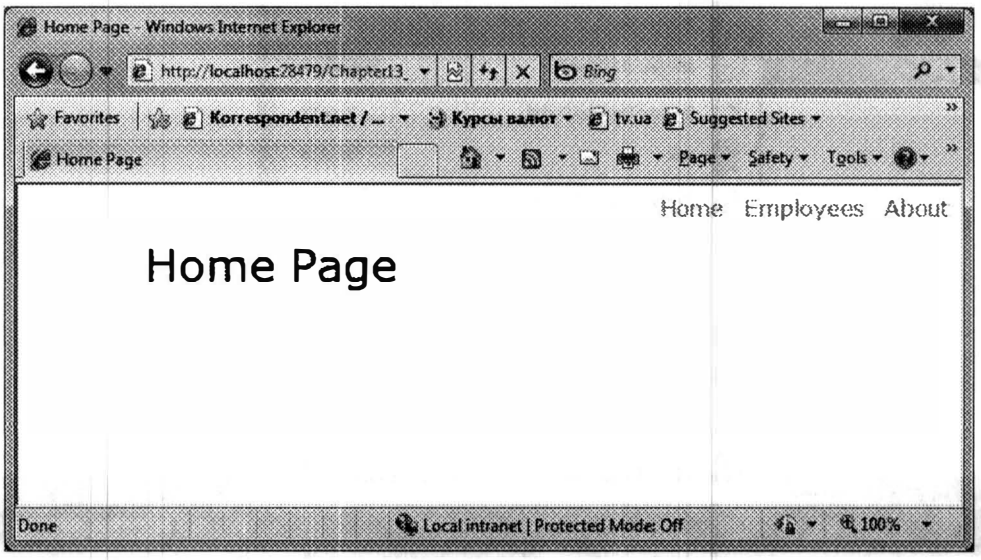


Рис. 13.8. Результат работы приложения

Как Вы видите, тут мы использовали достаточно простой метод **Navigate**, чтобы перейти на конкретную страницу внутри элемента **Frame**.

Результат работы приложения показан на рис. 13.8.

Выбрав один из пунктов меню, Вы сможете увидеть следующую ссылку в заголовке браузера: http://localhost:28479/Chapter13_TestTestPage.aspx#/Views/Home.xaml. Иными словами, имя XAML страницы передается в качестве параметров. Однако, Silverlight предоставляет возможность модифицировать подобную ссылку, заменив ее на что-то более привлекательное. Это возможно с помощью объекта **UriMapper**. Рассмотрим модифицированный код:

```
<UserControl x:Class="Chapter13_Test.MainPage"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:nav=
"clr-namespace:System.Windows.Controls;assembly=System.Windows.
Controls.Navigation"
     xmlns:mapper=
"clr-namespace:System.Windows.Navigation;assembly=System.Windows.
Controls.Navigation">
    <StackPanel Name="LayoutRoot" Background="White">
        <StackPanel Orientation="Horizontal"
HorizontalAlignment="Right">
            <HyperlinkButton Content="Home" FontSize="16"
Margin="5"
               Tag="Home" Click="Navigate_Clicked">
            </HyperlinkButton>
            <HyperlinkButton Content="Employees" FontSize="16"
Margin="5"
               Tag="Employees" Click="Navigate_Clicked">
            </HyperlinkButton>
            <HyperlinkButton Content="About" FontSize="16"
Margin="5"
               Tag="About" Click="Navigate_Clicked">
            </HyperlinkButton>
        </StackPanel>
        <StackPanel HorizontalAlignment="Center" Width="500">
            <nav:Frame Name="navFrame"
HorizontalAlignment="Stretch"
                Source="Home">
                <nav:Frame.UriMapper>
                    <mapper:UriMapper>
                        <mapper:UriMapping Uri="Home"
                            MappedUri="/Views/Home.xaml">
                        </mapper:UriMapping>
                        <mapper:UriMapping Uri="Employees"
                            MappedUri="/Views/Employees.xaml">
                        </mapper:UriMapping>
                        <mapper:UriMapping Uri="About"
                            MappedUri="/Views/About.xaml">
                        </mapper:UriMapping>
                    </mapper:UriMapper>
                </nav:Frame.UriMapper>
```

```
            </nav:Frame>
        </StackPanel>
    </StackPanel>
</UserControl>
```

Как видите, с помощью объекта **UriMapper** мы смогли не только изменить адрес страницы, но и использовать более простой способ для взаимодействия со страницами внутри приложения (Source и Tag).

Реализуем страницу Employees следующим образом:

```
<navigation:Page x:Class="Chapter13_Test.Views.Employees"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
           xmlns:navigation=
"clr-namespace:System.Windows.Controls;assembly=System.Windows.
Controls.Navigation"
           Title="Employees" Loaded="Page_Loaded">
    <StackPanel x:Name="LayoutRoot">
        <TextBlock Text="Employees Page" FontSize="32"></TextBlock>
        <ListBox Height="200" Width="400" Margin="5" Name="lEmpl"
            SelectionChanged="lEmpl_SelectionChanged">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <Border Background="Gray" BorderThickness="2"
                        CornerRadius="10" Height="30" Width="376">
                        <StackPanel Orientation="Horizontal"
Margin="5" >
                            <TextBlock Text="{Binding FirstName}">
                            </TextBlock>
                            <TextBlock Text=" "></TextBlock>
                            <TextBlock Text="{Binding @KOD =
LastName}"></TextBlock>
                        </StackPanel>
                    </Border>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
        <navigation:Frame Name="empFrame"></navigation:Frame>
    </StackPanel>
</navigation:Page>
```

Как видно, тут мы определили список, печатающий начальную информацию о сотрудниках. Детальную же информацию мы будем выдавать на отдельной странице, которую будем загружать с помощью того же **Frame**.

Прежде чем переходить к коду страницы, реализуем класс, описывающий информацию о сотруднике. Для этого мы воспользуемся классом из главы 6, расширив его двумя статическими методами:

```
static List<EmployeeClass> l=null;

public static List<EmployeeClass> GetEmployees()
{
    if (l == null)
```

```
    {
        l = new List<EmployeeClass>();

        EmployeeClass emp = new EmployeeClass();
        emp.FirstName = "Sergiy";
        emp.LastName = "Baydachnyy";
        emp.EMail = "sbaidachni@gmail.com";
        emp.Age = 31;
        emp.Salary = 1450;

        l.Add(emp);

        emp = new EmployeeClass();
        emp.FirstName = "Viktor";
        emp.LastName = "Baydachnyy";
        emp.EMail = "viktor@gmail.com";
        emp.Age = 30;
        emp.Salary = 1250;

        l.Add(emp);
    }
    return l;
}

public static EmployeeClass GetEmployee(int id)
{
    if ((l == null)||(l.Count<id+1))
    {
        throw new Exception("List doesn't exist");
    }
    return l[id];
}
```

Эти два метода позволяют генерировать тестовый список сотрудников и ращать ссылку на весь список или отдельный элемент.
Перейдем к событиям внутри страницы Employees.xaml:

```
private void Page_Loaded(object sender, RoutedEventArgs e)
{
    lEmpl.ItemsSource = EmployeeClass.GetEmployees();
}

private void lEmpl_SelectionChanged(object sender,
    SelectionChangedEventArgs e)
{
    empFrame.Navigate(new Uri(String.Format(
       "/Views/Employee.xaml?id={0}",lEmpl.SelectedIndex),
       UriKind.Relative));
```

Тут мы реализуем привязку данных из списка сотрудников к элемент **ListBox**, а также механизм навигации по внутреннему объекту типа **Frame**.

Остается реализовать лишь страницу, отображающую информацию о де талях сотрудника. Вот как будет выглядеть XAML код:

```
<navigation:Page x:Class="Chapter13_Test.Views.Employee"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
          xmlns:navigation=
"clr-namespace:System.Windows.Controls;assembly=System.Windows.
Controls.Navigation"
          Title="Employee">
    <StackPanel x:Name="LayoutRoot">
        <TextBlock Text="Employee" FontSize="32"></TextBlock>
        <Grid x:Name="empDetails" Background="White" Width="400">
            <Grid.RowDefinitions>
                <RowDefinition></RowDefinition>
                <RowDefinition></RowDefinition>
                <RowDefinition></RowDefinition>
                <RowDefinition></RowDefinition>
                <RowDefinition></RowDefinition>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition></ColumnDefinition>
                <ColumnDefinition></ColumnDefinition>
            </Grid.ColumnDefinitions>

            <TextBlock Text="First Name:" Grid.Row="0"
Grid.Column="0">
            </TextBlock>
            <TextBox Text="{Binding FirstName, Mode=TwoWay}"
            Grid.Row="0" Grid.Column="1">
            </TextBox>

            <TextBlock Text="Last Name:" Grid.Row="1"
Grid.Column="0">
            </TextBlock>
            <TextBox Text="{Binding LastName, Mode=TwoWay}"
            Grid.Row="1" Grid.Column="1">
            </TextBox>

            <TextBlock Text="EMail:" Grid.Row="2" Grid.Column="0">
            </TextBlock>
            <TextBox Text="{Binding EMail, Mode=TwoWay}"
            Grid.Row="2" Grid.Column="1">
            </TextBox>

            <TextBlock Text="Age:" Grid.Row="3" Grid.Column="0">
            </TextBlock>
            <TextBox Text="{Binding Age, Mode=TwoWay}"
           Grid.Row="3" Grid.Column="1">
            </TextBox>
```

```
            <TextBlock Text="Salary:" Grid.Row="4" Grid.Column="0">
            </TextBlock>
            <TextBox Text="{Binding Salary, Mode=TwoWay,
StringFormat=C}"
          Grid.Row="4" Grid.Column="1">
            </TextBox>
        </Grid>
    </StackPanel>
</navigation:Page>
```

А вот и код, реализующий привязку:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    empDetails.DataContext=EmployeeClass.GetEmployee(
        int.Parse(this.NavigationContext.QueryString["id"]));
}
```

Вот и все. Мы получили полностью работоспособное приложение, позволяющее выполнять навигацию верхнего уровня, а также навигацию вглубь приложения, используя дополнительные параметры.

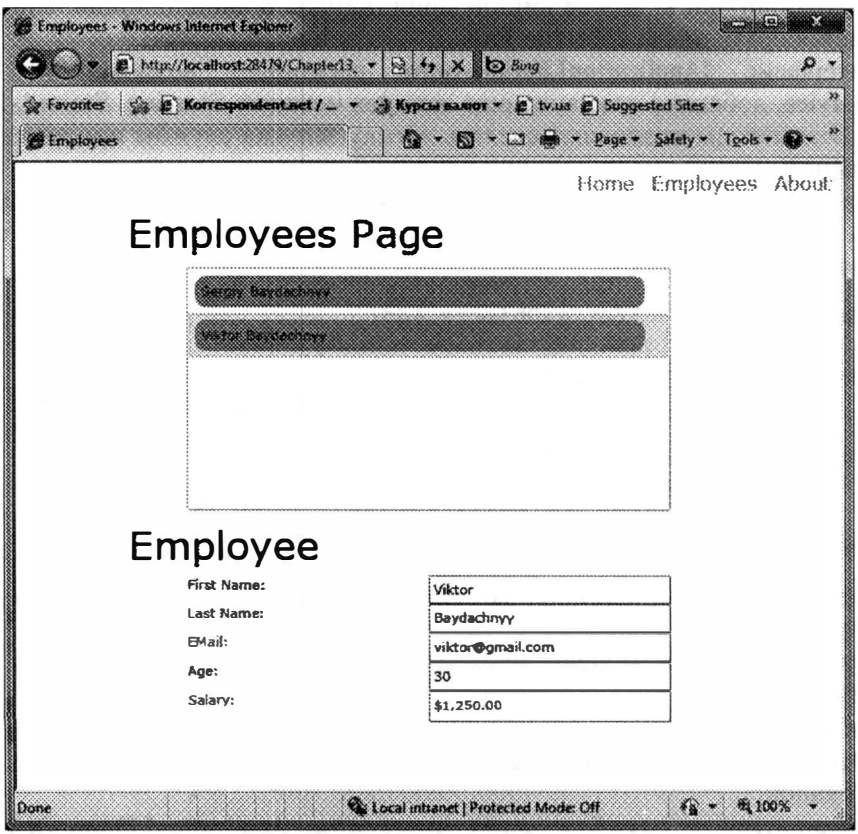


Рис. 13.9. Результат работы приложения

Расширение модели приложения

Используя объект типа **Application**, который ассоциируется с приложением, Вы можете получить: доступ к ресурсам, состоянию приложения, ссылку на встраиваемый компонент Silverlight и многое другое. Но если Вы хотите расширить возможности класса **Application**, то можете либо создать производный класс от **Application**, либо воспользоваться специальным механизмом, позволяющим расширить возможности приложения с помощью специальных служб приложения. Поскольку первый способ не очень удобный при создании большого количества проектов, обладающих одинаковыми свойствами, второй способ использовать предпочтительней.

Под службами приложения будем понимать набор объектов, созданных на базе классов, реализующих интерфейс **IApplicationService** или **IApplicationLifetimeAware**. Именно для объектов классов, реализующих эти интерфейсы, класс **Application** предоставляет специальную инфраструктуру, позволяющую хранить, запускать и получать доступ к подобным объектам.

Интерфейс **IApplicationService** описывает всего два метода: **StartService** и **StopService**. Первый метод инициируется объектом **Application** сразу перед событием **Startup**. Это позволяет службе приложений использоваться с первых шагов работы приложения. Задача **StartService** — выделить необходимые ресурсы и подготовиться к работе внутри приложения. В свою очередь, **StopService** вызывается сразу после события **Exit**.

Интерфейс **IApplicationLifetimeAware** расширяет возможности **IApplicationService**, добавляя описание еще четырех методов: **Starting**, **Started**, **Exiting** **Exited**. Эти методы позволяют сделать логику запуска службы приложения более сложной.

Итак, создав класс, реализующий один из двух интерфейсов, можно загрузить службу приложения в объект класса **Application**. Это можно сделать как в коде, так и в XAML:

```
this.ApplicationLifetimeObjects.Add(
    new SilverlightApplicationService.ApplicationService1());
this.ApplicationLifetimeObjects.Add(
    new SilverlightApplicationService.ApplicationService2());
```

```
<Application.ApplicationLifetimeObjects>
    <svc:ApplicationService1/>
    <svc:ApplicationService2/>
</Application.ApplicationLifetimeObjects>
```

Определив данные службы, можно обращаться к любой из них, в любом месте приложения.

При расширении модели приложения собственными службами нужно помнить две вещи:

- службы приложения инициализируются в порядке объявления в коде. Это позволяет строить зависимости между службами, полагаясь на их порядок;
- обращаться к службам через индексатор **ApplicationLifetimeObjects** не очень удобно, так как нужно помнить порядок следования службы. По-

этому в SDK рекомендуется объявлять статическое свойство (например, Current), которое возвращает ссылку на объект службы приложения. Тогда доступ к службе можно получать, используя имя нужного класса службы приложения.

Managed Extensibility Framework

В Silverlight 4 появился специальный механизм, позволяющий расширять ваши интерфейсы без перекомпиляции приложений, и даже без перезагрузки основного XAP, или перезагрузки самого приложения. Несмотря на впечатляющее название — Managed Extensibility Framework, механизм определяет несколько атрибутов, позволяющих определить части-расширения для одного из элементов управления и загрузить расширения во время инициализации элемента управления.

Ниже показан код, который определяет интерфейс, используемый для расширения DataGrid, а также конкретный класс, который будет использоваться как расширение:

