1. События: EVENTS

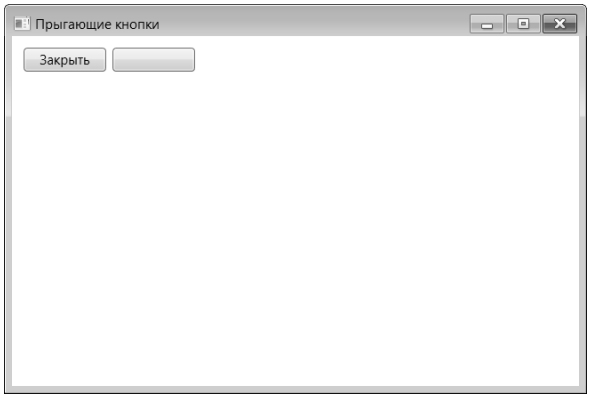

Рис. 1. Окно приложения EVENTS

1.1. Создание проекта для WPF-приложения

Для того чтобы создать проект в среде программирования Visual Studio, выполните команду File | New | Project (Ctrl+Shift+N), в появившемся окне New Project выберите в левой части вариант Visual C#, а в правой части – вариант WPF Application, в поле ввода Name укажите имя проекта (в нашем случае EVENTS), а в поле ввода Location укажите каталог, в котором будет создан каталог проекта. Желательно снять флажок Create directory for solution, чтобы не создавался промежуточный каталог для решения (каталог для решения удобно использовать в ситуации, когда решение содержит *несколько* проектов; в нашем случае решение всегда будет содержать единственный проект). После указания всех настроек нажмите кнопку «OK».

В результате будет создан каталог EVENTS, содержащий все файлы одноименного проекта, в том числе файл решения EVENTS.sln, файл проекта EVENTS.csproj, а также файлы для двух основных классов проекта, созданных автоматически: класса MainWindow, представляющего главное окно программы, и класса App, обеспечивающего запуск программы, в ходе которого создается и отображается на экране экземпляр главного окна.

Для каждого класса создаются два файла: с расширением xaml, котрый содержит часть определения класса в специальном формате, и с расширением cs (перед которым тоже содержится текст xaml), содержащий часть определения класса на языке C#. Файл с расширением xaml (*xaml-файл*) имеет формат XML (eXtensible Markup Language – расширяемый язык разметки). Аббревиатура XAML (произносится «зэмл» или «замл») означает, что используется специализированный вариант языка XML: eXtensible

Application Markup Language – расширяемый язык разметки *для приложений*.

Приведем содержимое файлов, связанных с классом App и созданных в Visual Studio 2015.

App.xaml:

```
<Application x:Class="EVENTS.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:EVENTS"
    StartupUri="MainWindow.xaml">
  <Application.Resources>

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

App.xaml.cs:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;

namespace EVENTS
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application
    {
    }
}
```

Анализ этих файлов показывает, что класс App наследуется от стандартного класса Application, а также что в cs-файле никакой новой функциональности в класс App не добавляется (обратите внимание на то, что при определении класса App в cs-файле указывается модификатор partial, означающий, что часть определения этого класса содержится в другом файле). Созданный класс (как и другие классы проекта, создаваемые автоматически) связывается с пространством имен EVENTS, совпадающим с именем проекта.

Перед анализом содержимого xaml-файла следует предварительно описать основные правила, по которым формируется любой XML-файл. Подобные файлы состоят из иерархического набора вложенных друг в друга именованных *XML-элементов*, причем каждый элемент может иметь любое количество *XML-атрибутов* и *дочерних элементов*. Элементы оформляются в виде *тегов*; открывающий тег элемента имеет вид <*имя_элемента список_атрибутов*>, а закрывающий тег – </*имя_элемента*>. Между этими тегами располагается *содержимое* элемента, которое может представлять собой обычный текст и/или другие (дочерние) элементы (а также другие *XML-узлы*, которые мы не будем обсуждать, так как в xaml-файле они не используются). Число уровней вложенности элементов может быть любым. Если элемент не имеет содержимого, то он может представляться в виде одного *комбинированного тега* вида <*имя_элемента список_атрибутов* />. Атрибуты в списке определяются следующим образом: *имя_атрибута*="*значение_атрибута*"; значение обязательно заключается в кавычки (одинарные или двойные). Все атрибуты одного элемента должны иметь *различные* имена, в то время как его дочерние элементы могут иметь совпадающие имена. Регистр в именах учитывается; имена как атрибутов, так и элементов могут содержать только буквы, цифры, символы «.» (точка), «-» (дефис) и «_» (подчеркивание) и начинаться либо с буквы, либо с символа подчеркивания. Пробелы в именах не допускаются. Перед именами элементов и атрибутов могут указываться *префиксы пространств имен*, отделяемые от собственно имени двоеточием (в файле App.xaml имеются два таких атрибута: xmlns:x и xmlns:local). Любой XML-файл должен содержать *единственный* XML-элемент верхнего уровня, называемый *корневым элементом* (в файле App.xaml это элемент Application).

