ГЛАВА 2

Технология XAML

XAML (Extensible Application Markup Language — расширяемый язык разметки приложений) предназначен для определения объектов .NET. Технология XAML применяется во многих областях, однако изначально она разрабатывалась как часть технологии WPF (Windows Presentation Foundations), которая позволяет разработчикам приложений Windows создавать мощные пользовательские интерфейсы. Для создания интерфейсов приложений Silverlight используется тот же стандарт XAML, что и в WPF.

Концептуально язык XAML играет ту же роль, что и HTML или XHTML, — он предназначен для определения элементов, размещаемых в области содержимого. Для манипуляции элементами XHTML используется клиентский код JavaScript, а элементами XAML — клиентский код C#. Документы как XHTML, так и XAML основаны на синтаксисе XML и состоят из элементов, вложенных друг в друга в любой последовательности.

В этой главе представлено подробное введение в XAML. Изучив общие правила XAML, вы легко поймете, что можно или нельзя сделать в пользовательском интерфейсе Silverlight и как можно изменить интерфейс путем редактирования разметки. Изучая дескрипторы в документах XAML, вы многое узнаете об объектной модели интерфейсов Silverlight и будете готовы к более глубокому изучению технологии Silverlight.

Примечание

Технология XAML была создана для WPF, тем не менее она, кроме Silverlight, используется в нескольких других высокоуровневых технологиях, например, для определения рабочих потоков WF (Workflow Foundation) и создания документов XPS (XML Paper Specification — спецификация документов XML).

Основы XAML

Синтаксис XAML базируется на следующих общих правилах.

- Каждый элемент документа XAML отображается на определенный экземпляр класса Silverlight. Имя элемента всегда точно совпадает с именем класса. Например, элемент `<Button>` вынуждает надстройку Silverlight создать объект `Button` (Кнопка).
- Как и в любом документе XML, элементы XAML можно вкладывать друг в друга. Каждый класс XAML достаточно гибкий для обеспечения нужного поведения в каждой ситуации. Вложение элементов разметки обычно отображает вложенность элементов интерфейса. Например, если элемент `<Button>` расположен в элементе `<Grid>`, то и в пользовательском интерфейсе кнопка `Button` включена в элемент `Grid` (Решетка).

- Свойства класса определяются с помощью атрибутов. Однако в некоторых ситуациях атрибутов для этого недостаточно. Тогда для определения дополнительных свойств класса используются специальные вложенные дескрипторы.

Совет

Если вы не знакомы с XML, то прежде, чем начать изучение XAML, я рекомендую почитать бесплатный сетевой учебник `www.w3schools.com/xml`.

Рассмотрим пустой документ XAML, сгенерированный программой Visual Studio и представляющий чистую страницу. Номера строк добавлены для облегчения ссылки на строки, в реальном документе их не должно быть.

```
1 <UserControl x:Class="SilverlightApplication1.Page"
2   xmlns="http://schemas.microsoft.com/client/2007"
3   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4   Width="400" Height="300">
5
6   <Grid x:Name="LayoutRoot" Background="White">
7   </Grid>
8 </UserControl>
```

Документ содержит два элемента: элемент верхнего уровня вложенности `UserControl` (Пользовательский элемент управления), охватывающий все содержимое Silverlight, и элемент `Grid`, в котором можно размещать элементы интерфейса.

Как и в документах XML, в документе XAML может присутствовать только один элемент верхнего уровня вложенности. В данном примере это — элемент `UserControl`. Следовательно, документ должен завершаться закрывающим дескриптором `</UserControl>`, после которого не должно быть никакой разметки.

Открывающий дескриптор элемента `UserControl` содержит несколько важных атрибутов, включая имя класса и два пространства имен XAML, которые рассматриваются в следующем разделе. Кроме того, в строке 4 определены ширина и высота области содержимого Silverlight:

```
4  Width="400" Height="300"
```

Каждый атрибут определяет одно свойство класса `UserControl`. В данном примере атрибуты `Width` (Ширина) и `Height` (Высота) сообщают надстройке Silverlight о необходимости создать область размером 400×300 пикселей. Аналогично определяется свойство `Background` (Фон) элемента `Grid`: оно сообщает о том, что содержимое должно выводиться на белом фоне.

Пространства имен XAML

При наличии в файле XAML элемента `<UserControl>` синтаксический анализатор Silverlight воспринимает его как задание создать экземпляр класса `UserControl`. Однако он не знает, что это за класс. Ведь несмотря на то, что пространство имен Silverlight содержит только один класс `UserControl`, нет никакой гарантии того, что разработчик не создал пользовательский класс под таким же именем. Следовательно, чтобы был создан нужный класс, необходим способ задания пространства имен.