```
public interface IGridExtension
{
    void Initialize(DataGrid grid);
}

[Export(typeof(IGridExtension))]
public class BoldExtension : IGridExtension
{
    public void Initialize(DataGrid grid)
    {
        grid.FontWeight = FontWeights.Bold;
    }
}
```

Как видите, расширение использует атрибут **Export**, позволяющее MEF найти этот класс в любой из загруженных сборок и подготовить объект этого класса для расширения элементов **DataGrid**. Чтобы использовать этот атрибут, необходимо подключить сборку **System.ComponentModel.Composition.dll**, а для подключения расширений нужна также сборка **System.ComponentModel.Composition.Initialization.dll**.

Ниже показан код, который загружает все расширения в свойство Extensions, помеченное атрибутом **ImportMany**, и применяет все доступные расширения:

```
public class ExtensibleDataGrid : DataGrid
{
    public ExtensibleDataGrid()
    {
        PartInitializer.SatisfyImports(this);

        foreach (IGridExtension extension in Extensions)
            extension.Initialize(this);
```

```
        [ImportMany]
        public IEnumerable<IGridExtension> Extensions { get; set; }
    }
```

Кроме описанных выше сборок, Вы сможете найти сборку **System.ComponentModel.Composition.Packaging.Toolkit.dll**. По идее эта сборка должна облегчать загрузку не просто отдельных сборок, а целых XAP пакетов. К сожалению, на момент написания книги документации на классы **Package** и **PackageCatalog** не было. Ведь дело даже не в том, как загрузить очередной пакет, а в наличии ограничений. Да и механизм создания XAP не очевиден. Поэтому остается использовать механизмы, описанные в третьей главе.

Заключение

В эту главу вошли несколько независимых разделов, которые объединяет тема создания корпоративных приложений. Если в Silverlight 1 и 2 основной акцент все же делался на Интернет приложениях для всех желающих, то начиная с Silverlight 3 был сделан акцент на приложения, работающие в корпоративной сети и обладающие дополнительными правами и возможностями по сравнению с «обычными» Silverlight-приложениями. Четвертая версия Silverlight лишь укрепила позиции технологии в корпоративном сегменте. Между тем, некоторые новшества могут быть эффективно использованы и для широкого круга пользователей. Так, изолированное хранилище можно использовать для хранения настроек пользователя или текущего состояния приложения, будь это состояние игры или набор данных в заполняемой форме.

Глава 14

ИСПОЛЬЗОВАНИЕ DEEP ZOOM

Что такое Deep Zoom?

Технология Deep Zoom представляет собой специальный механизм для отображения изображений с высоким разрешением.

Предположим, у Вас есть цифровая копия изображения в высоком качестве, которую Вы хотите разместить в Web. При наличии достаточно хорошего оборудования размер высококачественного изображения может составлять десятки мегабайт, что достаточно критично для Web. Кроме того существует несколько проблем, которые не ограничиваются шириной канала:

- обычно экран пользователя имеет разрешение меньшее, чем качественное изображение, сделанное на хорошем оборудовании. Это означает, что если мы публикуем изображение с хорошим разрешением, то необходимо предоставить простой механизм, позволяющий выполнять навигацию по изображению. При этом пользователь должен не только иметь возможность приближать желаемый фрагмент, но и иметь возможности навигации в виде закладок на определенные участки изображения;
- загружая полную версию изображения, пользователь может не воспользоваться предоставляемым качеством, оценив лишь изображение в общем или обратив внимание на отдельную часть изображения. Иными словами, изображение должно быть представлено в таком формате, чтобы дать возможность пользователю загрузки по требованию.

Описанные выше проблемы и помогает решить Deep Zoom технология. Ее основная идея состоит в преобразовании высококачественных изображений в целый набор файлов, предоставляющих возможность для загрузки частей изображения и для вставки маркеров, которые могут быть использованы для навигации. При этом и WPF, и Silverlight обладают механизмами, позволяющими отобразить преобразованные изображения.

Подобная технология уже нашла применение в музеях, при отображении карт, при отображении плакатов с большим количеством информации и др.

Фактически Deep Zoom позволяет разбить изображение на несколько слоев. На каждом из слоев изображение представляется в более худшем качестве, чем на предыдущем слое, и, в зависимости от разрешения, разбивается на несколько прямоугольных областей. На последнем слое изображение представлено в полном варианте, но ужатое до плохого разрешения. Именно по-

следний слой и будет использоваться для отображения всего изображения. При попытке увеличить отдельную часть изображения, будут загружаться необходимые области из следующего слоя. По мере движения пользователя вглубь изображения, слои будут меняться.

Если Вы хотите посмотреть на действующее Deep Zoom приложение, обратитесь по следующей ссылке: http://memorabilia.hardrock.com. Тут представлена полная коллекция предметов из Hard Rock кафе, принадлежащих различным знаменитостям. Благодаря Deep Zoom Вы можете перемещаться между фотографиями, используя меню или выполняя навигацию по изображениям непосредственно. При этом, когда один из предметов попадает в фокус пользователя, то тут же появляется его детальное описание.

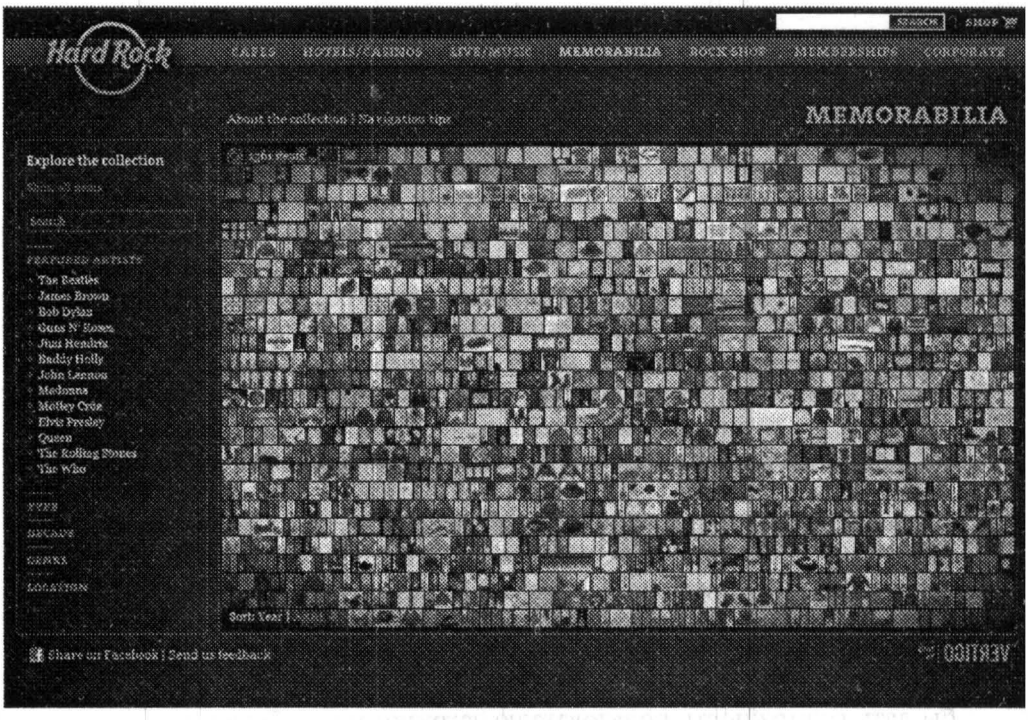


Рис. 14.1. Hard Rock кафе коллекция

Использование Deep Zoom Composer

Чтобы преобразовать изображение в формат Deep Zoom, необходимо использовать утилиту Deep Zoom Composer. Скачать эту утилиту Вы можете совершенно бесплатно с сайта Microsoft, задав в строке поиска название утилиты.

Установив утилиту на свой компьютер, можно приступить к созданию изображений в формате Deep Zoom. Следует отметить, что работа утилиты не ограничивается преобразованием только одного изображения в формат Deep Zoom. Тут Вы можете создавать коллекции изображений, добавлять механиз-

мы навигации, а также преобразовывать созданные коллекции в приложения, которые затем можно использовать в Silverlight или WPF.

Работа с утилитой Deep Zoom Composer разбита на три этапа.

На первом этапе создайте проект с помощью команды File->New Project и перетащите (импортируйте) все изображения, которые Вы хотите использовать для создания коллекции. Несомненно, что изображение может быть и одно, если речь идет о преобразовании какого-то документа в хорошем качестве.

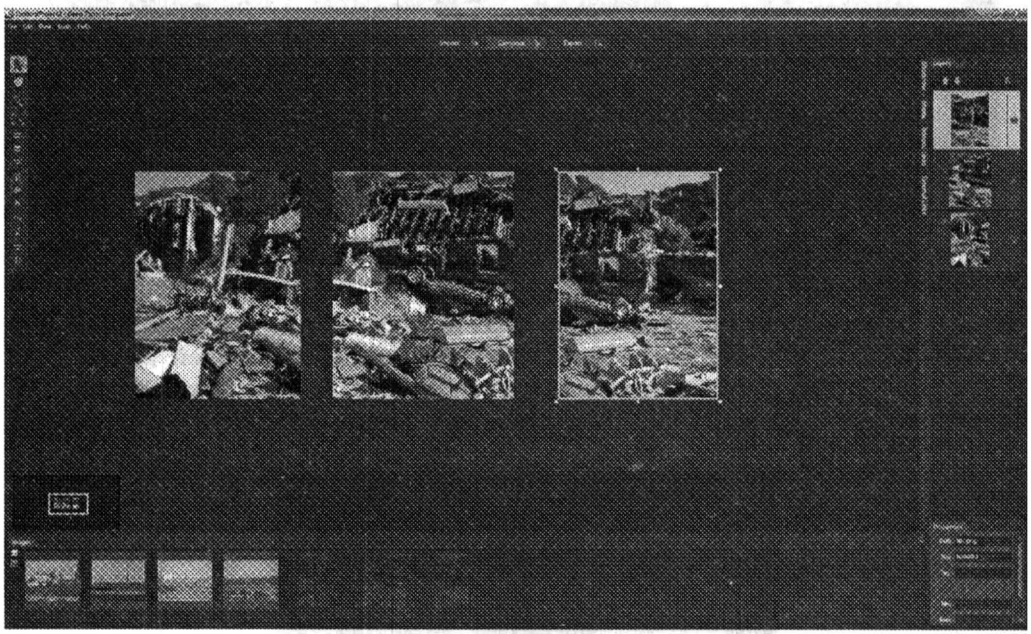


Рис. 14.2. Утилита Deep Zoom Composer

Второй этап состоит в подготовке изображения или коллекции для преобразования. На этапе подготовки Вы можете не просто позиционировать изображения относительно друг друга, но и накладывать изображения, варьируя их размер.

Одна из интересных возможностей Deep Zoom Composer состоит в создании панорамных фото. Так, недавно путешествуя по США, я проезжал мимо разбитого самолета (его сбили инопланетяне во время очередного вторжения). Мне удалось сделать несколько снимков, но, поскольку автобус ехал очень быстро, а самолет находился рядом с автобусом, я не смог сфотографировать всю панораму. С помощью Deep Zoom Composer я могу импортировать имеющиеся у меня фотографии, переместить их на рабочую поверхность и попробовать создать панорамное фото, используя контекстное меню (выбрав все нужные фотографии). На рис. 14.3 показано, что у меня получилось.

Судя по сформированному изображению, автобус очень сильно трусило. Но, благодаря Deep Zoom Composer, я могу выбрать наиболее удачную область сформированного изображения, обрезав все черные углы, и сохранить его на рабочей поверхности.

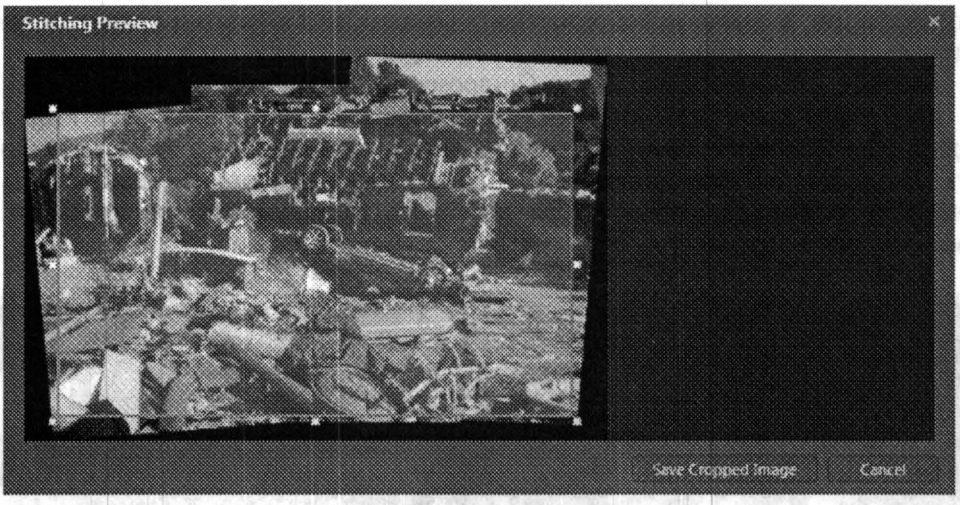


Рис. 14.3. Формирование панорамы

Сформировав изображение, можно перейти к механизмам навигации Так, используя панель навигации, Вы можете любое изображение привязать к внешней ссылке или сослаться на другое изображение в Вашей коллекции Раздел меню позволяет отобразить меню навигации по вашей коллекции. Ту Вы можете сгенерировать и отредактировать меню на основе создаваемой

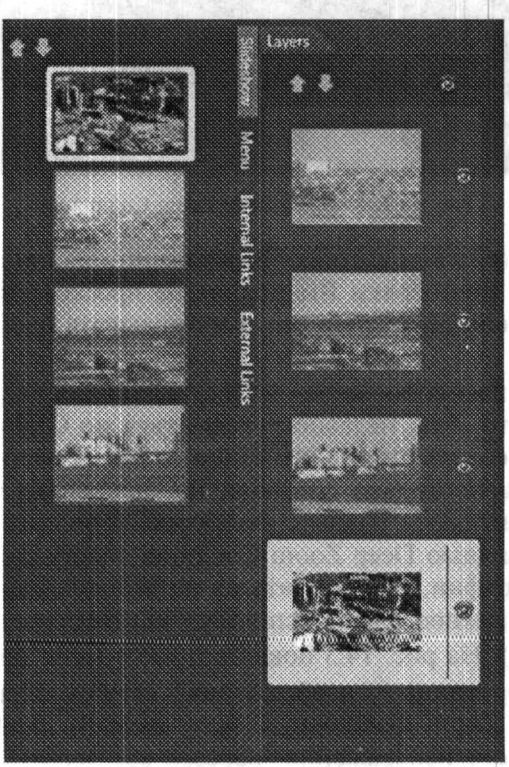


Рис. 14.4. Создание навигации

коллекции. Кроме того, есть возможность задать последовательность фотографий для автоматического отображения их пользователю с некоторым промежутком времени (рис. 14.4).

Задав все параметры коллекции, можно переходить к преобразованию коллекции в формат Deep Zoom.

На этом этапе у Вас есть выбор: либо разместить коллекцию на общедоступной службе DeepZoomPix, либо преобразовать коллекцию и сохранить результаты на локальном диске.

Если Вы решили разместить коллекцию фотографий на DeepZoomPix, то тут достаточно иметь Live ID аккаунт, который позволит осуществить аутентификацию на службу и опубликовать коллекцию с определенным именем (рис. 14.5).

Преимуществом этого подхода состоит в том, что Вы можете предоставить не просто ссылку на эту службу с Вашими фотографиями, но и получить элемент **iframe**, который можно вставить в свой блог или любой Web-сайт для отображения коллекции.

Естественно, что мы, разработчики, вероятнее всего захотим сохранить результаты преобразования на диск, чтобы использовать их в собственных Silverlight-приложениях. Для этого нужно перейти на вкладку Custom и выбрать необходимые параметры (рис. 14.6).

Рис. 14.5. Размещение коллекции в Web

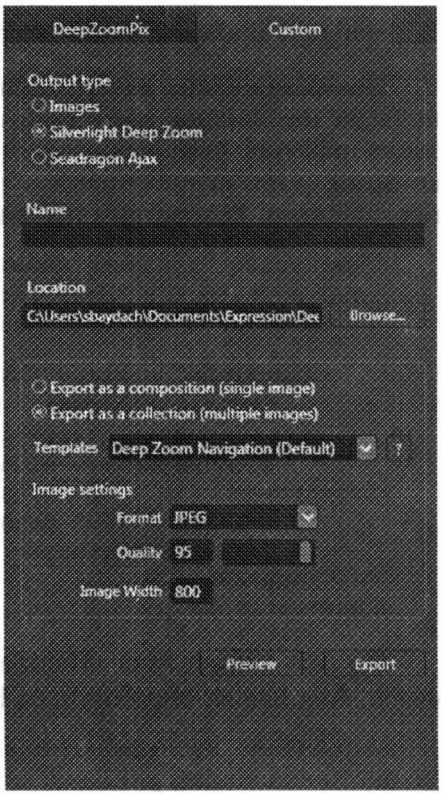


Рис. 14.6. Преобразование в формат Deep Zoom

Если Вы хотите просто получить набор изображений для вставки в свое Silverlight-приложение, то следует выбрать тип приложения Images, но если Вас устроит готовое Silverlight-приложение, то можно выбрать и Silverlight Deep Zoom.

Работа с Deep Zoom в Silverlight

Рассмотрим, как использовать Deep Zoom изображения в собственных Silverlight-проектах.

Если Вы хотите использовать сгенерированные файлы в своем проекте, то Вам понадобится файл **dzc_output.xml** и папка **dzc_output_images** и **dzc_output_files** с файлами изображения. Так, если Вы создавали изображение как коллекцию картинок (это позволяет в приложении работать с каждой картинкой, как с отдельным объектом), то потребуются обе папки, а если Вы планируете работать с изображением как с единым целым, то достаточно папки **dzc_output_files**.

Чтобы работать с Deep Zoom изображениями, Silverlight предлагает единственный элемент управления — **MultiScaleImage**. Фактически, чтобы отобразить изображение, достаточно передать этому элементу ссылку на dzc_output.xml:

```
<MultiScaleImage x:Name="deepZoomObject" Source="dzc_output.xml" />
```

Отобразив Ваше изображение в объекте **MultiScaleImage**, Вы не имеете никаких механизмов навигации по изображению. Фактически, чтобы реализовать обычное увеличение изображения, Вам необходимо описать весь код, который будет манипулировать свойствами объекта типа **MultiScaleImage** самостоятельно. Рассмотрим небольшой пример:

```
<UserControl x:Class="Chapter14_MultiScale.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <StackPanel x:Name="LayoutRoot" Background="White">
        <MultiScaleImage
           Source="http://localhost:51500/ClientBin/dzc_output.xml"
           MouseEnter="mImage_MouseEnter"
MouseLeave="mImage_MouseLeave"
           Width="400" Height="300" Name="mImage">
        </MultiScaleImage>
    </StackPanel>
</UserControl>
```

Тут мы определили **MultiScaleImage** и два обработчика событий для мыши. Код наших обработчиков будет выглядеть следующим образом:

```
void mImage_MouseLeave(object sender, MouseEventArgs e)
{
    mImage.ZoomAboutLogicalPoint(0.5, 0.5, 0.5);
```

```
void mImage_MouseEnter(object sender, MouseEventArgs e)
{
    mImage.ZoomAboutLogicalPoint(2, 0.5, 0.5);
}
```

В данном случае метод **ZoomAboutLogicalPoint** позволяет увеличивать в число раз, указанное в первом параметре. В нашем примере мы увеличиваем изображение в два раза при наведении мыши, и уменьшаем к исходному размеру при выходе мыши за пределы элемента. Кроме параметра, связанного с увеличением, этот метод позволяет задать точку, относительно которой происходит увеличение. Учитывая то, что область **MultiScaleImage** представляется как нормированный прямоугольник со сторонами равными единице, то значение (0.5, 0.5) как раз будет указывать на центр прямоугольника.

Кроме метода **ZoomAboutLogicalPoint**, **MultiScaleImage** предлагает два метода, позволяющих преобразовывать логические координаты в координаты мыши и наоборот, — это **LogicalToElementPoint** и **ElementToLogicalPoint**. Наконец, чтобы работать с коллекциями изображений, **MediaScaleElement** предлагает несколько полезных свойств:

- **SubImages** — ссылается на коллекцию изображений, каждое из которых представлено как объект класса **MultiScaleSubImage**. Этот класс предоставляет меньше свойств, чем **MultiScaleImage** и не предоставляет событий. Поэтому, если хотите работать с изображениями в коллекции, то Вам придется перехватывать событие основного элемента, а потом высчитывать координаты, определять изображение из коллекции, на котором сейчас мышь, а затем выполнять задуманную операцию;
- **ViewportOrigin** — определяет координаты верхнего левого угла по отношению к **MultiScaleImage**. Это свойство доступно и в **MultiScaleSubImage**;
- **ViewportWidth** — возвращает ширину изображения в относительных единицах. Это свойство доступно и в **MultiScaleSubImage**.

Советую, если Вы пишите свой код по работе с Deep Zoom, обратиться уже к готовому коду. К сожалению, большинство шаблонов проектов, которые поставляются с Deep Zoom Composer, уже преобразованы в XAP, но один проект с исходным кодом все-таки есть. Его можно найти в следующем каталоге: **«C:\Program Files (x86)\Microsoft Expression\Deep Zoom Composer\Export Templates\Blend 3 Behaviors + Source»**.

Заключение

С одной стороны Deep Zoom представляет собой замечательную технологию, которая позволяет отображать высококачественные изображения с минимальной нагрузкой на сеть, а с другой — тут полностью отсутствуют какие-либо SDK, позволяющие делать преобразования изображений на сервере. Хотя многие умельцы приспособились вызывать Deep Zoom Composer в серверных сценариях, говорить о реализации расширяемых библиотек изображений тяжело. Несмотря на это, Deep Zoom позволяет реализовать множество интересных приложений, которые нельзя сделать на альтернативных технологиях. При этом Deep Zoom Composer достаточно несложная утилита, что позволяет создавать простейшие библиотеки изображений даже обычным пользователям.

Глава 15

ИНТЕГРАЦИЯ С SHAREPOINT 2010

Обзор возможностей

Вчера мне позвонили партнеры, с вопросом о том, как в существующие проекты на SharePoint 2007 добавить Silverlight-компоненты, отображающие видео. Речь шла о корпоративных сайтах, доступных внутри сети предприятия. После этого звонка я решил добавить главу, посвященную интеграции Silverlight на сайты под управлением SharePoint. Речь пойдет о SharePoint 2010. Это связано с тем, что к моменту выхода книги этот продукт будет более востребован. Кроме того, при разработке SharePoint 2010 на Silverlight был сделан отдельный акцент, в результате чего можно говорить о тесной интеграции этих двух продуктов.

Еще в самом начале книги я говорил, что в Silverlight было уделено особое внимание не только общедоступным приложения, работающим в Интернет, но и корпоративным приложения. Между тем, SharePoint представляет собой продукт, который призван решить две задачи: реализовать механизм управления контентом на сайтах организации, обеспечить поддержку процессов документооборота. И если на рынке CMS систем SharePoint 2010 имеет сильных конкурентов, большинство из которых Open Source (с открытым кодом, но никак не свободным — эта игра слов, для новичков), то в корпоративном секторе, среди систем документооборота, ему нет равных. Тут есть и тесная интеграция с Windows Workflow, и ASP.NET ориентированный интерфейс, а теперь еще и интеграция с Silverlight.

Давайте рассмотрим, какие возможности, связанные с Silverlight, представлены в SharePoint.

Тут можно сделать акцент на следующих возможностях:

- поддержка специальной Web-части (Silverlight Web Part), которая позволяет отобразить Silverlight-приложение;
- поддержка специального Silverlight элемента управления, который способен отображать видео на SharePoint сайтах;
- тесная интеграция с Office Web Application — набором офисных пакетов, работающих в Web. Именно в Web Microsoft теперь выпускает такие продукты, как Word, Excel, PowerPoint, работающие в браузере. С одной стороны, эти продукты можно использовать в Интернет, например, редактируя документы в Sky Drive. С другой стороны, эти продукты по-

ставляются вместе с SharePoint Server 2010 — платной версией SharePoint. Office Web Application могут работать, используя только возможности HTML и JavaScript, но если на машине пользователя установлен Silverlight, то он сможет воспользоваться дополнительными преимуществами интерфейса;

- поддержка специально разработанных библиотек (Client API), позволяющих взаимодействовать с SharePoint из Silverlight-приложения, скрывая детали от разработчика;
- взаимодействие с данными из списков SharePoint с помощью REST служб, то есть служб, обмен данными с которыми происходит по HTTP, но с сохранением связей между объектами. Подобный механизм работает благодаря тому, что все списки в SharePoint имеют точку доступа к соответствующей REST-службе. В свою очередь, Silverlight обладает мощными механизмами взаимодействия со службами по HTTP;
- поддержка развертывания Silverlight-приложений в режиме «песочницы». Подобный режим позволяет выполнять развертывание решений под SharePoint, используя только права администратора коллекции сайтов. Ранее, чтобы развернуть любое решение в SharePoint, необходимо было иметь права администратора всего сервера. Теперь чтобы добавить Silverlight-приложение на свой сайт, Вам не нужно быть администратором сервера. На самом деле, если Вы разворачиваете только .xap-файл, то можно обойтись и без прав владельца коллекции, но если Вы разворачиваете решение, содержащие набор фич (feature — вот так в SharePoint называется единица развертывания), то без прав владельца не обойтись;

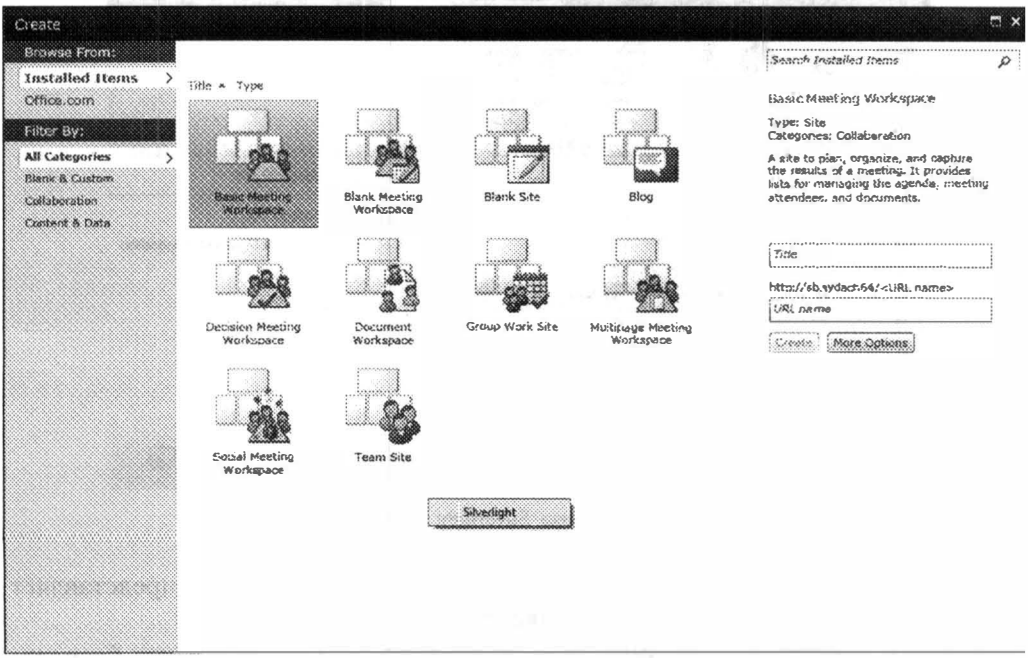


Рис. 15.1. Пример диалогового окна Create, реализованного в SharePoint на **Silverlight**

- тесная интеграция с существующим интерфейсом SharePoint 2010. Так если у пользователя установлен Silverlight, то многие диалоговые окна и элементы навигации приобретают дополнительные интерфейсные возможности, делая интерфейс более «приятным». В качестве примера можно рассмотреть интерфейс окна, позволяющего создавать сайты внутри коллекции (рис. 15.1).