В той части определения класса App, которая размещается в xaml-файле, содержится единственная, но очень важная настройка – указание на класс, экземпляр которого будет создан при запуске программы. Это атрибут StartupUri элемента Application, его значение равно MainWindow.xaml. Фактически данный атрибут является *свойством* класса Application. Как и другие свойства, его можно настроить либо непосредственно в тексте xaml-файла, либо в окне свойств Properties, которое отображает доступные для редактирования свойства текущего объекта из xaml-файла (если в редакторе отображается не xaml-, а cs-файл, то окно Properties является пустым).

При указании или изменении свойств в xaml-файле очень помогает предусмотренная в редакторе xaml-файлов возможность *контекстной подсказки* при выборе значений свойств. Окно Properties удобно в том отношении, что позволяет просмотреть *все* доступные свойства текущего объекта. В xaml-файле отображаются только те свойства, значения которых

отличаются от значений по умолчанию для данного объекта. Чтобы добавить в xaml-файл новое свойство, достаточно в окне Properties указать для данного свойства значение, отличное от значения по умолчанию.

В процессе компиляции программы все xaml-файлы конвертируются в специальный двоичный формат и затем обрабатываются совместно с cs-файлами проекта.

Класс App обычно не требуется редактировать. По этой причине после создания проекта в редактор не загружаются файлы, связанные с классом App.

Приведем файлы, связанные с классом MainWindow; именно эти файлы автоматически загружаются в редактор после создания (или открытия) проекта.

MainWindow.xaml:

```
<Window x:Class="EVENTS.MainWindow"
    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"
    xmlns:local="clr-namespace:EVENTS"
    mc:Ignorable="d"
    Title="MainWindow" Height="350" Width="525">
  <Grid>

  </Grid>
</Window>
```

MainWindow.xaml.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace EVENTS
```

```
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

В дальнейшем при выводе текста xaml-файлов мы не будем указывать атрибуты корневого элемента, предшествующие атрибуту Title, поскольку они генерируются автоматически и не требуют изменения.

Одновременно с отображением xaml-файла для класса окна на экране выводится *окно дизайнера* – визуального редактора. Следует заметить, что при разработке WPF-приложений визуальный редактор используется не так активно, как при разработке приложений, использующих библиотеку Windows Forms. Это связано с тем, что относительное расположение компонентов в окне WPF-приложения обычно не определяется явным образом, с указанием абсолютных оконных координат, а *вычисляется* по специальным правилам, связанным с особенностями тех или иных группирующих компонентов (называемых также *панелями*). Таким образом, окно дизайнера используется преимущественно для того, чтобы быстро определить, как те или иные изменения, внесенные в xaml-файл или сделанные с помощью окна свойств, повлияют на внешний вид окна. С помощью выпадающего списка можно настраивать масштаб для окна дизайнера.

Обратите внимание на то, что по умолчанию в окно приложения уже включен группирующий компонент Grid – наиболее универсальный из группирующих компонентов, позволяющий размещать свои дочерние компоненты в нескольких строках и столбцах. Кроме того, для класса MainWindow определены три свойства: Title, Height и Width. Гораздо большее число свойств приведено в окне Properties (как уже было отмечено выше, их отсутствие в xaml-файле объясняется тем, что данный файл содержит только свойства, значения которых отличаются от значений по умолчанию). При просмотре списка свойств в окне Properties можно использовать либо режим, при котором «родственные» свойства объединяются в группы, либо режим, при котором свойства располагаются в алфавитном порядке. Кроме того, с помощью поля ввода, расположенного над списком свойств, можно выполнять фильтрацию этого списка, отображая только те свойства, в именах которых содержится указанная строка.

Например, после ввода в это поле текста Height в списке свойств останутся лишь три свойства: Height, MaxHeight и MinHeight.

В файле MainWindow.xaml.cs содержится частичное определение класса MainWindow, включающее конструктор без параметров, в котором вызывается метод InitializeComponent, обеспечивающий начальную инициализацию всех компонентов окна. Все действия с компонентами можно выполнять только после их начальной инициализации, поэтому пользовательский код добавляется в конструктор *после* вызова данного метода.