В Silverlight классы разрешаются путем добавления пространств имен XML в пространства имен Silverlight. В приведенной выше разметке определены два пространства имен.

```
2 xmlns="http://schemas.microsoft.com/client/2007"
3 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Примечание

Пространства имен XML объявляются с помощью атрибутов, размещенных в открывающих дескрипторах. Существует соглашение, согласно которому все используемые в документе пространства имен должны быть объявлены в первом же дескрипторе, как в данном примере. Это существенно облегчает визуальный анализ документа, поскольку избавляет от необходимости искать объявления во всей разметке. После объявления пространства имен его можно использовать в любом месте документа.

Специальный атрибут `xmlns` зарезервирован в XML для объявления пространств имен. В приведенной выше разметке объявлены два пространства имен, используемые в каждом документе XAML приложения Silverlight.

- ***http://schemas.microsoft.com/client/2007*** — базовое пространство имен Silverlight 2.0. Оно содержит все классы Silverlight 2.0, включая `UserControl` и `Grid`. Обычно оно объявляется без префикса, поэтому оно служит пространством имен, заданным по умолчанию для всего документа. Иными словами, если не указано обратное, каждый элемент автоматически располагается в этом пространстве имен.
- ***http://schemas.microsoft.com/winfx/2006/xaml*** — пространство имен XAML. Содержит различные средства XAML, позволяющие манипулировать способами интерпретации документа. Данное пространство имен обозначается префиксом `x`. Это означает, что его можно применить, расположив префикс перед именем элемента (`<x:`*`имя_элемента`*`>`).

Информация о пространстве имен позволяет синтаксическому анализатору XAML найти нужный класс. Например, когда синтаксический анализатор видит элементы `UserControl` и `Grid`, он знает, что они принадлежат установленному по умолчанию пространству имен `http://schemas.vicrosoft.com/client/2007`. Благодаря этому анализатор находит соответствующие классы `System.Windows.UserControl` и `SystemWindows.Controls.Grid`.

Пространства имен XAML и Silverlight

Пространство имен XAML не добавлено в одно пространство имен Silverlight. Вместо этого все пространства имен Silverlight находятся в одном и том же пространстве имен XML. Создатели спецификации XAML предпочли такую архитектуру по нескольким причинам. По общепринятым соглашениям пространства имен XML часто имеют форму URI (как в данном случае). Имена пространств имен выглядят так, будто они указывают на некоторый ресурс в Интернете, но в действительности они ни на что не указывают. Формат URI применяется, чтобы гарантировать уникальность имен, присвоенных разным пространствам имен. Разные организации не имеют одинаковых URI, поэтому они могут присваивать свои URI собственным пространствам имен, не опасаясь, что кто-либо применит такое же имя. Домен `schemas.microsoft.com` принадлежит компании Microsoft, поэтому только Microsoft может использовать этот URI в именах пространств имен XML.

Еще одна причина отсутствия взаимно однозначного соответствия между пространствами имен XAML и Silverlight состоит в том, что это существенно усложнило бы документы XAML. Если бы каждое пространство имен Silverlight было представлено как отдельное пространство имен XML, вам пришлось бы выбирать правильное пространство имен для каждого элемента управления. Документы XAML и Silverlight быстро превратились бы в беспорядочное месиво. Поэтому создатели Silverlight предпочли отобразить все пространства имен Silverlight, содержащие элементы пользовательских интерфейсов, на единственное пространство имен XML. Это допустимо, поскольку Microsoft гарантирует, что в разных пространствах имен Silverlight никакие два класса не имеют одно и то же имя.

Во многих ситуациях необходим доступ к собственным пространствам имен в файле XAML. Чаще всего он необходим, когда нужно применить пользовательский

элемент управления Silverlight (*пользовательским* называется элемент управления, созданный вами или другим разработчиком, а не компанией Microsoft). В этом случае нужно определить новый префикс пространства имен XML и добавить его в пользовательскую сборку. Это можно сделать следующим образом.

```
<UserControl x:Class="SilverlightApplication1.Page"
  xmlns:w="clr-namespace:Widgets;assembly=WidgetLibrary"
  ...
```

Объявление пространства имен определяет три важных параметра.

- **Префикс пространства имен XML.** Префикс применяется для ссылки на пространство имен в документе XAML. В данном примере задан префикс `w`, однако можно задать любой идентификатор, не конфликтующий с другими префиксами пространств имен.
- **Название пространства имен .NET.** В данном примере все классы расположены в пространстве имен `Widgets`. Если есть классы, которые нужно использовать в разных пространствах имен, их можно отобразить на разные или одно и то же пространство имен XML при условии, что они не конфликтуют с другими именами классов.
- **Имя сборки.** В данном примере классы находятся в сборке `WidhetLibrary.dll`. Расширение `.dll` в имя сборки не добавляется. Драйвер Silverlight ищет данную сборку в том же пакете XAP, в котором определена сборка проекта.

Примечание

Не забывайте, что в Silverlight используется облегченная версия CLR. Поэтому в приложении Silverlight нельзя применять сборки полнофункциональных библиотек классов. Вместо них нужно применять библиотеки классов Silverlight. В рабочей среде Visual Studio легко создать пользовательскую библиотеку классов Silverlight, выбрав шаблон проекта Silverlight Class Library (Библиотека классов Silverlight).

Отобразив пространство имен .NET на пространство имен XML, его можно использовать в любом месте документа XAML. Например, если пространство имен `Widgets` содержит элемент управления `HotButton`, создать его экземпляр можно следующим образом.

```
<w:HotButtonj Text="Щелкните здесь!" Click="DoSomething">
</w:HotButton>
```

Фоновый класс

Технология XAML позволяет определить пользовательский интерфейс, однако этого недостаточно. Чтобы приложение не просто выводилось на экран, а что-нибудь делало, к интерфейсу нужно подключить обработчики событий, содержащие код приложения. В документе XAML можно легко подключить обработчик с помощью атрибута `Class`.

```
1 <UserControl x:Class="SilverlightApplication1.Page"
```

Префикс `x` размещает атрибут `Class` в пространстве имен XAML. Это означает, что атрибут `Class` является частью языка XAML, а не инфраструктуры Silverlight.

Атрибут `Class` приказывает синтаксическому анализатору Silverlight сгенерировать новый класс, имеющий заданное имя. В данном примере создается новый класс `SilverlightApplication1.Page`, производный от класса `UserControl`. Автоматически сгенерированная часть класса объединяется с кодом, добавленным разработчиком в фоновый файл.

Обычно каждому файлу XAML соответствует фоновый класс, содержащий клиентский код C#. Для файла `Page.xaml` рабочая среда Visual Studio создает фоновый класс `Page.xaml.cs`. Ниже приведено содержимое файла `Page.xaml.cs`.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
namespace SilverlightApplication1
{
  public partial class Page : UserControl
  {
    public Page()
    {
      InitializeComponent();
    }
  }
}
```

В данный момент код класса `Page` не содержит никакой функциональности. Однако он все же содержит нечто очень важное, а именно — конструктор, вызывающий метод `InitializeComponent()` при создании экземпляра класса. Метод `InitializeComponent()` проходит по разметке и создает соответствующие объекты, устанавливает их свойства и подключает определенные вами обработчики событий.

Примечание

Метод `InitializeComponent()` играет ключевую роль в содержимом Silverlight. Поэтому никогда не удаляйте его вызов из конструктора. Кроме того, добавив на страницу другой конструктор, убедитесь в том, что он тоже вызывает метод `InitializeComponent()`.