Итак, сделав обзор всех аспектов интеграции Silverlight и SharePoint 2010 перейдем к детальному изучению некоторых из них.

Работа с Web-частями

Самый простой способ добавить Silverlight-приложение на страницу — это использовать уже встроенные Web-части. Для этого нужно выполнить два действия:

- Разместить существующий .xap-файл в один из списков SharePoint. Список или библиотека могут быть абсолютно произвольными. Я предпочитаю использовать библиотеки, определенные в шаблоне Global, например галерею master-страниц. Загрузив .xap-файл в выбранный список, обязательно скопируйте полный путь к нему (например, http://sbaydach64/_catalogs/masterpage/SilverLight/DragAndDrop_Chapter0.xap);
- Перейдите на страницу, где Вы планируете разместить Silverlight-приложение, войдите в режим редактирования и выберите Silverlight Web-часть, указав Url на Ваш .xap-файл:

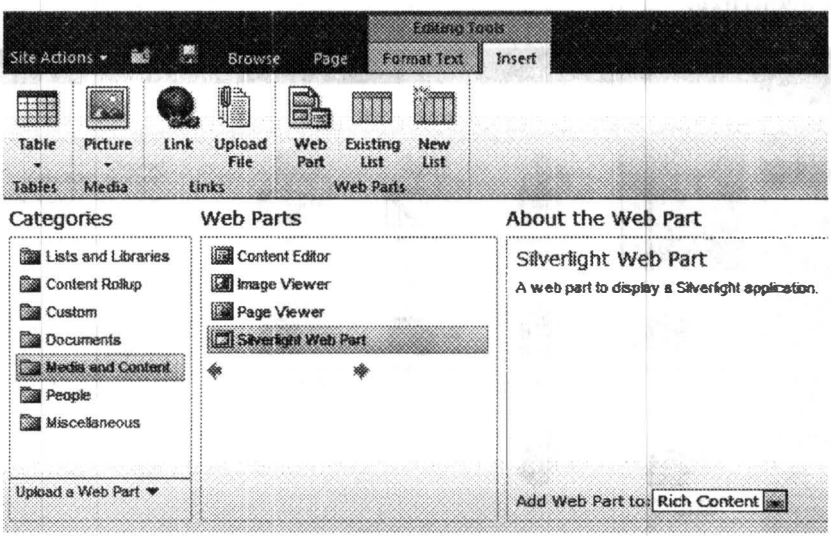


Рис. 15.2. Вставка Silverlight Web-части

Как видно, разместить готовое Silverlight-приложение не представляет трудностей даже для обычного пользователя.

Когда я говорю о SharePoint 2010, то обычно подразумеваю как бесплатную версию продукта — SharePoint Foundation 2010, так и платную —

SharePoint Server 2010. Silverlight web-часть присутствует в обоих версиях продукта. А вот Web-часть, отображающая видео, присутствует только в платной версии.

Фактически, Silverlight Media Web-часть представляет собой готовый медиа плейер, позволяющий принимать имя видео файла в одном из списков SharePoint. Эта web-часть может быть использована пользователем для размещения видео на любой из страниц. Кроме этого, она используется в библиотеках, предназначенных для хранения видео, изображений и другой информации.

Несмотря на то, что Silverlight Media Web-часть отсутствует в бесплатной версии, ее легко реализовать самостоятельно, но для этого Вам понадобится Visual Studio 2010.

Развертывание Silverlight-приложения с помощью Visual Studio 2010

Попробуем использовать Visual Studio 2010 для создания решения для SharePoint, включающего в себя web-часть, содержащую Silverlight-приложение.

Поскольку создание Web-части — дело тривиальное, будем использовать шаблон решения, предназначенного для развертывания в «песочнице» SharePoint 2010. Для этого необходимо создать проект на основе шаблона **Empty SharePoint Project** и выбрать адрес сайта, где Вы планируете провести тестирование Web-части, а также отметить опцию **Deploy as a sandbox solution**.

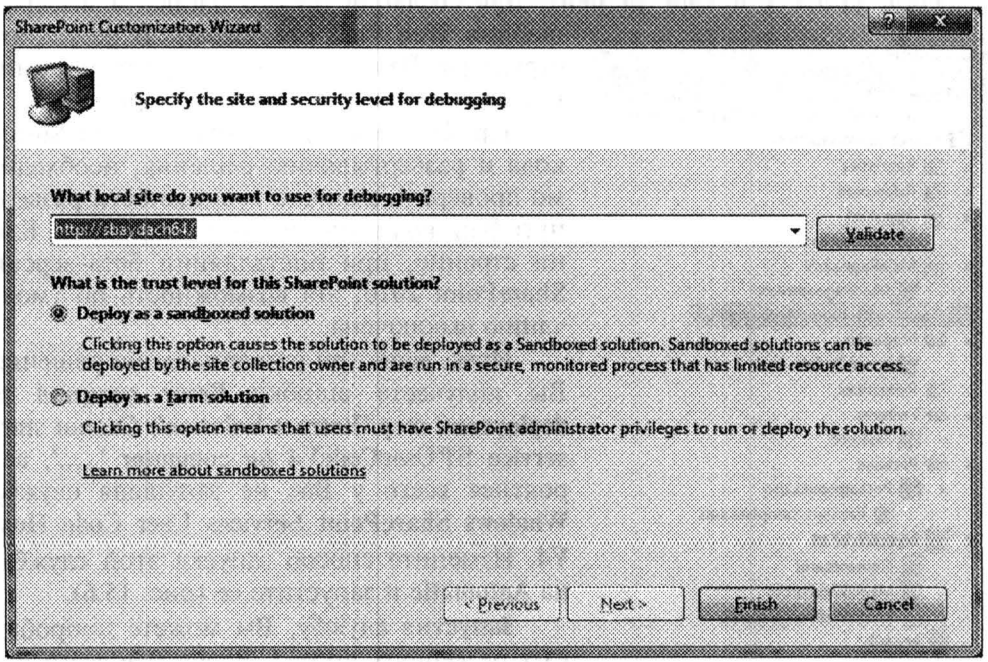


Рис. 15.3. Создание проекта для SharePoint 2010

В созданное решение добавьте еще один проект на основе шаблон **Silverlight Application** (у меня проект называется SharePointMediaWebPart). При этом снимите галочку **Host the Silverlight application in a new Web Site** (он нам не понадобится):

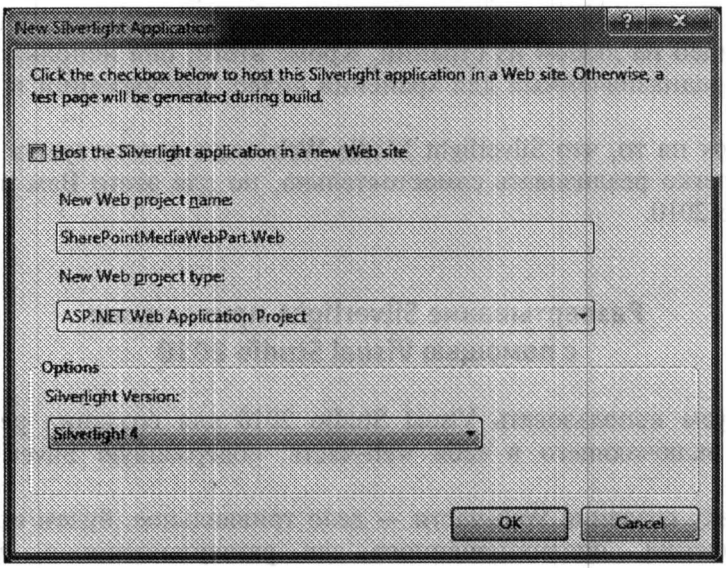


Рис. 15.4. Создание Silverlight-приложения

На последнем шаге при подготовке каркаса нашего решения, добавим к SharePoint проекту новый элемент. Для создания нового элемента выберем шаблон **Web Part** (не Visual). В результате структура Вашего проекта будет напоминать ту, которая отображена на рис. 15.5.

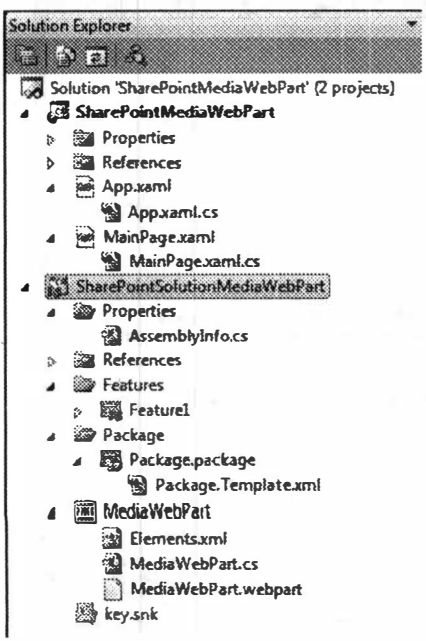


Рис. 15.5. Структура проекта

Прежде чем приступить к написанию кода и развертыванию решения, необходимо проверить наличие возможности развернуть Ваш код в «песочницу» SharePoint. Как ни странно, при инсталляции бета-версии SharePoint 2010, эта возможность по умолчанию выключена.

Итак, если при развертывании решения Вы получаете ошибку **Error occurred in deployment step 'Retract Solution': Cannot start service SPUserCodeV4 on computer '.....'**, вероятнее всего у Вас не запущена служба **Windows SharePoint Services User Code Host V4**. Измените способ запуска этой службы на **Automatic** и запустите ее (рис. 15.6).

Запустив службу, Вы можете попробовать развернуть проект. Вероятнее всего Вы получите следующую ошибку: **Error occurred**

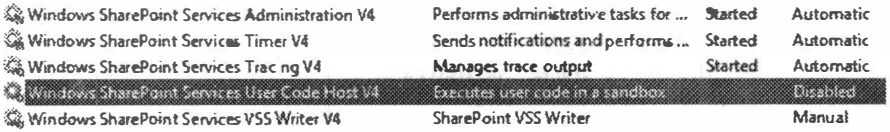


Рис. 15.6. Запуск службы

in deployment step 'Activate Features': This feature cannot be activated at this time. The contents of the feature's solution requires the Solution Sandbox service to be running. Чтобы, наконец, развернуть наше решение, необходимо запустить утилиту администрирования SharePoint 2010 и запустить **SharePoint Foundation User Code Service** службу и там. Хотя последнее действие скорее активирует возможность, чем запускает службу. Для запуска службы Вам необходимо перейти в раздел **System Settings->Manage Services on Server** и нажать **Start**.

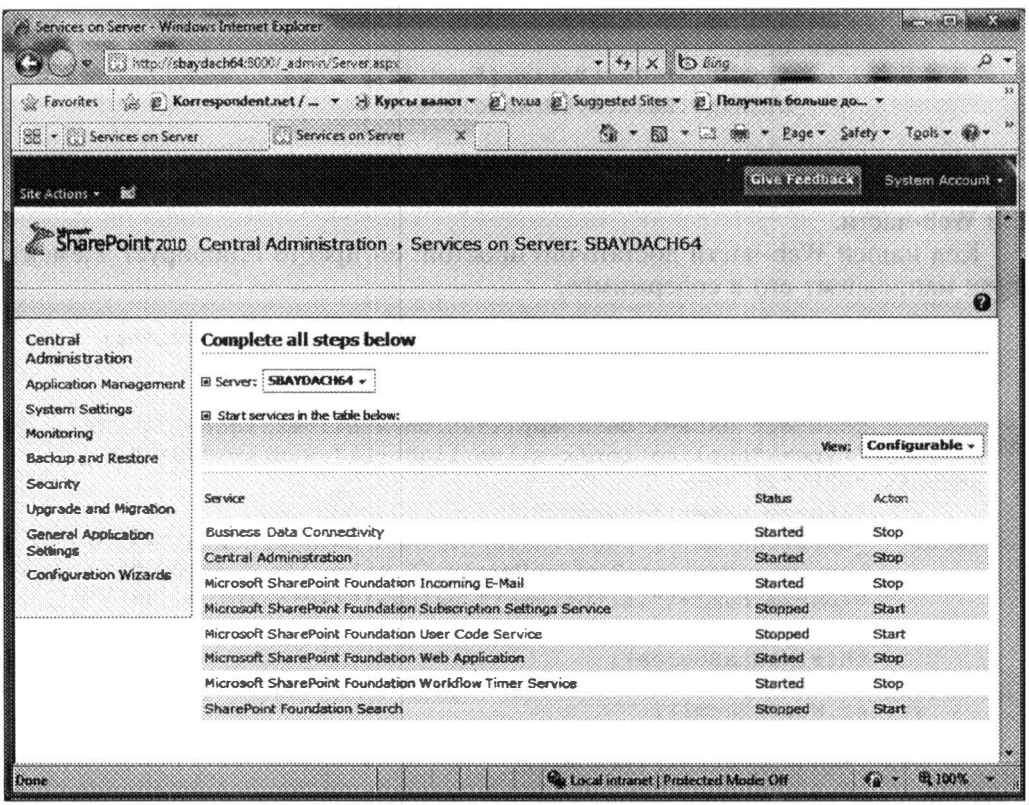


Рис. 15.7. Активация службы с помощью утилиты администрирования

Запустив все необходимые службы, можно приступить к написанию кода.

Первым делом реализуем Silverlight-приложение, заполнив его содержимое простым **MediaElement** (мы не будем создавать сложный интерфейс).

```
<UserControl x:Class="SharePointMediaWebPart.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
```

```
    <Grid x:Name="LayoutRoot" Background="White"
HorizontalAlignment="Left">
        <MediaElement Name="myMedia"
AutoPlay="True"></MediaElement>
    </Grid>
</UserControl>
```

Выбирать имя отображаемого видео будем, используя конфигурационный параметр элемента **object**, который добавляет встраиваемый компонент Silverlight на страницу. Поэтому реализуем код, выбирающий параметр с ключом media:

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    this.RootVisual = new MainPage();

    ((MainPage)this.RootVisual).myMedia.Source =
        new Uri(
            this.Host.InitParams["media"],
            UriKind.RelativeOrAbsolute);
}
```

Silverlight-приложение полностью готово. Перейдем к разработке созданной Web-части.

Код нашей Web-части достаточно простой: он просто генерирует элемент object ивписывает его в содержимое:

```
protected override void RenderContents(HtmlTextWriter writer)
{
    string html = String.Format(
        "<object data=\"data:application/x-silverlight-2,\" "+
        "type=\"application/x-silverlight-2\" width=\"90%\"
height=\"90%\">" +
        "<param name=\"source\"
value=\"_catalogs/masterpage/SharePointMediaWebPart.xap\"/>" +
        "<param name=\"minRuntimeVersion\" value=\"4.0.41108.0\" />" +
        "<param name=\"initparams\" value=\"media={0}\" />" +
        "</object>",
        this.MediaSource);

    writer.Write(html);

    base.RenderContents(writer);
}
```

Кроме этого, необходимо определить свойство MediaSource, которое будет задаваться редактором сайта:

```
private string _mediaSource = "";

[WebBrowsable(true),
WebDescription("Select .wmv file from a list"),
WebDisplayName("Media Source"),
Personalizable(PersonalizationScope.User)]
```

```
public string MediaSource
{
    get
    {
        return _mediaSource;
    }
    set
    {
        _mediaSource = value;
    }
}
```

Собственно говоря, это весь код. Остается настроить проект таким образом, чтобы он включал .xap-файл приложения и выполнял его развертывание в один из существующих списков (я выбрал список с эталонными страницами).

Чтобы включить .xap-файл в решение SharePoint 2010, войдите в свойства Web-части (MediaWebPart) и установите ссылку на выходной файл нашего SilverLight-проекта. Для этого используется окно **Project Output Reference**:

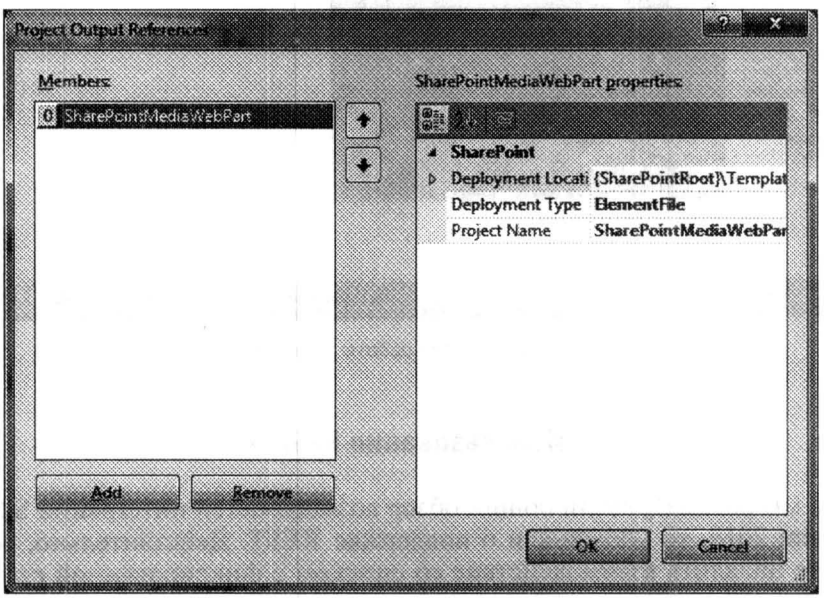


Рис. 15.8. Добавление .xap-файла в пакет решения

Имея .xap-файл внутри нашего пакета, остается развернуть его внутри списка с эталонными страницами. Для этого отредактируйте файл elements.xml добавив внутрь элемента **Elements** следующий код:

```
<Module Name="MediaWebPartXap" Url="_catalogs/masterpage">
  <File Path="MediaWebPart\SharePointMediaWebPart.xap"
     Url="SharePointMediaWebPart.xap" Type="GhostableInLibrary">
  </File>
</Module>
```

Проект готов к развертыванию.