1.2. Добавление компонентов и настройка их свойств

Разрабатываемое нами приложение отличается от традиционных WPF-приложений тем, что мы хотим произвольным образом перемещать отдельные компоненты в пределах окна. В подобной ситуации вместо группирующего компонента Grid удобнее пользоваться компонентом Canvas. Поэтому нам необходимо изменить «внешний» компонент окна и, кроме того, добавить на новый внешний компонент кнопку Button.

Эти действия можно выполнить двумя способами: с помощью окна дизайнера, удалив в нем лишние компоненты и добавив новые путем их перетаскивания с панели компонентов Toolbox, и с помощью непосредственного редактирования xaml-файла.

Опишем первый способ.

Вначале необходимо выделить в окне дизайнера компонент Grid, щелкнув на нем мышью. То, что выделен именно компонент Grid, можно проверить по тексту xaml-файла (в котором также будет выделен элемент <Grid>) или по окну Properties (где указываются свойства выделенного компонента). После выделения компонента его надо удалить, нажав клавишу Delete. Обратите внимание на то, что в результате такого удаления элемент Window в xaml-файле будет представлен в виде комбинированного тега <Window … />, поскольку теперь он не содержит дочерних элементов.

Затем необходимо добавить в окно компонент Canvas. Для этого надо развернуть панель Toolbox, которая обычно располагается у левой границы окна Visual Studio в свернутом состоянии. Если данная панель отсутствует, то ее можно отобразить с помощью команды меню View | Toolbox (Ctrl+W, X). Для быстрого поиска нужного компонента на панели Toolbox достаточно ввести начальную часть его имени в поле ввода, расположенное в верхней части панели. Например, в нашем случае достаточно ввести текст Can, чтобы на панели отобразился единственный компонент Canvas. Можно обойтись и без быстрого поиска, просто выбрав данный компонент в списке All WPF Controls. После выбора компонента Canvas достаточно перетащить его в окно дизайнера. В результате компонент Canvas появится в окне и соответствующий текст будет добавлен в xaml-файл (при этом бу-

дет восстановлено представление элемента Window в xaml-файле в виде двух тегов – открывающего <Window> и закрывающего </Window>):

```
<Window x:Class="EVENTS.MainWindow"
    …
    Title="MainWindow" Height="350" Width="525">
  <Canvas HorizontalAlignment="Left" Height="100"
     Margin="150,87,0,0" VerticalAlignment="Top" Width="100"/>
</Window>
```

Здесь и в дальнейшем мы часто будем опускать фрагменты xaml-файла, оставшиеся неизменными, указывая вместо них символ многоточия «…». Измененную часть xaml-файла мы выделили полужирным шрифтом.

Примерный вид окна дизайнера приведен на рис. 2.

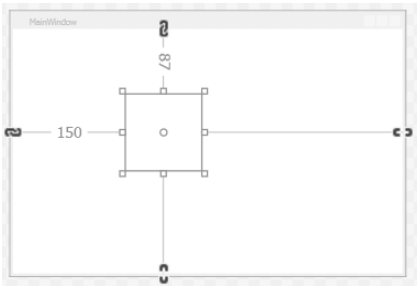

Рис. 2. Окно дизайнера после добавления компонента Canvas

Разумеется, нам не требуется такое размещение компонента Canvas. Необходимо, чтобы он занимал всю клиентскую область окна. Для того чтобы добиться этого, достаточно просто *удалить* в xaml-файле все атрибуты элемента Canvas (удаляемые фрагменты будем изображать перечеркнутыми):

```
<Canvas HorizontalAlignment="Left" Height="100"
    Margin="150,87,0,0" VerticalAlignment="Top" Width="100" />
```

Новый вид окна дизайнера приведен на рис. 3.

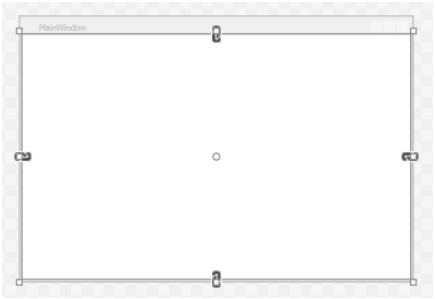

Рис. 3. Окно дизайнера после удаления атрибутов компонента Canvas

Как правило, после добавления в окно какого-либо компонента путем его перетаскивания из панели Toolbox, всегда требуется выполнить действия, связанные с удалением «лишних» атрибутов.