Именование элементов

В фоновом классе часто возникает необходимость манипулировать элементами программно, например, для чтения или изменения свойств, для подключения или отключения обработчиков событий во время выполнения и т.д. Чтобы это можно было делать, элемент управления должен иметь атрибут `Name` (Имя) пространства имен XAML. В предыдущем примере элемент управления `Grid` уже содержит атрибут `Name`, поэтому им можно манипулировать в коде фонового файла.

```
6 <Grid x:Name="LayoutRoot" Background="White">
7 </Grid>
```

Атрибут `Name` приказывает синтаксическому анализатору XAML добавить поле в автоматически сгенерированную часть класса:

```
private System.Windows.Controls.Grid layoutRoot;
```

Теперь к решетке можно обращаться в коде класса страницы с помощью имени `LayoutRoot`.

Свойство `Name` является частью языка XAML и применяется для облегчения интеграции фонового класса. Однако во многих классах (например, в базовом классе `FrameworkElement`, от которого производятся все элементы Silverlight) определены

собственные свойства `Name`, что существенно запутывает ситуацию. В этом случае заданное вами имя используется в коде автоматически сгенерированного файла и **не** используется для установки свойства `Name`.

Совет

В традиционных приложениях Windows Forms каждый элемент управления имеет имя. В приложении Silverlight это не обязательно. Если в коде не нужно обращаться к элементу, можете удалить атрибут `Name` из разметки. В примерах книги имена элементов обычно опущены (когда они не нужны). Благодаря этому разметка становится более краткой.

Свойства и события в XAML

До сих пор рассматривался довольно скучный пример — пустая страница, хостирующая пустой элемент управления `Grid`. Прежде чем двигаться дальше, рассмотрим более реалистичную страницу, содержащую несколько элементов. На рис. 2.1 показан пример приложения `EightBall`, которое автоматически отвечает на вопросы. На то, что ответы не очень осмысленные, пока не обращайте внимания. В ответ на любой вопрос программа случайным образом выбирает одну из заранее подготовленных фраз.

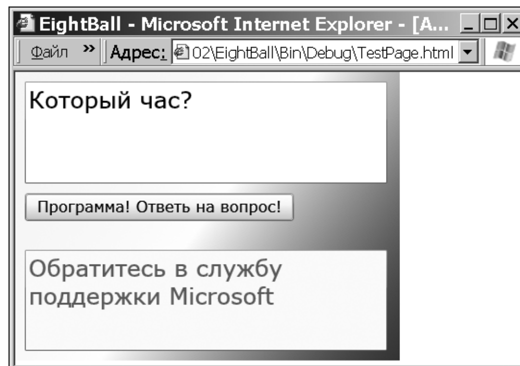

Рис. 2.1. Щелкните на кнопке, чтобы получить ответ

Тестовая страница содержит четыре элемента: `Grid` (этот элемент обычно используется для макетирования почти каждой страницы Silverlight), два объекта `TextBox` (Текстовое поле) и `Button` (Кнопка). Разметка, необходимая для компоновки и конфигурирования этих элементов, намного длиннее, чем в предыдущих примерах. Ниже приведена сокращенная разметка, отображающая общую структуру (опущенные подробности обозначены многоточиями).

```
<UserControl x:Class="EightBall.Page"
  xmlns="http://schemas.microsoft.com/client/2007"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Width="400" Height="300">

    <Grid x:Name="grid1">
      <Grid.Background>
        ...
      </Grid.Background>
      <Grid.RowDefinitions>
```

```
        ...
      </Grid.RowDefinitions>
      <TextBox x:Name="txtQuestion" ... >
        ...
      </TextBox>
      <Button x:Name="cmdAnswer" ... >
        ...
      </Button>
      <TextBox x:Name="txtAnswer" ... >
        ...
      </TextBox>
    </Grid>
</UserControl>
```

В следующих разделах рассматриваются части этого документа и синтаксис XAML.

Преобразование свойств и типов

Атрибуты элемента задают свойства соответствующего объекта. Например, атрибуты элемента TextBox конфигурируют выравнивание, внешние рамки и шрифт.

```
<TextBox x:Name="txtQuestion"
VerticalAlignment="Stretch" HorizontalAlignment="Stretch"
FontFamily="Verdana" FontSize="24" Foreground="Green" ... >
```

Чтобы разметка была работоспособной, класс System.Windows.Controls.TextBox должен предоставить следующие свойства: VerticalAlignment (Выравнивание по вертикали), HorizontalAlignment (Выравнивание по горизонтали), FontFamily (Семейство шрифтов), FontSize (Размер шрифта) и Foreground (Цвет переднего плана). Назначение каждого из этих свойств рассматривается в следующих главах.

Совет

Есть три специальных символа, которые нельзя ввести непосредственно в строку значения атрибута: кавычка, открывающая и закрывающая скобки. Чтобы применить эти символы в строке атрибута, нужно заменить их эквивалентными представлениями XML. Кавычку нужно заменить представлением ", открывающую скобку — <, а закрывающую скобку — >. Учитывайте, что указанные представления нужно использовать в разметке XAML, но не в коде C#.

Значение атрибута XML — это простая текстовая строка. Однако свойство объекта может быть любого типа, допустимого в .NET. В предыдущем примере есть несколько свойств: два свойства, в которых используются перечисления (VerticalAlignment и HorizontalAlignment), одно строковое значение (FontFamily), одно целочисленное (FontSize) и один объект типа Brush (объект Foreground).

Чтобы отобразить строковые значения на нестроковые свойства, синтаксический анализатор XAML выполняет преобразование типов, как и в полнофункциональной инфраструктуре .NET Framework. Синтаксический анализатор находит нужный метод преобразования в два этапа.

1. Поиск объявления свойства, а в нем — атрибута TypeConverter (Преобразователь типов). Если этот атрибут присутствует, в нем указан класс, который может выполнить преобразование. Например, если используется свойство Foreground, синтаксический анализатор в первую очередь ищет его объявление.

2. Если в объявлении свойства нет атрибута `TypeConverter`, синтаксический анализатор проверяет объявление класса соответствующего типа данных. Например, в свойстве `Foreground` используется объект `Brush` (Кисть). В классе `Brush` и его производных классах используется преобразователь `BrushConverter`, поскольку класс `Brush` содержит значение `typeof(BrushConverter)` атрибута `TypeConverter`.

Если в объявлении свойства или класса нет преобразователя типов, ассоциированного со свойством, синтаксический анализатор XAML генерирует сообщение об ошибке.