Попробуйте выполнить развертывание и создать произвольный список, а затем разместить в него видео-файл. Добавьте созданную Web-часть на любую из страниц и укажите ссылку на видео-файл в вашем списке, используя свойство Media Source в окне настроек Web-части. Если Вы все сделали верно, то на экране отобразится видео:

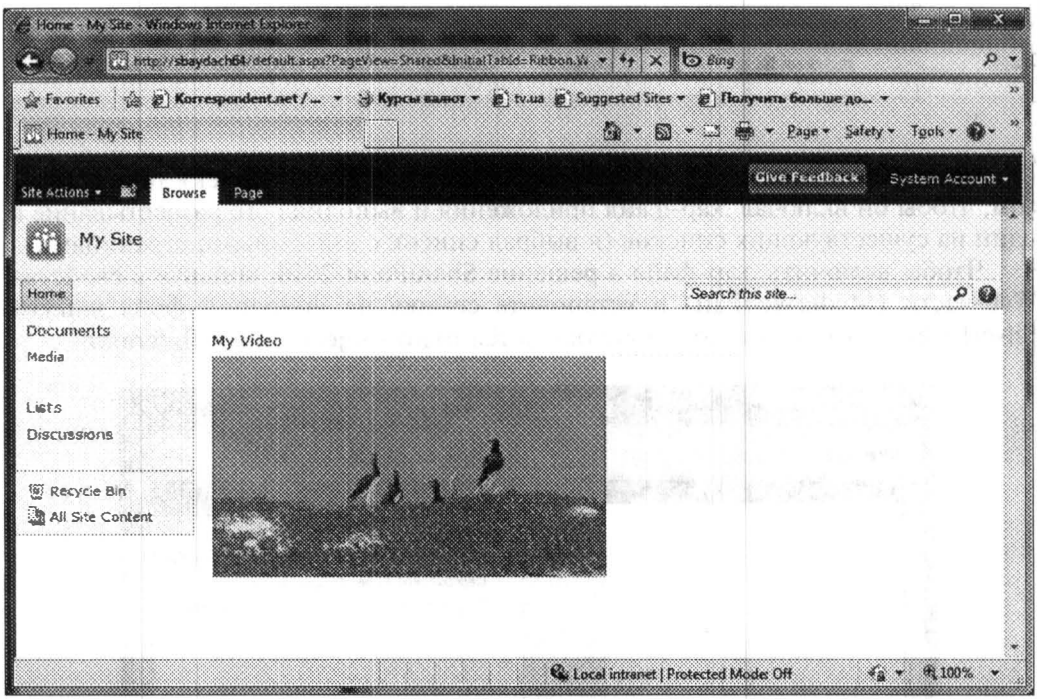


Рис. 15.9. Результат работы программы

Использование REST

Когда мы рассматривали общий обзор возможностей интеграции Silverlight и SharePoint 2010, то упоминали о поддержке REST. Действительно, если Вы планируете наладить взаимодействие со списком с фиксированной схемой, то можете получить доступные списки на заданном сайте, обратившись к службе по следующему адресу (аналогичному): http://localhost/_vti_bin/ListData.svc/. Получить данные из конкретного списка (например, Media), можно, используя следующую ссылку: http://sbaydach64/_vti_bin/ListData.svc/Media. Если Вы хотите получить конкретный элемент из списка, то ссылка будет выглядеть аналогично этой: http://sbaydach64/_vti_bin/ListData.svc/Media(1). Во всех случаях данные возвращаются в формате Atom, а, следовательно, базируются на XML.

Попробуем переписать предыдущий пример так, чтобы Web-часть, отображающая видео, позволяла задавать имя библиотеки (а не файла), содержащей видеоматериалы. При этом изменим Silverlight-приложение таким образом, чтобы оно позволяло выбрать любой видео файл из библиотеки в режиме реального времени. То есть, попробуем реализовать примитивную видео библиотеку.

Для начал, реализуем новый интерфейс для Silverlight-приложения:

```
<UserControl x:Class="SharePointMediaWebPart.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Loaded="UserControl_Loaded">
    <StackPanel x:Name="LayoutRoot" Background="White"
        Orientation="Horizontal">
        <ListBox Name="myList"
SelectionChanged="myList_SelectionChanged">
        </ListBox>
        <MediaElement Name="myMedia"
AutoPlay="True"></MediaElement>
    </StackPanel>
</UserControl>
```

Тут мы изменили главный контейнер, в который добавили **ListBox**, необходимый для выдачи списка файлов. Кроме того, мы определили два обработчика для событий **Load** и **SelectionChanged**. Первый обработчик будет выбирать данные из указанного списка и заполнять **ListBox**. Второй обработчик позволит устанавливать новый источник для медиа элемента.

Перейдем к реализации кода.

Внесем изменения в конструктор класса MainPage, добавив код, выбирающий параметр с именем списка из коллекции параметров для встраиваемого компонента (удалите аналогичный код из **Application_Startup** — в этом примере его там держать не удобно).

```
private String curList="";

public MainPage()
{
    InitializeComponent();

    if (App.Current.Host.InitParams.ContainsKey("media"))
    {
       curList = App.Current.Host.InitParams["media"];
    }
}
```

Реализация обработчика события **SelectionChanged** также будет простой:

```
private void myList_SelectionChanged(
   object sender, SelectionChangedEventArgs e)
{
    myMedia.Source = new
       Uri(String.Format("http://localhost/{0}/{1}",
          curList, myList.SelectedValue.ToString()),
       UriKind.Absolute);
}
```

(можно было бы использовать контекст, чтобы выбрать адрес сайта из коллекции)

Тут мы формируем ссылку на файл, лежащий в нашей библиотеке, и устанавливаем источник для **MediaElement**.

Давайте рассмотрим код последнего обработчика:

```
private void UserControl_Loaded(object sender, RoutedEventArgs e)
{
    if (curList.Length>0)
    {
        WebClient client = new WebClient();
        client.OpenReadCompleted +=
           new OpenReadCompletedEventHandler(
              client_OpenReadCompleted);

        client.OpenReadAsync(
           new Uri(string.Format(
              "http://sbaydach64/_vti_bin/ListData.svc/{0}",
              curList),
           UriKind.Absolute));
    }
}
```

Тут мы устанавливаем асинхронное соединение со службой, используя имя списка, передаваемого в параметрах. Как только данные будут получены, вызовется метод client_OpenReadCompleted, который и будет реализовывать всю работу по обработке XML и заполнению **ListBox**.

Прежде чем перейти к написанию кода client_OpenReadCompleted, создайте в SharePoint библиотеку с любым именем (например, Media), и загрузите в нее несколько медиа файлов. После этого выполните команду, аналогичную этой:

```
http://localhost/_vti_bin/ListData.svc/Media
```

Ниже я привожу часть XML кода, который я получил, выполняя команду выше:

```
  <entry m:etag="W/&amp;quot;4&amp;quot;">
    <id>http://sbaydach64/_vti_bin/ListData.svc/Media(1)</id>
    <title type="text">Test 1</title>
    <updated>2010-02-10T16:12:06+02:00</updated>

. . . . . .

<d:ContentTypeID>0x0101009110D2200BDC0F429ED6372AF0A68C01</d:Conten
tTypeID>
      <d:ContentType>Document</d:ContentType>
      <d:Created
m:type="Edm.DateTime">2010-02-10T16:10:13</d:Created>
      <d:CreatedByID m:type="Edm.Int32">1073741823</d:CreatedByID>
      <d:Modified
m:type="Edm.DateTime">2010-02-10T16:12:06</d:Modified>
      <d:ModifiedByID
m:type="Edm.Int32">1073741823</d:ModifiedByID>
      <d:CopySource m:null="true"></d:CopySource>
```

```
      <d:ApprovalStatus>0</d:ApprovalStatus>
      <d:Path>/Media</d:Path>
      <d:CheckedOutToID m:type="Edm.Int32"
m:null="true"></d:CheckedOutToID>
      <d:Name>Test 1.wmv</d:Name>
      <d:VirusStatus>26246026</d:VirusStatus>
      <d:IsCurrentVersion
m:type="Edm.Boolean">true</d:IsCurrentVersion>
      <d:Owshiddenversion m:type="Edm.Int32">4</d:Owshiddenversion>
      <d:Version>1.0</d:Version>
      <d:Title>Test 1</d:Title>
    </m:properties>
  </entry>
```

Как видно, имя файла содержится в поле **d:Name**. Поскольку в этом случае нам другие данные и не нужны, воспользуемся простым объектом XmlReader для выбора нужной информации. Вот как будет выглядеть наш код:

```
void client_OpenReadCompleted(object sender,
    OpenReadCompletedEventArgs e)
{
    XmlReader reader = XmlReader.Create(e.Result);

    while (reader.Read())
    {
        if (reader.NodeType==XmlNodeType.Element)
        {
                if (reader.Name.Contains("Name"))
                {
                    reader.Read();

                    myList.Items.Add(reader.Value);

                }
                break;
        }
    }
    myList.SelectedIndex = 0;
}
```

Получая XML документ в качестве потока, мы используем **XmlReader** для поиска нужного элемента, у которого выбираем значение.

Теперь можно откомпилировать и развернуть данный пример. Указав в качестве параметров Web-части?? имя созданного списка с видео, Вы сможете увидеть работу приложения (рич. 15.10).

Если Вы разрабатываете более сложный компонент, то для взаимодействия с XML данными можно использовать один из следующих подходов:

- используйте прокси классы, созданные Visual Studio — для этого добавьте ссылку на службу с помощью Add Service Reference. Прокси классы будут сгенерированы автоматически. В нашем примере они не очень помогут, так как мы не знаем имя списка;

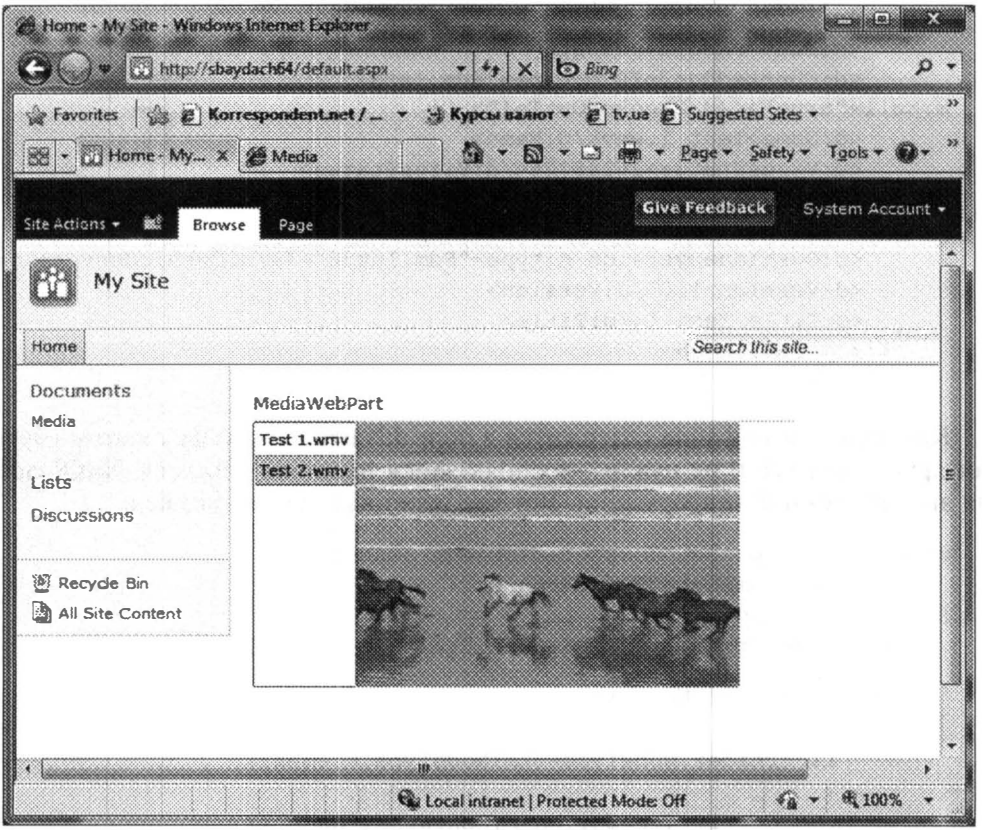


Рис. 15.10. Результат работы приложения

- используйте механизмы сериализации (уж не знаю, как это правильно называть по-русски) — сложно, но эффективно. Зная, что в нашем списке есть определенные поля, мы могли бы не делать привязку к конкретному списку;
- используйте XLinq — нужно подключать библиотеки из SDK, а в данном случае это не имеет смысла;
- используйте специальный класс из SDK (SyndicationFeed). Опять же, нужно подключать дополнительную библиотеку. Кроме того, этот класс отказался работать с контентом, возвращаемым SharePoint (хотя должен работать с Atom).

Поддержка Client API

Напоследок в теме по интеграции Silverlight и SharePoint 2010 рассмотрим возможность использования Client API, поставляемого с SharePoint.

В отличие от служб REST, Client API позволяет делать практически все: взаимодействовать с любыми объектами, создавать сайты, управлять процессами.

Чтобы использовать Client API для Silverlight, необходимо подключить две сборки: Microsoft.SharePoint.Client.SilverLight.dll и Microsoft.SharePoint.Client.SilverLight.Runtime.dll. Первая сборка отвечает за классы, описывающие объекты SharePoint, а вторая сборка берет на себя задачи, связанные с коммуникациями. Обе сборки находятся в папке C:\Program Files\Common Files\Microsoft Shared\Web Server Extensions\14\TEMPLATE\LAYOUTS\ClientBin.

Замечание. В Visual Studio 2010 beta 2 существует странная ошибка. Если Вы попытаетесь добавить описанные сборки по указанному пути, то Visual Studio сгенерирует ошибку. Поэтому скопируйте данные сборки в директорию Вашего проекта.

Модифицируем код нашего примера так, чтобы он использовал Client API. Приведу сразу готовый код:

```
ListItemCollection items=null;

private void UserControl_Loaded(object sender, RoutedEventArgs e)
{
    ClientContext context = new ClientContext("http://sbaydach64");

    Web myWeb=context.Web;

    List myListSP = myWeb.Lists.GetByTitle(curList);

    items = myListSP.GetItems(
        CamlQuery.CreateAllItemsQuery());

    context.Load(items);
    context.ExecuteQueryAsync(client_Completed, null);
}

private void client_Completed(object sender,
    ClientRequestSucceededEventArgs e)
{
    System.Windows.Deployment.Current.Dispatcher.BeginInvoke(
    delegate()
    {
        foreach (ListItem item in items)
        {
            myList.Items.Add(item.FieldValues["FileLeafRef"]);
        }

        myList.SelectedIndex = 0;
    }
    );
}
```

Основной класс в коде выше — это **ClientContext**, который является единственным механизмом для получения доступа ко всем объектам SharePoint. Создав объект этого класса, разработчику требуется сформировать запрос. При формировании запроса, Вы можете ссылаться на любые свойства объектов, при условии, что не собираетесь их явно использовать. На этом этапе реальные свойства объектов еще не доступны.