Теперь добавим на компонент Canvas кнопку Button, зацепив ее мышью на панели Toolbox и перетащив в окно. После появления кнопки в ок-

не следует перетащить ее в левый верхний угол окна (при подобном перетаскивании кнопка будет автоматически «притянута» к области, расположенной на расстоянии 10 единиц от левой и верхней границы клиентской области окна, – рис. 4).

Рис. 4. Окно дизайнера после добавления компонента Button

Содержимое xaml-файла изменится следующим образом:

```
<Canvas >
  <Button x:Name="button" Content="Button" Canvas.Left="10"
      Canvas.Top="10" Width="75"/>
</Canvas>
```

Мы видим, что теперь элемент Canvas тоже оформляется в виде парных тегов, так как он содержит дочерний элемент – кнопку.

Обсудим атрибуты, автоматически добавленные к элементу Button. Атрибут с именем x:Name определяет *имя*, с помощью которого можно обращаться к данному компоненту в cs-файле. Это имя будет являться одним из свойств класса MainWindow. Обратите внимание на то, что элемент Canvas аналогичного имени не содержит. Это означает, что в классе MainWindow мы не сможем обращаться по имени к компоненту Canvas. Если это является неудобным, то всегда можно определить имя (или с помощью окна свойств, в котором свойство Name указывается первым, или непосредственно в xaml-файле).

Свойство Content определяет *содержимое* кнопки. В качестве значения свойства Content может указываться не только строка, но и любой компонент. Более того, на кнопку можно поместить группирующий компонент, в котором, в свою очередь, можно разместить любое количество других компонентов. Это позволяет создавать в WPF-приложении сложные интерфейсные элементы, конструируя их из базовых. Например, можно создать кнопку, содержащую не только текст, но и изображение (в дальнейшем мы воспользуемся этой возможностью – см., например, проект ZOO, п. 7.7).

Следует также обратить внимание на то, что для кнопки не указано свойство Height (хотя свойство Width имеется). Если свойство Height отсутствует, то высота компонента определяется по размерам его содержимого, что в большинстве случаев является оптимальным. Можно было бы

удалить и свойство Width, тогда все размеры кнопки будут подстроены под ее содержимое, однако обычно свойство Width указывается, поскольку желательно, чтобы все кнопки в приложении имели одинаковую ширину.

В отличие от компонентов из библиотеки Windows Forms, компоненты библиотеки WPF не имеют свойств Top и Left, определяющих позицию, в которой они размещаются. Это связано с тем, что явное указание позиции компонентов в окне WPF обычно не требуется (положение компонентов определяется другими их свойствами, а также свойствами содержащих их группирующих компонентов). Однако для любого компонента можно задать свойства Top и Left, «полученные» от класса Canvas. Если данный компонент будет размещен на одном из компонентов типа Canvas, то эти полученные свойства будут учтены при определении его позиции. Возможность подобной «передачи» свойств от одного компонента к другому является одним из аспектов особого механизма, реализованного в WPF и связанного с так называемыми *свойствами зависимости* (dependency properties). Почти все свойства компонентов WPF являются свойствами зависимости, что позволяет их использовать при реализации различных возможностей, доступных в WPF, например, для привязки свойств или определения стилей. Частным случаем свойств зависимости являются *присоединенные свойства* (attached properties), которые, будучи определенными в одном классе, могут использоваться в другом. Свойства Top и Left компонента Canvas – типичный пример присоединенных свойств.

Присоединенные свойства панели Canvas особенно просто указывать в xaml-файле. Однако к ним можно обращаться и в программном коде, что будет продемонстрировано далее.

Итак, в результате добавления новых компонентов в окно наш xaml-файл изменился следующим образом:

```
<Window x:Class="EVENTS.MainWindow"
    …
    Title="MainWindow" Height="350" Width="525">
  <Canvas >
    <Button x:Name="button" Content="Button" Canvas.Left="10"
        Canvas.Top="10" Width="75"/>
  </Canvas>
</Window>
```

Того же результата можно было достичь, просто введя данный текст в xaml-файл, хотя на практике оказывается более удобным добавлять новый компонент с помощью панели Toolbox, а уже затем редактировать связанный с ним текст, добавленный в xaml-файл.