Как видите, система простая, но гибкая. Когда преобразователь типов установлен на уровне класса, он применяется ко всем свойствам, в которых используется данный класс. С другой стороны, если для некоторого свойства нужно тонко настроить способ преобразования, можно вместо преобразователя на уровне класса применить атрибут `TypeConverter` на уровне свойства.

Преобразователи типов можно применять также в коде, однако их синтаксис довольно запутанный. Почти всегда лучше установить свойство непосредственно в разметке. Это не только быстрее, но и позволяет избежать потенциальных ошибок, связанных с опечатками при вводе строк. Такие ошибки обнаруживаются только во время выполнения. Данная проблема не затрагивает документы XAML, поскольку синтаксический анализатор, обрабатывая разметку, проверяет ее на валидность во время компиляции.

Примечание

Язык XAML, как и любой другой язык на основе XML, чувствителен к регистру букв. Это означает, что написать `<button>` вместо `<Button>` нельзя. Однако преобразователи типов обычно не чувствительны к регистру букв. Например, записи `Foreground="White"` и `Foreground="white"` эквивалентны.

Составные свойства

Преобразователи типов довольно удобны, однако не во всех ситуациях. Например, некоторые свойства являются полнофункциональными объектами, обладающими собственными наборами свойств. В принципе для них можно создать строковое представление, распознаваемое преобразователем типов, однако их синтаксис весьма сложный, и, кроме того, такой подход чреват коварными ошибками.

К счастью, спецификация XAML предоставляет возможность задавать составные свойства. В этом синтаксисе дочерний элемент имеет имя в форме *имя_родительского_элемента.имя_свойства*. Например, объект `Grid` имеет свойство `Background`, позволяющее задать кисть для прорисовки фона элемента. Если нужно применить более сложную кисть, а не просто залить элемент одним цветом, нужно добавить дочерний дескриптор `Grid.Background`, как в приведенной ниже разметке.

```
<Grid x:Name="grid1">
  <Grid.Background>
    ...
  </Grid.Background>
    ...
</Grid>
```

Ключевая деталь, благодаря которой вся конструкция работает, — точка в имени элемента. Она позволяет отличить составное свойство от вложенного содержимого других типов.

Как присвоить значение составному свойству? Трюк состоит в добавлении во вложенный элемент другого дескриптора для создания экземпляра класса. В приложении EightBall (см. рис. 2.1) фон заполняется градиентом. Для определения градиента создан объект LinearGradientBrush:

```
<Grid x:Name="grid1">
  <Grid.Background>
    <LinearGradientBrush>
    </LinearGradientBrush>
  </Grid.Background>
  ...
</Grid>
```

Имя LinearGradientBrush входит в пространство имен Silverlight, поэтому в дескрипторах можно использовать пространство имен XML, установленное по умолчанию.

Однако создать объект LinearGradientBrush недостаточно. Кроме этого, нужно задать цвета градиента. Это делается путем заполнения свойства LinearGradientBrush.GradientStops коллекцией объектов GradientStop. Свойство GradientStops слишком сложное, чтобы его можно было установить только с помощью атрибута. Вместо этого необходимо применить составное свойство.

```
<Grid x:Name="grid1">
  <Grid.Background>
    <LinearGradientBrush>
      <LinearGradientBrush.GradientStops>
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Grid.Background>
    ...
</Grid>
```

Коллекцию GradientStops можно заполнить набором объектов GradientStop. Каждый объект GradientStop обладает свойствами Offset и Color. Два этих значения можно установить с помощью обычных атрибутов.

```
<Grid x:Name="grid1">
  <Grid.Background>
    <LinearGradientBrush>
      <LinearGradientBrush.GradientStops>
        <GradientStop Offset="0.00" Color="Yellow" />
        <GradientStop Offset="0.50" Color="White" />
        <GradientStop Offset="1.00" Color="Purple" />
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Grid.Background>
  ...
</Grid>
```

Примечание

Описанный выше синтаксис составных свойств можно применить для любого свойства, однако в большинстве случаев используется более простой, обычный, синтаксис на основе значений атрибутов (если для свойства есть подходящий преобразователь типов). Использование синтаксиса на основе атрибутов делает код более компактным.

Любой набор дескрипторов XAML можно заменить набором операторов кода C#, выполняющих те же операции. Например, приведенные выше дескрипторы, заполняющие фон заданным градиентом, эквивалентны следующему коду.

```
LinearGradientBrush brush = new LinearGradientBrush();
GradientStop gradientStop1 = new GradientStop();
gradientStop1.Offset = 0;
gradientStop1.Color = Colors.Yellow;
brush.GradientStops.Add(gradientStop1);
GradientStop gradientStop2 = new GradientStop();
gradientStop2.Offset = 0.5;
gradientStop2.Color = Colors.White;
brush.GradientStops.Add(gradientStop2);
GradientStop gradientStop3 = new GradientStop();
gradientStop3.Offset = 1;
gradientStop3.Color = Colors.Purple;
brush.GradientStops.Add(gradientStop3);
grid1.Background = brush;
```

Подключенные свойства

Кроме обычных свойств, спецификация XAML поддерживает концепцию *подключенных свойств*, которые могут применяться к нескольким элементам и определяться в разных классах. В Silverlight подключенные свойства часто используются для конфигурирования элементов управления.

Каждый элемент управления имеет собственный набор внутренних свойств. Например, текстовое поле имеет свойства `FontFamily` (Семейство шрифтов), `Foreground` (Цвет текста) и `Text` (Текстовое содержимое элемента управления). При размещении элемента управления в контейнере желательно присвоить ему дополнительные средства в зависимости от типа контейнера. Например, при размещении текстового поля в решетке возникает необходимость задать ячейку решетки, в которой он позиционируется. Дополнительные средства добавляются в элемент управления с помощью подключенных свойств.

Подключенное свойство всегда имеет имя, состоящее из двух частей в форме *`определяющее_свойство.имя_свойства`*. Это позволяет синтаксическому анализатору XAML отличать подключенные свойства от обычных.