Сформировав запрос, необходимо подготовить его к исполнению. Для этого и служит метод **Load**. Этот метод подготавливает контекст к чтению всех необходимых объектов, которые используются для получения желаемого результата.

Наконец, для загрузки всех объектов необходимо выполнить метод **ExecuteQueryAsync**, который принимает в качестве параметров ссылку на метод, обрабатывающий результат работы запроса (и метод, вызывающийся при сбое).

В отличие от **OpenReadAsync**, класса **WebClient**, метод **ExecuteQueryAsync** инициирует вызов метода, обрабатывающего результаты асинхронного вызова НЕ в интерфейсном потоке. Поэтому, чтобы обновить интерфейс, необходимо инициировать вызов еще одного метода, но в интерфейсном потоке, предварительно получив и обработав данные. Этим и занимается объект **Dispatcher**.

Даже на первый взгляд использование Client API значительно проще, чем работа с REST. Использование клиентских библиотек обладает гибкостью и универсальностью. При этом нужно помнить, что размер Вашего приложения увеличится примерно на 400 килобайт — таков размер дополнительных сборок.

Заключение

Как видно, Microsoft делает на Silverlight достаточно большую ставку. Это позволило привлечь достаточно ресурсов, чтобы позволить Silverlight использоваться не только на новых сайтах, но и в корпоративных Web-приложениях. А раз мы говорим о корпоративном секторе, то стоит вспомнить о возможности работы Silverlight-приложений вне браузера (с повышением полномочий). Эта возможность позволяет не просто устанавливать Silverlight-приложения на сторону клиента, но и обеспечить доступ к кэшу с данными, даже в отключенном режиме, что совсем не реализовано в SharePoint.

Нужно отметить, что в SharePoint 2007 также можно было интегрировать Silverlight-приложения, но для этого нужно было затратить больше усилий. Конечно, ни о каком клиентском API для SharePoint 2007 нельзя было и мечтать.

Глава 16

ВВЕДЕНИЕ В MICROSOFT EXPRESSION STUDIO

Обзор продуктов

Как Вы смогли убедиться, Silverlight предоставляет большие возможности по разработке насыщенных приложений. Но если Вы планируете написать приложение, которое действительно понравится пользователям, то Вам не обойтись без дизайнера. Подбор цветовой гаммы, дизайн элементов управления, создание сложных изображений, — все это разработчику не по силам. Но если мы привлекаем дизайнера, то он, скорее всего, не сможет использовать Visual Studio даже для создания интерфейса, не говоря уже о сложных изображениях. Во-первых, Visual Studio не обладает мощным инструментарием создания векторной графики, а во-вторых, дизайнер должен иметь инструменты, которые специально адаптированы для него и не требуют прямого редактирования XAML (как и любого кодирования).

Microsoft Expression Studio — это специальный пакет, который был выпущен в первую очередь для дизайнеров. Хотя его могут использовать и разработчики, тут сделано все, чтобы дизайнер почувствовал себя в родном окружении.

Первая версия **Expression Studio** вышла одновременно с выходом Windows Presentation Foundation и была направлена на работу с WPF. Сейчас доступна третья версия этого продукта, которая поддерживает работу не только с WPF, но и с Silverlight 3. Если Вы планируете создавать приложения, используя Silverlight 4, то разработчикам доступен Expression Blend 4 Preview. Именно этот продукт больше всего связан с созданием интерфейса и требует перехода на новую версию с выходом новой версии Silverlight.

Проведем обзор тех продуктов, которые входят в пакет Expression Studio.

Expression Encoder.

Поскольку одно из достоинств Silverlight — это возможность отображать видео различных форматов, включая видео в формате Smooth, было бы странно, если бы Microsoft не выпустила продукт, позволяющий работать с видео. Именно **Expression Encoder** позволяет преобразовывать видео из одного формата в другой, вырезать отдельные части записанного видео, осуществлять преобразование «живого» видео. А с появлением **Expression Encoder 3** тут появилась дополнительная утилита **Screen Capture**, способная осуществлять за-

пись действий пользователя в любой области экрана. Таким образом, Вы мо жете не просто обрабатывать видео, но и создавать различного рода веб-кас ты. Детальный обзор Expression Encoder ожидает нас в следующем разделе;

Expression Web.

Чтобы сделать набор продуктов по созданию Web-приложений полным, пакет Expression Studio комплектуется таким продуктом, как Expression Web Этот продукт напрямую не связан с Silverlight и предназначен для разработчи ков и дизайнеров, которые занимаются версткой страниц. Данный продукт поддерживает создание и редактирование стилей, HTML страниц, ASP.NET страниц и даже PHP страниц. Таким образом, он полностью подходит для любого из Web разработчиков. Кроме того, в состав продукта входит утилита Expression Web 3 Super Preview, которая позволяет просматривать, сравнивать и отлаживать страницы для различных браузеров. Это позволит гарантировать работоспособность Ваших приложений в любых ситуациях. Этот продукт выходит за рамки данной книги и не будет рассматриваться дальше.

Expression Design.

Следующий продукт напрямую связан с Silverlight и предназначен лишь для дизайнеров — это **Expression Design 3**. Несмотря на то, что и Visual Studio и Expression Blend позволяют создавать интерфейсы Silverlight- и WPF-прило жений, тут нет никаких механизмов для рисования. **Expression Design 3** — это продукт, который предназначен только для рисования. Причем он способен работать как с векторной, так и с растровой графикой. Expression Design спо собен преобразовывать созданное изображение в различные форматы. Один из таких форматов — это XAML. Таким образом, если Вы задумали написать приложение, например шахматы, где требуется нарисовать множество различ ных элементов, Вы используете **Expression Design**, а затем переносите XAML уже в Ваш интерфейс.

Expression Blend.

Последний, но самый важный продукт — это **Expression Blend**. Как Вы уже догадались, именно этот продукт позволяет создавать интерфейсы, редак тируя XAML файлы в визуальном дизайнере. Причем Expression Blend работа ет с проектами в формате Visual Studio, что позволяет разработчику редакти ровать код, а дизайнеру — интерфейс. Весь процесс происходит одновремен но, а дизайнеры и разработчики работают как единая команда.

Рассмотрев состав пакета Microsoft Expression Studio, остановимся на не которых возможностях этого набора продуктов, применимых в Silverlight.

Работа с Expression Encoder

Преобразование видео

Рассмотрим Expression Encoder применительно к преобразованию фай лов из одного формата в другой. Нужно отметить, что Expression Encoder способен открывать практически любые форматы (есть проблемы при нали

чии нескольких звуковых дорожек), кодеки для которых присутствуют на компьютере.

На первом этапе необходимо добавить один или несколько файлов в Expression Encoder, что можно сделать с помощью команды **File->Import**, и выбрать нужные файлы.

Все файлы будут располагаться в окне **Media Content**, где отображаются не только их названия, но и длительность, размер и разрешение.

Рис. 16.1. Панель Media Content

Сразу над панелью **Media Content** отображается панель инструментов по работе с активным видео. Тут можно выполнять предварительный просмотр видео, добавлять другие видеофайлы в конец или начало видео, а также вырезать нежелательные куски.

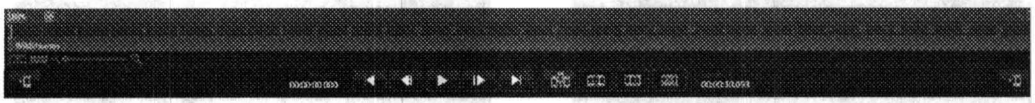


Рис. 16.2. Панель инструментов по работе с видео

Если Вы хотите вырезать любую часть видео, то первым делом необходимо установить специальные маркеры редактирования. Для этого нужно перевести ползунок в желаемое место начала или конца выделения и вызвать контекстное меню, где выбрать пункт **Add Edit**. Выделив нужный участок, активируйте контекстное меню и вызовите команду **Remove Clip**. Естественно, что в исходном файле никаких изменений произведено не будет, но в выходном файле удаленного фрагмента не будет.

С помощью контекстного меню можно также добавить маркер, который также будет встроен в выходной файл.

Как только видео готово к преобразованию, необходимо установить параметры выходного файла. Если у Вас несколько файлов, то для одновременной установки параметров их необходимо выделить в одну группу. Прежде чем переходить к тонкой настройке, обратите внимание на окно Presets, содержащее профили для наиболее популярных настроек. Если один из профилей Вам подходит, то просто выберите его и нажмите кнопку **Apply**. После этого Вы сможете изменить настройки (рис. 16.3).

Для более тонкой настройки необходимо перейти на панель **Encode** (рис. 16.4).

Тут Вы сможете оптимизировать параметры видео и аудио.

Остается нажать кнопку **Encode** и дождаться результата. По умолчанию результат складируется в папку **<Мои Документы>\Expression\Expression Encoder\Output**.

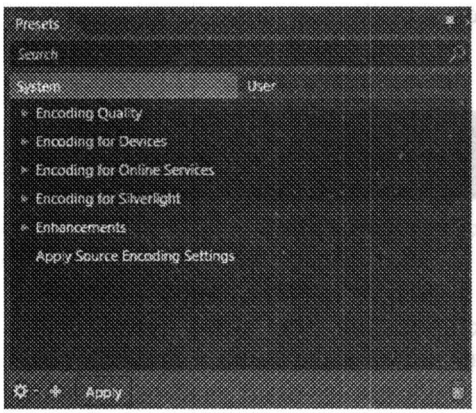


Рис. 16.3. Окно Presets

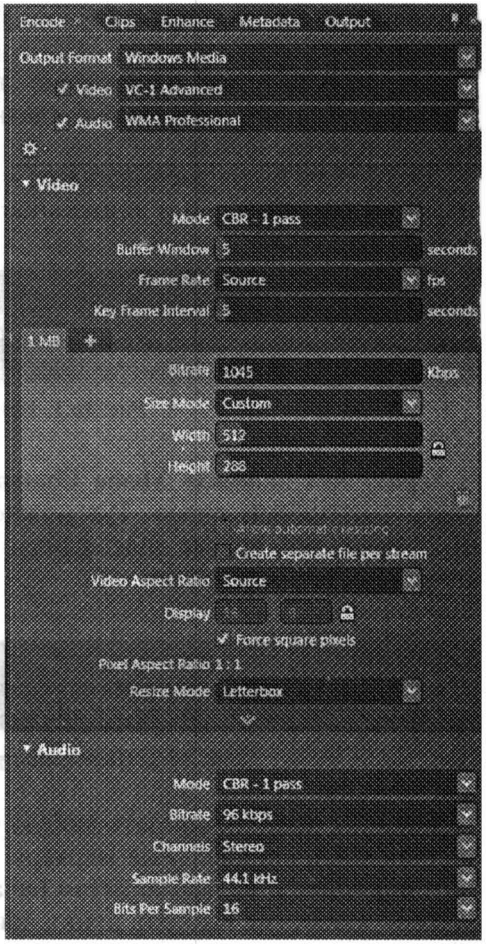


Рис. 16.4. Панель Encode

Использование встроенных шаблонов

Если Вы хотите не просто преобразовать видео из одного формата в другой, а сразу же получить готовое Silverlight-приложение, то это легко сделать с помощью одного из шаблонов, поставляемых вместе с **Expression Encoder**.

Чтобы воспользоваться одним из шаблонов, достаточно перейти на вкладку **Output** и выбрать нужный шаблон из списка **Template** (рис. 16.5).

Среди шаблонов присутствуют готовые Silverlight-плейеры как для работы с работы с видео с прогрессивной загрузкой, так и для видео в формате Smooth (в этом случае отображаются дополнительные параметры, характерные для Smooth).

Используя вкладку Output, можно не просто создать видео вместе с готовым плейером, но и развернуть их в одну из директорий Web-приложения.

Самое интересное то, что все представленные шаблоны поставляются вместе с исходным кодом, который Вы можете использовать для более универсаль-

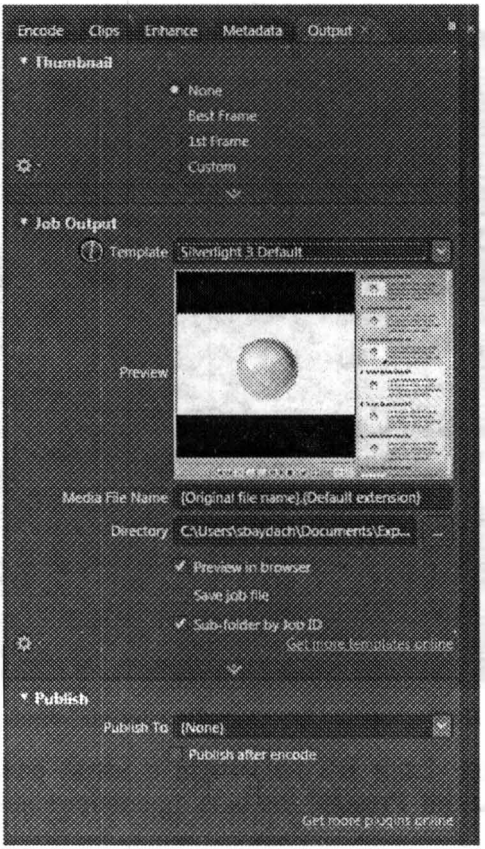


Рис. 16.5. Выбор шаблона медиа плейера для создания Silverlight-приложения

ного сценария отображения видео в Вашем проекте. Все шаблоны доступны в каталоге: **C:\Program Files (x86)\Microsoft Expression\Encoder 3\Templates\en**

Использование Expression Encoder для трансляции живого видео

Чтобы перейти в режим трансляции «живого» видео, щелкните на кнопку **Live Encoding**, которая находится сразу под панелью **Media Content**. После этого действия Expression Encoder перейдет в режим **Live Encoding** и будет полностью готов для создания потокового видео и публикации его на Media Service (рис. 16.6).

Первое, что необходимо задать в режиме Live Encoding, — это входящий файл или поток. Обычно используется поток, который может поступать с записывающего устройства или со спутника. В любом случае, за получение сигнала отвечают видео и аудио устройства, а задача Expression Encoder — получив сигнал, преобразовать его в потоковое видео нужного формата. Однако если Вы задумали трансляцию записанного видео (например, на DVD), то в качестве входного параметра следует задать лишь файл или файлы.