В приложении `EightBall` подключенные свойства позволяют отдельным элементам размещать себя в разных строках невидимой решетки.

```
<TextBox ... Grid.Row="0">
 [Здесь введите вопрос]
</TextBox>

<Button ... Grid.Row="1">
 Программа! Ответь на вопрос!
</Button>

<TextBox ... Grid.Row="2">
[Здесь появится ответ]
</TextBox>
```

Подключенные свойства фактически не являются свойствами. В действительности они транслируются в вызовы методов. Синтаксический анализатор XAML вызывает статический метод, имя которого определено в форме *`определяющий_тип.`* `Set`*`имя_свойства`*`()`. В предыдущем примере определяющим типом является класс `Grid`, а именем свойства — идентификатор `Row` (Строка), поэтому синтаксический анализатор вызывает метод `Grid.SetRow()`.

При вызове метода `Set`*`имя_свойства`*`()` синтаксический анализатор передает ему два параметра: изменяемый объект и задаваемое значение свойства. Напри-

мер, при установке свойства Grid.Row элемента управления TextBox синтаксический анализатор XAML неявно выполняет следующий код.

```
Grid.SetRow(txtQuestion, 0);
```

Применяемый шаблон вызова статического метода определяющего типа скрывает выполняемые операции. На первый взгляд кажется, что номер строки хранится в объекте Grid, однако на самом деле он хранится в объекте, к которому он применяется, в данном случае — в объекте TextBox.

Этот прием работоспособен благодаря тому, что класс TextBox производится от базового класса DependencyObject, как и любой другой элемент Silverlight. В классе DependencyObject может храниться неограниченная коллекция *зависимых свойств* (dependency properties). Подключенные свойства — это один из типов зависимых свойств. Зависимые свойства рассматриваются в главе 4.

Имя метода Grid.SetRow() фактически является псевдонимом, вызывающим метод *управляющий_объект*.SetValue(). Ниже приведен пример фактического вызова в приложении EightBall.

```
txtQuestion.SetValue(Grid.RowProperty, 0);
```

Подключенные свойства — ключевой компонент технологии Silverlight. Они служат для облегчения расширяемости системы. Например, определив свойство Row как подключенное, вы гарантируете возможность его использования с любым элементом управления. Другой вариант — определение свойства как части базового класса (например, класса FrameworkElement) — существенно усложняет приложение. Открытый (public) интерфейс будет загроможден свойствами, имеющими смысл только в некоторых ситуациях (в нашем примере — при использовании элемента в объекте Grid). Кроме того, будет невозможно добавить новый тип контейнера, требующий дополнительных свойств.

Вложение элементов

Документ XAML представляет собой древовидную структуру вложенных элементов. В рассматриваемом примере элемент окна содержит элемент Grid, который, в свою очередь, содержит элементы TextBox и Button.

Спецификация XAML позволяет каждому элементу самостоятельно решить, как он будет обрабатывать вложенные элементы. Взаимодействие с вложенными элементами выполняется одним из трех способов.

- Если родительский элемент реализует интерфейс IList<T>, синтаксический анализатор вызывает метод IList<T>.Add() и переходит в дочерний элемент.
- Если родительский элемент реализует интерфейс IDictionary<T>, синтаксический анализатор вызывает метод IDictionary<T>.Add() и переходит в дочерний элемент. При использовании коллекции типа словаря нужно также установить атрибут x:Key, чтобы присвоить имя ключа каждому элементу.
- Если родительский элемент содержит атрибут ContentProperty, синтаксический анализатор использует дочерний элемент для установки свойства.

Например (см. приложение EightBall на рис. 2.1), элемент LinearGradientBrush может содержать коллекцию объектов GradientStop, заданных следующим образом.

```
<LinearGradientBrush>
  <LinearGradientBrush.GradientStops>
    <GradientStop Offset="0.00" Color="Red" />
    <GradientStop Offset="0.50" Color="Indigo" />
    <GradientStop Offset="1.00" Color="Violet" />
```

```
  </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
```

Синтаксический анализатор XAML распознает элемент LinearGradient-Brush.GradientStops как составное свойство, поскольку его имя содержит точку. Однако вложенные дескрипторы (три элемента GradientStop) должны быть обработаны несколько иначе. В данном случае синтаксический анализатор знает, что свойство GradientStops возвращает объект GradientStopCollection, который реализует интерфейс IList. Поэтому синтаксический анализатор считает (вполне правильно), что каждый объект GradientStop должен быть добавлен в коллекцию с помощью метода IList.Add().

```
GradientStop gradientStop1 = new GradientStop();
gradientStop1.Offset = 0;
gradientStop1.Color = Colors.Red;
IList list = brush.GradientStops;
list.Add(gradientStop1);
```

Некоторые свойства могут поддерживать несколько типов коллекций. В этом случае нужно добавить дескриптор, задающий класс коллекции.

```
<LinearGradientBrush>
  <LinearGradientBrush.GradientStops>
    <GradientStopCollection>
      <GradientStop Offset="0.00" Color="Red" />
      <GradientStop Offset="0.50" Color="Indigo" />
      <GradientStop Offset="1.00" Color="Violet" />
    </GradientStopCollection>
  </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
```

Примечание

Если переменная объекта коллекции по умолчанию указывает на null, нужно добавить дескриптор, задающий класс коллекции, создав таким образом объект коллекции. Если же экземпляр коллекции уже существует и нужно просто заполнить его, дескриптор, задающий класс коллекции, можно опустить.

Вложенное содержимое не всегда указывает на коллекцию. Например, рассмотрим элемент Grid, содержащий несколько других элементов.

```
<Grid x:Name="grid1">
  ...
  <TextBox x:Name="txtQuestion" ... >
    ...
  </TextBox>
  <Button x:Name="cmdAnswer" ... >
    ...
  </Button>
  <TextBox x:Name="txtAnswer" ... >
    ...
  </TextBox>
</Grid>
```

Эти вложенные дескрипторы не являются составными свойствами, поскольку в их именах нет точек. Более того, элемент управления Grid не является коллекцией и не реализует интерфейс IList или IDictionary. Вместо этого элемент Grid поддерживает атрибут ContentProperty, указывающий на свойство, которое должно быть присвоено любому вложенному содержимому. Атрибут ContentProperty принадлежит классу Panel, от которого производится класс Grid.

```
[ContentPropertyAttribute("Children")]
public abstract class Panel : FrameworkElement
```

Приведенная выше запись сообщает о необходимости присвоения любому вложенному элементу свойства Children (Дочерний). Синтаксический анализатор XAML интерпретирует свойство содержимого по-разному, в зависимости от того, является ли оно коллекцией (если да, то оно должно реализовать интерфейс IList или IDictionary). Свойство Panel.Children возвращает объект UIElementCollection, реализующий интерфейс IList, поэтому синтаксический анализатор применяет метод IList.Add() для добавления вложенного содержимого в решетку.

Обрабатывая приведенную выше разметку, синтаксический анализатор XAML создает экземпляр каждого вложенного элемента и передает его объекту Grid с помощью метода Grid.Children.Add().

```
txtQuestion = new TextBox();
...
grid1.Children.Add(txtQuestion);

cmdAnswer = new Button();
...
grid1.Children.Add(cmdAnswer);

txtAnswer = new TextBox();
...
grid1.Children.Add(txtAnswer);
```

Что происходит после этого, полностью определяется тем, как элемент управления реализует содержимое свойства. Элемент Grid выводит на экран все элементы, расположенные на невидимом слое строк и столбцов (см. главу 3).

Проход по вложенным элементам

Инфраструктура Silverlight предоставляет класс VisualTreeHelper, позволяющий проходить по иерархии элементов. Класс VisualTreeHelper содержит три статических метода: GetParent(), возвращающий элемент, содержащий заданный элемент; GetChildrenCount(), возвращающий количество элементов, вложенных в заданный; GetChild(), извлекающий один из вложенных элементов, заданный с помощью индекса позиции.

Преимущество класса VisualTreeHelper состоит в том, что он работает унифицированным способом, поддерживая все элементы Silverlight независимо от используемой модели содержимого. Предположим, известно, что список предоставляет несколько элементов посредством свойства Items, контейнеры компоновки предоставляют дочерние элементы посредством свойства Children, а элементы содержимого предоставляют вложенные элементы посредством свойства Content (Содержимое). В этой ситуации только класс VisualTreeHelper может углубиться во все три уровня вложенности с помощью одного и того же бесшовного кода.

Недостаток класса VisualTreeHelper состоит в том, что он извлекает каждую деталь визуальной композиции элемента, включая все несущественные средства. Например, при использовании VisualTreeHelper для прохода по элементу управления ListBox вы пройдете по всем низкоуровневым деталям, которые вас не интересуют, таким как рамка Border, полоса прокрутки ScrollViewer, решетка Grid, служащая для компоновки элементов, и т.д. По этой причине класс VisualTreeHelper обычно используется только в рекурсивном коде для прохода вниз по дереву до тех пор, пока не будет найден элемент нужного типа, после чего найденный элемент обрабатывается. Ниже приведен пример использования этого метода для очистки всех текстовых полей в иерархии элементов.

```
private void Clear(DependencyObject element)
{
  // Если это текстовое поле, текст очищается
```

```
    TextBox txt = element as TextBox;
    if (txt != null) txt.Text = "";
    // Проверка вложенных элементов
    int children = VisualTreeHelper.GetChildrenCount(element);
    for (int i = 0; i < children; i++)
    {
      DependencyObject child = VisualTreeHelper.GetChild(element, i);
      Clear(child);
    }
}
```

Чтобы запустить проход, вызовите метод `Clear()` через проверяемый объект верхнего уровня. Например, для прохода по всей текущей странице нужно выполнить оператор `Clear(this);`.

События

До сих пор мы рассматривали привязку атрибутов к свойствам. Однако атрибуты можно также использовать и для подключения обработчиков событий. Для этого используется синтаксис *имя_события*="*имя_обработчика*".

Например, элемент управления `Button` (Кнопка) предоставляет событие `Click` (Щелчок). Подключить к нему обработчик можно следующим образом.

```
<Button ... Click="cmdAnswer_Click">
```

Предполагается, что в коде фонового класса определен метод `cmdAnswer_Click`. Обработчик события должен иметь правильную сигнатуру, т.е. такую же, как и делегат события `Click`. Ниже приведен пример обработчика.

```
private void cmdAnswer_Click(object sender,
                             RoutedEventArgs e)
{
  AnswerGenerator generator = new AnswerGenerator();
  txtAnswer.Text =
            generator.GetRandomAnswer(txtQuestion.Text);
}
```

Во многих ситуациях атрибуты используются для установки свойств и подключения обработчиков событий одного и того же элемента. Компилятор Silverlight всегда придерживается одной и той же последовательности: сначала он устанавливает свойство `Name` (если нужно), затем подключает обработчик и устанавливает остальные свойства. Благодаря этому при установке свойства в первый раз будет запущен любой обработчик, реагирующий на изменение свойства.

Описание приложения EightBall

Теперь, когда вы знакомы с основами XAML, можно рассмотреть определение страницы, показанной на рис. 2.1. Ниже приведена полная разметка XAML страницы.

```
<UserControl x:Class="EightBall.Page"
  xmlns="http://schemas.microsoft.com/client/2007"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Width="400" Height="300">

    <Grid x:Name="grid1">
      <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
      </Grid.RowDefinitions>
```

```
    <TextBox VerticalAlignment="Stretch"
             HorizontalAlignment="Stretch"
    Margin="10,10,13,10" x:Name="txtQuestion"
    TextWrapping="Wrap" FontFamily="Verdana" FontSize="24"
    Grid.Row="0">
      [Здесь введите вопрос]
    </TextBox>

    <Button VerticalAlignment="Top"
            HorizontalAlignment="Left"
    Margin="10,0,0,20" Width="127" Height="23"
    x:Name="cmdAnswer"
    Click="cmdAnswer_Click" Grid.Row="1">
      Программа! Ответь на вопрос!
    </Button>

    <TextBox VerticalAlignment="Stretch"
    HorizontalAlignment="Stretch"
    Margin="10,10,13,10" x:Name="txtAnswer"
    TextWrapping="Wrap"
    IsReadOnly="True" FontFamily="Verdana" FontSize="24"
    Foreground="Green"
    Grid.Row="2">
      [Здесь появится ответ]
  </TextBox>

  <Grid.Background>
    <LinearGradientBrush>
      <LinearGradientBrush.GradientStops>
        <GradientStop Offset="0.00" Color="Yellow" />
        <GradientStop Offset="0.50" Color="White" />
        <GradientStop Offset="1.00" Color="Purple" />
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Grid.Background>

  </Grid>
</UserControl>
```

Не забывайте, что для графически мощных пользовательских интерфейсов разметка XAML обычно не пишется вручную. Это было бы слишком скучно. Однако редактировать разметку XAML чаще всего приходится вручную. Кроме того, разметку XAML обычно просматривают визуально, чтобы глубже проанализировать принцип работы страницы.