Задав параметры, связанные с входящим потоком, можно перейти к настройке исходящего потока. Для этих целей на вкладке **Encoding** существует

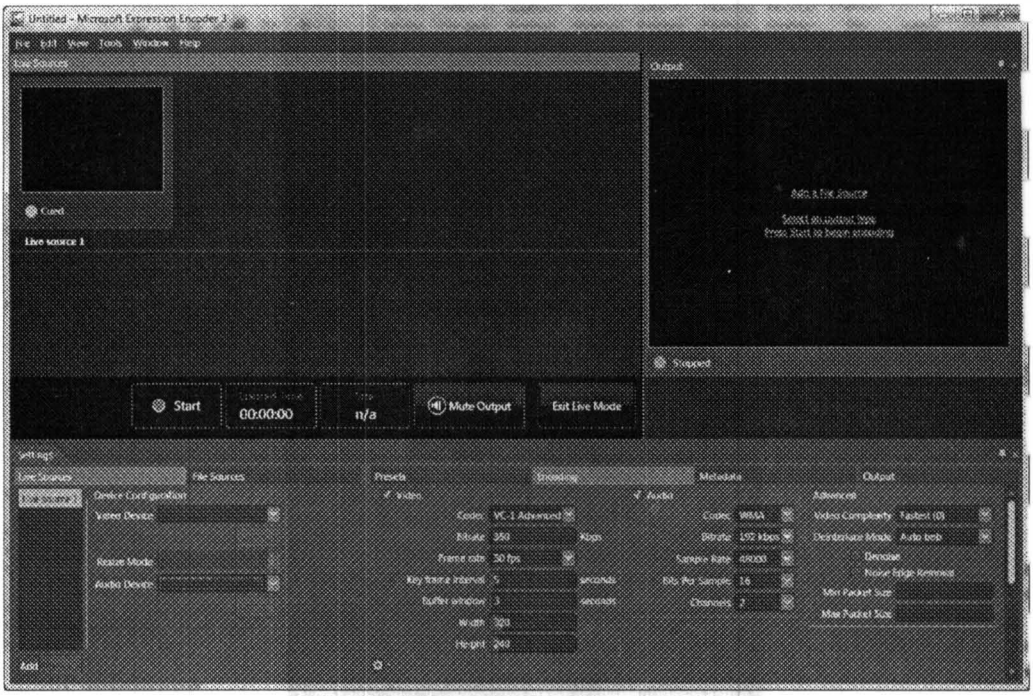


Рис. 16.6. Использование Expression Encoder для «живого» видео

множество параметров. Если Вы не хотите с ними разбираться, то можно воспользоваться вкладкой **Presets**, где есть предопределенный набор конфигураций.

На последнем этапе необходимо настроить способ трансляции потока. Для этого на вкладке Output необходимо выбрать одну из двух опций: **Broadcast** или **Publishing Point**.

Первая опция позволяет запустить передачу потока на один из локальных портов компьютера, где происходит кодирование. Данная возможность хороша, если Вы хотите быстро наладить трансляцию внутри корпоративной сети, ограничившись 50 соединениями. В теории эту трансляцию можно передавать и на Media Service, но для этого существует опция **Publishing Point**, которая позволяет транслировать поток на Media Service, который замечательно справляется с трансляциями видео в Internet.

Остается нажать кнопку Start. Если Вы выбрали опцию Broadcast, то сможете запустить предварительный просмотр, с помощью Silverlight-приложения, установленного вместе с **Expression Encoder**.

Захват изображения и звука

В завершение обзора **Expression Encoder** рассмотрим утилиту, которая позволяет выполнять захват видеоизображения с экрана монитора. Это специальная утилита Expression Encoder 3 Screen Capture.

Внешний вид этой утилиты достаточно прост. Тут всего четыре кнопки, две из которых включают и выключают запись звука и изображения с веб-ка-

меры, третья кнопка позволяет перейти к окну настроек, а последняя приступить к записи.

Рис. 16.7. Expression Encoder Screen Capture

Хочу обратить внимание на то, что в моем случае кнопка, позволяющая включить микрофон — активна, а кнопка, позволяющая включить веб-камеру — не активна. Но это вовсе не означает, что в данной конфигурации звук будет записываться. Чтобы включить запись звука, необходимо щелкнуть на кнопке с микрофоном. Кнопка изменит цвет, а рядом появится регулятор громкости. Вот в такой конфигурации будет вестись запись и изображения с экрана, и звука.

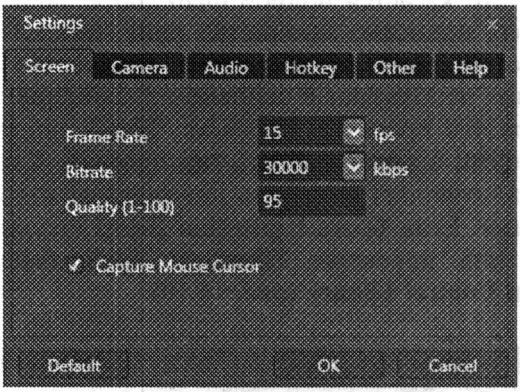


Рис. 16.8. Окно настроек Screen Capture

Чтобы установить параметры записи, достаточно щелкнуть на третью кнопку в главном окне утилиты, и на экране отобразится окно настроек.

Тут легко выбрать нужный микрофон, камеру, установить горячие клавиши и задать параметры записи.

Установив необходимые параметры, можно приступать к записи. Для этого нужно нажать последнюю кнопку в окне утилиты и выбрать регион на экране для записи. В появившемся окне Вы также можете воспользоваться уже готовыми настройками и выбрать Full Screen для записи всего экрана.

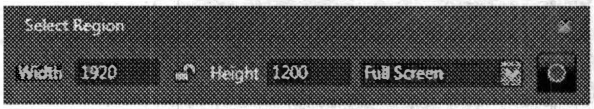


Рис. 16.9. Установка области записи

Выбрав нужный регион и еще раз нажав на кнопку Record, на экране отобразятся цифры, ведущие обратный отчет, после чего начнется запись.

Полученный файл нельзя тут же использовать и проигрывать через **MediaElement** или любой медиа плейер. Но формат полученного файла вполне

доступен **Expression Encoder**, который и следует использовать для преобразования файла к желаемому формату.

На этом мы и закончим рассматривать **Expression Encoder**.

Работаем с Expression Blend

Общий обзор

Этот раздел я решил разбить на три части. В первой части я планирую дать обзор интерфейса Expression Blend, а в остальных частях продемонстрировать возможности продукта, которые могут быть полезны именно разработчику. Так, если мне нужно получить стандартный шаблон для кнопки или текстового поля, то я, не задумываясь, открою Expression Blend, но если и задуматься, то я не знаю другой утилиты, которая позволила бы мне извлечь шаблон из сборки. Что касается полной функциональности Expression Blend, то по этому продукту написано множество книг, каждая не менее 500 страниц. Продукт действительно предоставляет множество возможностей, но большую часть из них используют в основном дизайнеры.

Итак, чтобы создать проект в Expression Blend, достаточно выбрать команду **File->New Project**, а затем выбрать тип проекта. Как я уже писал раньше, Expression Blend позволяет создавать интерфейсы как для WPF, так и для Silverlight-приложений. Если брать Expression Blend Preview for .NET 4, то тут присутствует три типа проектов: **Silverlight Application+Website**, **Silverlight Application**, **Silverlight Control Library** (рис. 16.10).

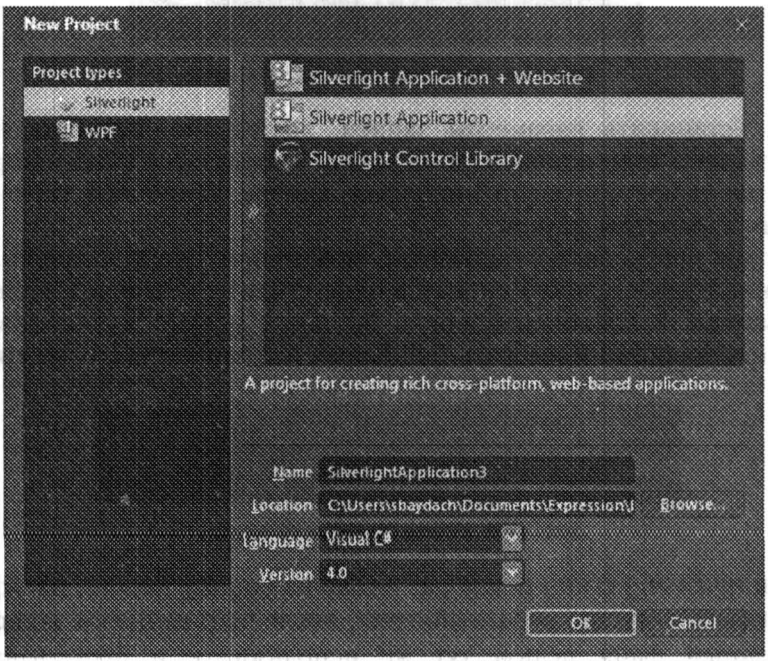


Рис. 16.10. Проекты в Expression Blend

Следующее окно, которое расположено на закладке после окна **Projects**, — это окно **Assets**. Тут можно получить доступ ко всем объектам, доступным в проекте, включая элементы управления, стили, видео файлы и

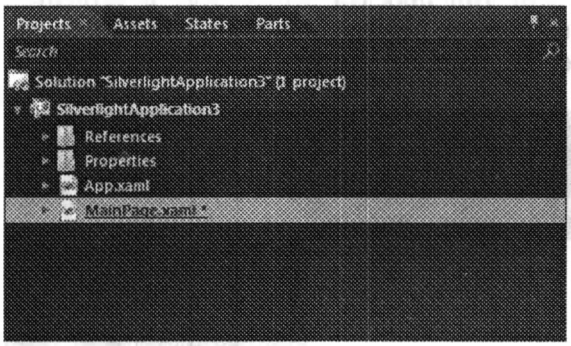


Рис. 16.13. Окно Projects

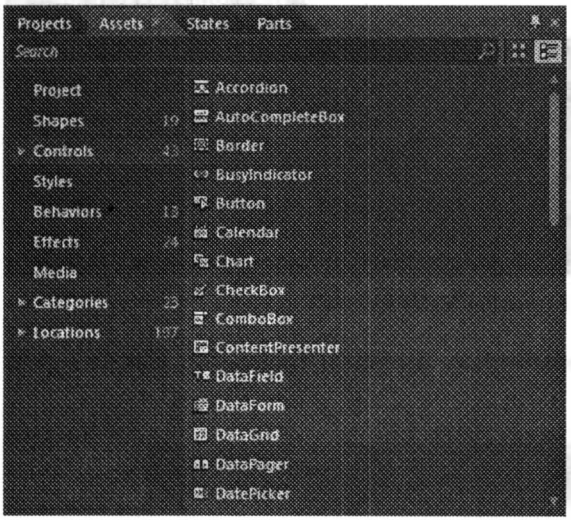


Рис. 16.14. Окно Assets

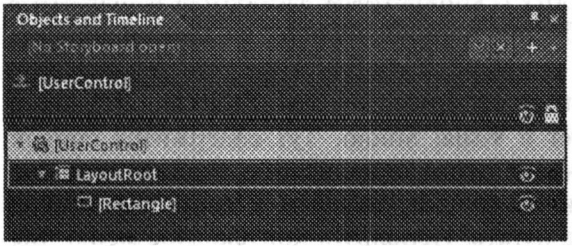


Рис. 16.15. Окно Objects and Timeline

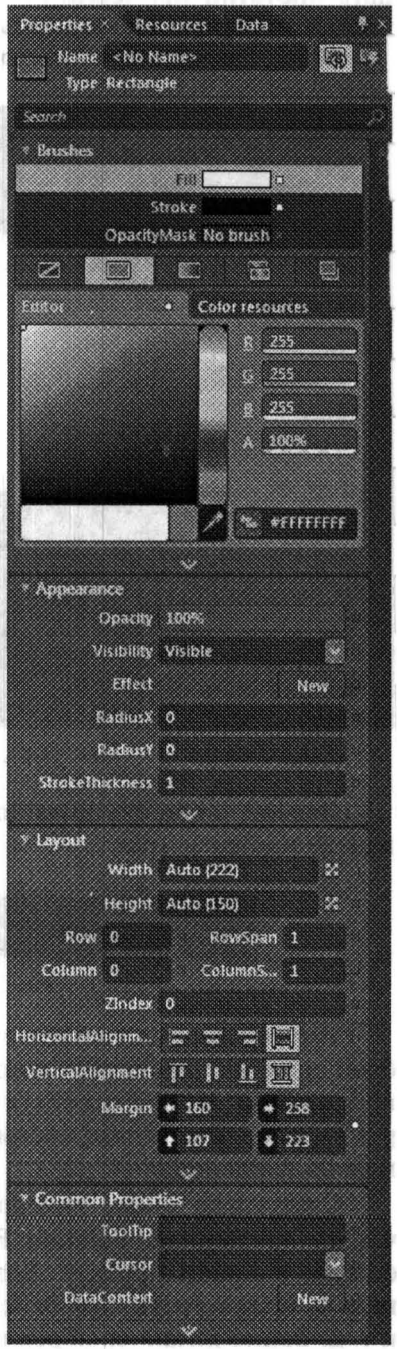


Рис. 16.16. Окно Properties

др. Понравившийся элемент управления можно перетянуть на панель инструментов для последующей с ним работы (рис. 16.14).

Следующее окно **Objects and Timeline** позволяет выполнять навигацию по дереву объектов Silverlight-приложения, а также работать с анимацией (рис. 16.15).

Выбирая элемент в окне **Objects and Timeline** или непосредственно в окне редактирования интерфейса, Вы можете редактировать свойства активного объекта, используя окно **Properties**. Тут Вы можете задать трансформацию, кисти и другие свойства элемента (рис. 16.16).

И, наконец, если Вы хотите перейти в режим редактирования XAML файла вне визуального дизайнера, то можете воспользоваться меню **View->Active Document View**.

Работа с анимацией

Expression Blend позволяет достаточно просто создавать анимацию любой сложности. Для этого нужно воспользоваться окном **Objects and Timeline**, нажав кнопку **New...** В результате этого действия на экране отобразится окно **Create Storyboard Resource** (рис. 16.17).

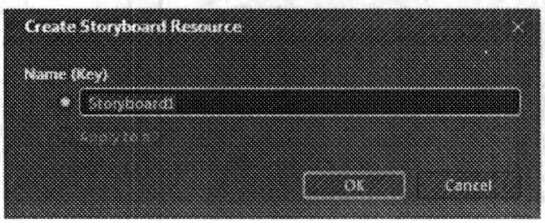


Рис. 16.17. Создание Storyboard

Expression Blend создает **Storyboard** в ресурсах приложения, что вполне прогнозируемо, учитывая слабые возможности триггеров.

Создав новый **Storyboard,** приложение переходит в режим «записи» анимации (рис. 16.18).

Остается передвинуть ползунок на шкале, отображающей Timeline, и установить новые свойства для выбранного объекта (объектов). После выполнения этого действия, можно выполнить предварительный просмотр анимации.

Работая с анимацией, следует обратить внимание на дерево объектов. В режиме записи анимации тут отображаются все свойства, которые подвергаются анимации. Выбрав одно из свойств, можно дополнительно настроить анимацию через окно свойств (рич. 16.19, 16.20).

Создание шаблонов для элементов управления

Используя **Expression Blend,** разработчик достаточно просто может получить доступ к шаблону элемента по умолчанию и отредактировать любые его части.

Чтобы создать новый шаблон на основе стандартного шаблона, достаточно выбрать желаемый элемент управления и инициировать вызов контекстно-

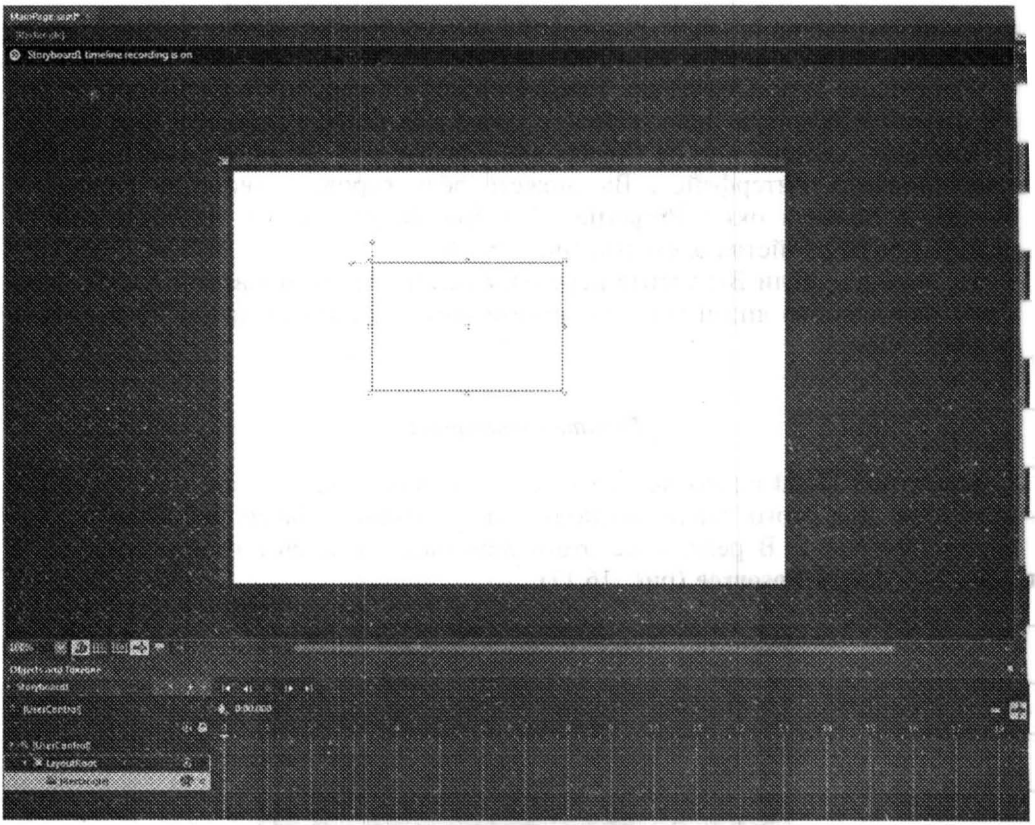


Рис. 16.18. Переход в режим «записи» анимации

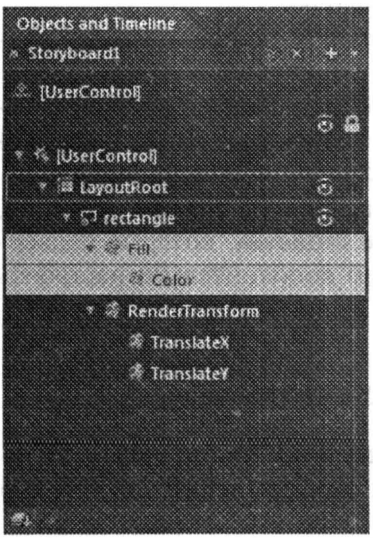


Рис. 16.19. Выбор свойств, подвергшихся номинации

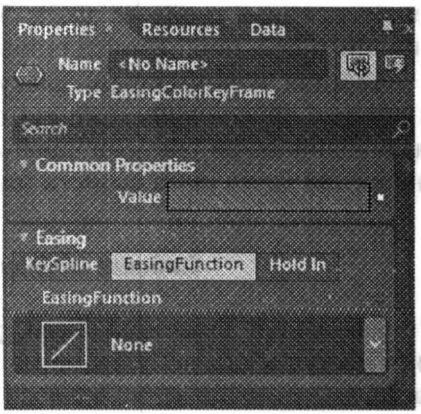


Рис. 16.20. Установка свойств анимации

го меню, где выбрать команду **Edit Template->Edit a Copy...** Эта команда позволяет создать новый ресурс, где определить стиль с копией стандартного шаблона. Окно, позволяющее создать шаблон, открывает возможности выбрать место размещения ресурса и имя ресурса (рис. 16.21).

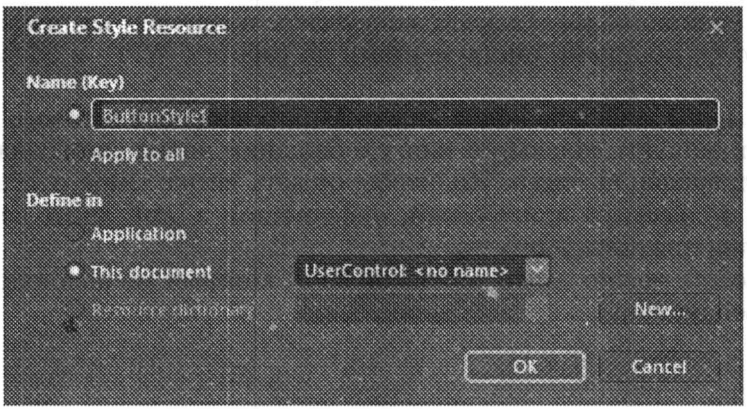


Рис. 16.21. Создание шаблона

После создания шаблона Вы можете перейти в Visual Studio для редактирования XAML, или продолжить редактирование в **Expression Blend**. Так, в окне **Objects and Timeline** Вы можете видеть все составляющие шаблона, а в окне **States** — доступные состояния (рис. 16.22, 16.23).

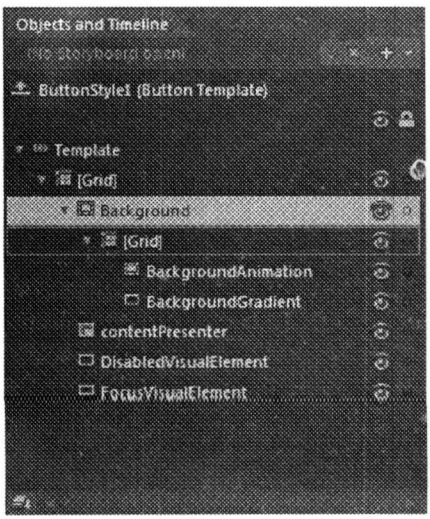


Рис. 16.22. Составляющие шаблона

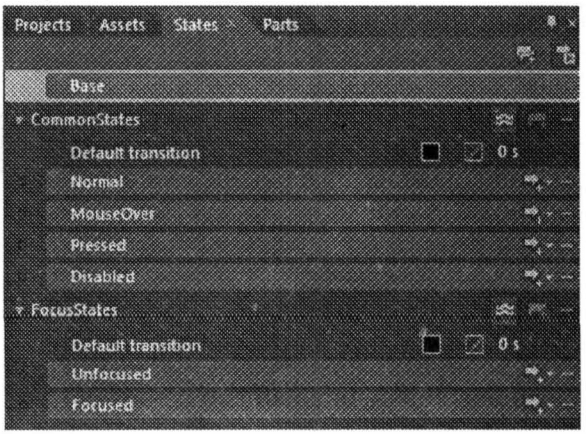


Рис. 16.23. Управление состояниями

Заключение

Microsoft — первая компания на рынке, которая построила своеобразный мост между дизайнерами и разработчиками. Именно в Expression Blend впервые был реализован подход совместной работы дизайнера и разработчика над одним проектом. Нужно отметить, что Expression Design и Expression Blend никогда не позиционировались как конкуренты продуктам компании Adobe, а лишь являются мощными инструментами по работе с WPF- и Silverlight-приложениями.

Вот мы и добрались до конца книги. Много вопросов осталось за кадром. Так, можно написать отдельную книгу о создании новых элементов управления или о трансляции видео. Но, чтобы создать впечатление о технологии и приступить к созданию приложений, информации в этой книге должно хватить. Остается пожелать лишь удачи.

Меня всегда можно найти через мой блог, который расположен по адресу http://baydachnyy.com.

Оглавление

От автора 3

Глава 1. ВВЕДЕНИЕ В SILVERLIGHT 4 5

Поддержка Drag&Drop 5

Печать из Silverlight-приложений 8

Обработка нажатия правой кнопки мыши 9

Работа с буфером обмена 11

Элементы управления WebBrowser и HtmlBrush 14

RichTextArea элемент управления 16

Управление окном приложения 18

Поддержка уведомлений 19

Поддержка микрофона и камеры 21

Поддержка колесика мыши 27

Элемент управления ViewBox 28

Повышение доверия 29

Расширенные возможности работы в полноэкранном режиме 30

Отсутствие сообщений о доступе к ресурсам 30

Запросы между доменами 31

Доступ к некоторым папкам 31

Взаимодействие с COM 31

Неявные стили 32

Заключение 32

Глава 2. НАЧИНАЕМ РАБОТУ С SILVERLIGHT 33

Что такое Silverlight? 33

Инструменты для создания Silverlight-приложений. 35

Первое приложение в Expression Blend 4 37

Создание приложения в Visual Studio 2010 42

Обзор технологии ... 45
XAML ... 45
Элементы компоновки ... 45
Элементы управления ... 46
Графические примитивы ... 46
Управление видео ... 46
Работа с данными ... 46
Работа со службами ... 47
Работа вне браузера ... 47
Базовые классы ... 47
Заключение ... 47

Глава 3. АРХИТЕКТУРА SILVERLIGHT ... 48
Структура приложения ... 48
Развертывание приложения ... 53
Кэширование сборок и загрузка по требованию ... 54
Загрузка сборки по требованию ... 54
Кэширование сборки ... 58
Размещение Silverlight-элемента на странице ... 60
Использование элемента <object> ... 60
Немного о классах в JavaScript ... 62
Использование Silverlight.js ... 64
Анимация во время загрузки ... 66
Взаимодействие со встраиваемым элементом ... 69
Использование JavaScript ... 69
Переход в полноэкранный режим ... 70
Взаимодействие Silverlight и JavaScript ... 72
Вызов управляемых методов из JavaScript ... 72
Вызов JavaScript методов из управляемого кода ... 74
Взаимодействие между Silverlight-приложениями ... 75
Заключение ... 78

Глава 4. ИСПОЛЬЗОВАНИЕ XAML ... 79
Введение в XAML ... 79
Основные конструкции ... 80
Пространства имен в XAML ... 84
Подключение кода и обработчиков событий ... 85
Расширение разметки ... 88

Зависимые свойства ... 89
Динамическая загрузка XAML ... 90
Заключение ... 91

Глава 5. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ И СОБЫТИЯ ... 92
Немного об элементах управления ... 92
Элементы компоновки ... 94
Элемент управления Canvas ... 95
Элемент управления StackPanel ... 96
Элемент управления Grid ... 99
Базовые элементы управления ... 105
Класс Control ... 105
Кнопки ... 106
Текстовые элементы управления ... 108
Элементы управления списками ... 109
Элементы управления, основанные на диапазоне значений ... 111
Элемент управления ToolTip ... 112
Использование диалоговых окон ... 112
Заключение ... 112

Глава 6. ПРИВЯЗКА К ДАННЫМ ... 113
Привязка к свойству элемента управления ... 113
Привязка к объекту ... 117
Привязка к коллекции ... 122
Конвертеры данных ... 125
Проверка данных при связывании ... 128
ValidatesOnExceptions и NotifyOnValidationError ... 128
ValidatesOnDataErrors ... 130
ValidatesOnNotifyDataErrors ... 133
Заключение ... 133

Глава 7. ВЗАИМОДЕЙСТВИЕ С СЕРВЕРОМ ... 135
Использование WebClient ... 135
Использование HttpWebRequest и HttpWebResponse ... 138
Использование прокси-классов для взаимодействия со службами ... 140
Доступ к службам в других доменах ... 143
Заключение ... 144

Глава 8. ГРАФИКА, ТРАНСФОРМАЦИЯ И АНИМАЦИЯ 145

Графические примитивы 145

Кисти 149

SolidColorBrush 149

Поддержка системных цветов 150

LinearGradientBrush 150

RadialGradientBrush 151

ImageBrush и VideoBrush 153

Использование геометрических объектов 154

Работа с изображениями 155

Работа с эффектами 155

Pixel API 159

Работа с кэшем 161

Трансформация 161

Основные виды трансформаций 161

CompositeTransform в Silverlight 4 164

Трехмерные проекции 165

Введение в анимацию 167

Общие типы анимации 167

Запуск анимации 168

Анимация с помощью ключевых кадров 169

Простая анимация 170

Заключение 171

Глава 9. РАБОТА С АУДИО И ВИДЕО 172

Использование MediaElement 172

Общие сведения 172

Использование маркеров 181

Поддержка GPU 183

Возможности Internet Information Services 7 184

Запуск Web Platform Installer 184

Создание списков 187

Возможности Bit Rate Throttling 187

Использование Smooth Streaming 188

Защита видео с помощью DRM 191

Заключение 191

Глава 10. РЕСУРСЫ И СТИЛИ 193
Ресурсы 193
Ресурсы приложения 193
Ресурсы объектов 194
Выделение ресурсов объектов в отдельные файлы 196
Стили 197
Понятие стилей 197
Динамическая установка стилей 200
BasedOn стили 201
Заключение 201

Глава 11. СОЗДАНИЕ ШАБЛОНОВ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ 202
Понятие шаблона 202
Разбор шаблона для элемента Button 203
Составляющие элемента управления 203
Состояния и переходы 205
Заключение 208

Глава 12. ОТЛАДКА ПРИЛОЖЕНИЙ И ТЕСТИРОВАНИЕ 209
Отладка с помощью Visual Studio 2010 209
Обработка ошибок в Silverlight 211
Обработка ошибок в управляемом коде 211
Обработка ошибок в JavaScript 212
Асинхронный вызов методов 213
Тестирование Silverlight-приложений 214
Заключение 218

Глава 13. СОЗДАНИЕ СЛОЖНЫХ ПРИЛОЖЕНИЙ 219
Разработка приложений, работающих вне браузера 219
Isolated Storage 223
IsolatedStorageSettings 223
IsolatedStorageFile 224
Навигация в Silverlight-приложениях 226
Расширение модели приложения 234
Managed Extensibility Framework 235
Заключение 236

Глава 14. ИСПОЛЬЗОВАНИЕ DEEP ZOOM 237
Что такое Deep Zoom? 237
Использование Deep Zoom Composer 238
Работа с Deep Zoom в Silverlight 242
Заключение 243

Глава 15. ИНТЕГРАЦИЯ С SHAREPOINT 2010 244
Обзор возможностей 244
Работа с Web-частями 246
Развертывание Silverlight-приложения с помощью Visual Studio 2010 247
Использование REST 252
Поддержка Client API 256
Заключение 258

Глава 16. ВВЕДЕНИЕ В MICROSOFT EXPRESSION STUDIO 259
Обзор продуктов 259
Работа с Expression Encoder 260
Преобразование видео 260
Использование встроенных шаблонов 262
Использование Expression Encoder для трансляции живого видео . 263
Захват изображения и звука 264
Работаем с Expression Blend 266
Общий обзор 266
Работа с анимацией 269
Создание шаблонов для элементов управления 269
Заключение 272

Серия «Библиотека профессионала»

С. С. Байдачный

SILVERLIGHT 4: СОЗДАНИЕ НАСЫЩЕННЫХ WEB-ПРИЛОЖЕНИЙ

Москва
СОЛОН-ПРЕСС
2010