Ресурсы

В инфраструктуру Silverlight включена система ресурсов, плотно интегрированная с XAML. С помощью ресурсов можно решать следующие задачи.

- **Создание невизуальных объектов.** Невизуальные объекты полезны, когда их используют другие элементы. Например, можно создать объект данных как ресурс, а затем применить связывание данных для вывода хранящейся в нем информации в других элементах.
- **Повторное использование объектов.** После определения ресурса его можно использовать в разных элементах. Например, можно определить одну

кисть, используемую для заливки разных фигур. В следующих главах книги ресурсы применяются для определения стилей и шаблонов, повторно используемых во многих элементах.

- **Централизация данных.** В некоторых ситуациях лучше сосредоточить часто изменяемую информацию в одном месте (в разделе ресурсов), а не разбрасывать по всему файлу разметки, где ее будет тяжело изменять и отслеживать.

Не путайте систему ресурсов с ресурсами сборки, которые являются блоками данных, внедряемых в скомпилированную сборку Silverlight. Например, файл XAML, добавленный в проект, является внедренным ресурсом сборки. Ресурсы сборки рассматриваются в главе 6.

Коллекция ресурсов

Каждый элемент обладает свойством Resources (Ресурсы), в котором хранится коллекция ресурсов. Коллекция ресурсов может содержать объекты любого типа, индексированные строками.

Ресурсы чаще всего определяются на уровне страницы, поскольку каждый элемент имеет доступ к ресурсам, хранящимся не только в собственной коллекции, но и в коллекциях ресурсов родительских элементов. Поэтому после определения ресурса на странице его могут использовать все элементы страницы.

Рассмотрим приложение EighBall. Объект кисти GradientBrush, рисующий фон объекта Grid, определяется и устанавливается в одном и том же месте. Однако кисть можно извлечь из разметки Grid и разместить в коллекции ресурсов.

```
<UserControl x:Class="EightBall.Page"
xmlns="http://schemas.microsoft.com/client/2007"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Width="400" Height="300">
  <UserControl.Resources>
    <LinearGradientBrush x:Key="BackgroundBrush">
      <LinearGradientBrush.GradientStops>
        <GradientStop Offset="0.00" Color="Yellow" />
        <GradientStop Offset="0.50" Color="White" />
        <GradientStop Offset="1.00" Color="Purple" />
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </UserControl.Resources>
  ...
</UserControl>
```

В этой разметке единственное важное новое средство — атрибут Key, добавленный в кисть (префикс x используется для задания пространства имен XAML вместо Silverlight). Атрибут Key присваивает кисти имя, под которым она будет индексирована в коллекции ресурсов. Допустимо любое имя, нужно лишь, чтобы оно же использовалось и для извлечения ресурса. Рекомендуется присвоить ресурсу имя исходя из его назначения (которое не будет изменяться), а не способа его реализации (который может изменяться). По этой причине имя BackgroundBrush (Кисть фона) лучше, чем LinearGradientBrush (Кисть линейного градиента) или ThreeColorBrush (Трехцветная кисть).

Примечание

В разделе ресурсов можно создать экземпляр любого класса .NET (включая пользовательские классы). Нужно лишь, чтобы класс был "дружественным" к XAML. Это означает, что класс должен обладать несколькими базовыми средствами, такими как изменяемые свойства и конструктор без аргументов.

Чтобы использовать ресурс в разметке XAML, должен существовать способ ссылки на него. Ссылка на ресурс реализуется с помощью *расширения разметки* (markup extension) — специального синтаксиса, предназначенного для установки свойств нестандартным способом. Расширения разметки дополняют язык XAML и распознаются по фигурным скобкам. Ниже приведено расширение разметки `StaticResource`, позволяющее использовать ресурс в элементе `Grid`.

```
<Grid x:Name="grid1"
        Background="{StaticResource BackgroundBrush}">
```

Такой шаблон не сокращает разметку приложения `EightBall`, однако если необходимо применять одну и ту же кисть в разных элементах, то использование ресурса — лучший способ избежать дублирования разметки. Даже если кисть задействуется только один раз, использование ресурса предпочтительнее, когда пользовательский интерфейс содержит много изменяемых графических элементов. Например, разместив все кисти в коллекции ресурсов, вы будете тратить меньше времени на их поиск и редактирование. Некоторые разработчики применяют ресурсы практически во всех сложных объектах, создаваемых в разметке XAML для установки свойств.

Примечание

Слово `Static` (Статический) отображает тот факт, что в WPF существует два типа ресурсов: статические и динамические. Однако в Silverlight поддерживаются только статические ресурсы.

Иерархия ресурсов

Каждый элемент обладает собственной коллекцией ресурсов, и Silverlight выполняет рекурсивный поиск нужного ресурса вверх по дереву элементов. Рассмотрим следующую разметку.

```
<UserControl x:Class="Resources.ResourceHierarchy"
xmlns="http://schemas.microsoft.com/client/2007"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Width="400" Height="300">
  <Grid x:Name="LayoutRoot" Background="White">

    <StackPanel>
      <StackPanel.Resources>
        <LinearGradientBrush x:Key="ButtonFace">
          <GradientStop Offset="0.00" Color="Yellow" />
          <GradientStop Offset="0.50" Color="White" />
          <GradientStop Offset="1.00" Color="Purple" />
        </LinearGradientBrush>
      </StackPanel.Resources>
      <Button Content="Первая кнопка" Margin="5"
       Background="{StaticResource ButtonFace}"></Button>
      <Button Content="Вторая кнопка" Margin="5"
       Background="{StaticResource ButtonFace}"></Button>
    </StackPanel>

  </Grid>
</UserControl>
```

На рис. 2.2 показана страница, созданная этой разметкой[1].

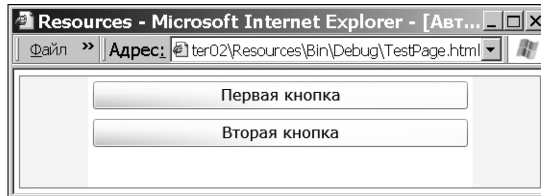

Рис. 2.2. Использование одной кисти для заливки двух кнопок

Фон обеих кнопок устанавливается на основе одного и того же ресурса. Проходя по приведенной выше разметке, Silverlight сначала проверяет коллекцию ресурсов самой кнопки, а затем — ее родительского элемента `StackPanel` (если он существует). Если же и в `StackPanel` нет нужного ресурса, Silverlight продолжает его поиск, перейдя сначала к коллекции ресурсов элемента `Grid`, а затем — элемента `UserControl`. Если и после этого ресурс с заданным именем не будет найден, Silverlight проверяет ресурсы приложения, определенные в разделе `<Application.Resources>` файла `App.xaml`. Следовательно, ресурсы приложения — самое удобное место для хранения сущностей, повторно используемых во всем приложении. В рассмотренном примере в ресурсах приложения можно хранить кисть, используемую на разных страницах.

Примечание

Перед созданием ресурсов приложения оцените баланс между сложностью и возможностью повторного использования. Ресурсы приложения можно многократно использовать во многих местах, однако их применение усложняет код, поэтому их целесообразность не очевидна. Рекомендуется использовать ресурсы приложения, если они применяются на многих страницах. Если же они нужны не более чем на двух или трех страницах, лучше определить ресурсы на уровне каждой страницы.

При определении ресурса в разметке важна последовательность элементов. Необходимо размещать определение ресурса перед его использованием в разметке. Например, если разместить раздел `<StackPanel.Resources>` после разметки, в которой объявляются кнопки, документ XAML будет валидным, однако приложение не будет работать. Встретив во время выполнения ссылку на незнакомый ресурс, синтаксический анализатор XAML сгенерирует исключение.

Обратите внимание на то, что имена ресурсов могут повторяться. Важно лишь, чтобы одноименные, но разные ресурсы находились в разных коллекциях. Если в одной коллекции два разных ресурса имеют одно имя, Silverlight применит ресурс, встретившийся первым. Например, если в предыдущем примере добавить в раздел `<UserControls.Resources>` еще одну кисть с тем же именем, она будет проигнорирована.

Обращение к ресурсам в коде

Обычно ресурсы определяются и используются в разметке. Однако при необходимости коллекции ресурсов можно обрабатывать в коде. Наиболее простой способ состоит в обращении к ресурсу, размещенному в соответствующей коллекции, по имени. Например, если кисть `LinearGradientBrush` хранится в разделе

---

[1]В папке `Chapter02\Resources` несколько примеров объединены в одном приложении. Чтобы открыть пример, запустите приложение и щелкните на соответствующей кнопке. При создании рисунка для книги меню примеров было удалено. Аналогичным образом примеры организованы и в других главах. — *Примеч. ред.*

<UserControl.Resources> под именем ButtonFace, ею можно манипулировать в коде следующим образом.

```
LinearGradientBrush brush =
         (LinearGradientBrush)this.Resources["ButtonFace"];
// Поменять цвета местами
Color color = brush.GradientStops[0].Color;
brush.GradientStops[0].Color = brush.GradientStops[2].Color;
brush.GradientStops[2].Color = color;
```

При изменении ресурса в коде автоматически обновляется каждый элемент, в котором он используется. Например, при щелчке на верхней кнопке изменяется цвет обеих кнопок (рис. 2.3).

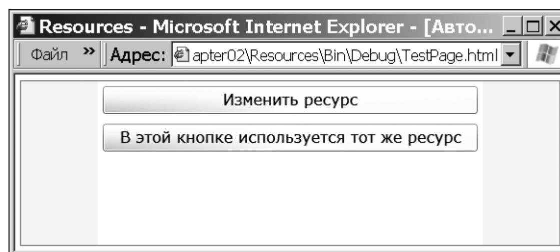

Рис. 2.3. Изменение ресурса

Существует одно важное ограничение. Платформа Silverlight не поддерживает динамические ресурсы, поэтому изменить ссылку на ресурс невозможно. Это означает, что заменить ресурс другим объектом нельзя. Ниже приведен код, в котором это правило нарушено, в результате чего генерируется ошибка времени выполнения.

```
SolidColorBrush brush = new SolidColorBrush(Colors.Yellow);
this.Resources["ButtonFace"] = brush;
```

Чтобы не просматривать коллекцию ресурсов в поиске нужного объекта, присвойте ресурсу имя с помощью атрибута Name. Затем к ресурсу можно обратиться в коде непосредственно по имени. Однако присвоить одному и тому же объекту одновременно имя и ключ невозможно. Расширение разметки StaticResource распознает только ключи. Следовательно, если создан именованный ресурс, его нельзя будет использовать в разметке с помощью ссылки StaticResource. По этой причине ключи используются чаще, чем имена.

Резюме

В этой главе рассмотрены синтаксические правила XAML на примере простого файла разметки. При разработке приложения нет необходимости писать код XAML вручную. В рабочей среде Visual Studio код XAML можно создавать, перетаскивая элементы управления в разметку. Поэтому на первый взгляд может показаться, что изучать синтаксис XAML тоже нет необходимости.

Конечно, это не так. Понимание синтаксиса XAML критически важно для создания приложений Silverlight. Только зная XAML, вы сможете понять ключевые концепции Silverlight и создавать правильную разметку. Кроме того, редактировать многие компоненты разметки XAML и решать многие задачи (в том числе подключение обработчиков к событиям, определение ресурсов, создание шаблонов элементов управления, связывание данных, создание анимации) можно только вручную.

В будущем, возможно, разработчики приложений Silverlight будут сочетать визуальные и ручные методы верстки пользовательских интерфейсов, размещая элементы управления с помощью программ Visual Studio или Expression Blend и настраивая их вручную путем редактирования разметки XAML. Однако в текущей версии Silverlight поддержка визуальных инструментов весьма ограниченная. Конечно, вы уже заметили, что элементы невозможно перетаскивать из окна инструментов в окно конструктора, как вы привыкли делать в других визуальных средах разработки. Перетаскивать их можно только в разметку. Ситуация быстро изменяется, Microsoft “не дремлет”, и в ближайшем будущем у нас будут более мощные инструменты. Тем не менее даже в отдаленном будущем разработчики несомненно будут работать как с визуальными инструментами, так и с разметкой XAML.