Отредактируем полученный xaml-файл: изменим заголовок окна на текст «Прыгающие кнопки», надпись на кнопке – на «Закрыть», ее имя – на button1 (поскольку в дальнейшем мы добавим к окну еще одну

кнопку). Кроме того, укажем для окна свойство WindowStartupLocation, положив его равным CenterScreen (это значение обеспечивает автоматическое центрирование окна программы при ее запуске):

```
<Window x:Class="EVENTS.MainWindow"
    …
    Title="Прыгающие кнопки" Height="350" Width="525"
    WindowStartupLocation="CenterScreen">
  <Canvas >
    <Button x:Name="button1" Content="Закрыть" Canvas.Left="10"
        Canvas.Top="10" Width="75"/>
  </Canvas>
</Window>
```

Хотя свойства можно настраивать с помощью окна Properties, обычно бывает удобнее это делать непосредственно в xaml-файле. Более того, даже добавлять новые свойства в xaml-файл не составляет труда, так как при вводе уже нескольких начальных символов свойства появляется список всех свойств, начинающихся с этих символов, что позволяет быстро завершить ввод имени, нажав клавишу Tab, после чего в xaml-файл будет не только добавлено полное имя свойства, но и вставлены символы ="", а если свойство принимает фиксированный набор значений, то сразу отобразится список этих значений, из которых можно выбрать требуемый (мы могли это заметить, определяя свойство WindowStartupLocation).

В дальнейшем при описании действий, которые требуется выполнить для добавления в окно новых компонентов или изменения их свойств, мы будем просто указывать новое содержимое xaml-файла, выделяя в нем **полужирным шрифтом** новые или измененные фрагменты. Иногда (достаточно редко) мы будем также дополнительно помечать фрагменты, которые требуется удалить, оформляя их в виде ~~перечеркнутого текста~~. Аналогичные способы выделения будем использовать и для фрагментов программного кода на языке C#.

Результат. После запуска программы (для которого достаточно нажать клавишу F5) в центре экрана появится ее окно с кнопкой «Закрыть» (рис. 5).

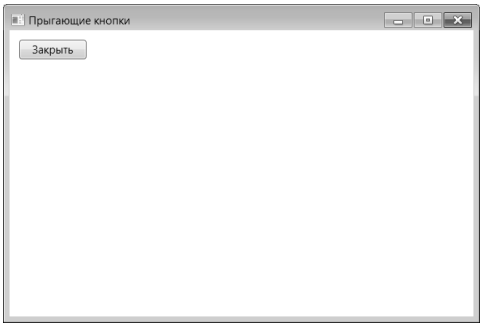

Рис. 5. Окно приложения EVENTS (первый вариант)

Нажатие на кнопку пока не приводит ни к каким действиям, однако уже сейчас для пользователя доступны все стандартные действия, связанные с управлением окном (сворачиванием, разворачиванием, закрытием, изменением размеров и положения).

Комментарий

При запуске WPF-приложения из среды Visual Studio в режиме Debug поверх окна отображается черная панель с дополнительными средствами отладки (рис. 6).

Рис. 6. Панель с дополнительными отладочными средствами XAML

Поскольку мы не будем использовать эти средства, имеет смысл скрыть панель. Для этого следует выполнить команду меню Tools | Options, в появившемся диалоговом окне Options выбрать раздел Debugging и в этом разделе *снять* флажок Enable UI Debugging Tools for XAML.

1.3. Связывание события с обработчиком

Теперь мы хотим связать определенное действие с нажатием кнопки button1. Для этого можно выполнить следующие шаги:

1) выделите в окне дизайнера кнопку button1;
2) в окне Properties перейдите к разделу со списком событий, нажав на кнопку с изображением молнии:  ;
3) выберите в разделе со списком событий строку Click и выполните на ее пустом поле ввода двойной щелчок мышью;
4) в результате активизируется вкладка редактора с файлом MainWindow.xaml.cs, где появится заготовка для нового метода класса MainWindow – обработчик события Click для компонента button1:

```
private void button1_Click(object sender, RoutedEventArgs e)
{

}
```

5) в эту заготовку надо ввести код, который будет выполняться при нажатии кнопки button1; мы добавим в нее единственный оператор:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    Close();
}
```

Заметим, что соответствующее изменение будет внесено и в xaml-файл:

```
<Button x:Name="button1" Content="Закрыть" Canvas.Left="10"
    Canvas.Top="10" Width="75" Click="button1_Click"/>
```

Именно благодаря заданию атрибута Click в xaml-файле метод button1_Click будет связан с событием Click компонента button1 (при отсутствии такого атрибута метод button1_Click будет считаться обычным методом класса, для выполнения которого требуется его явный вызов).

Описанный выше способ создания нового обработчика события был реализован еще для библиотеки Windows Forms. Однако в WPF-проекте имеется более быстрый способ определения нового обработчика, не требующий использования окна Properties. Необходимо ввести имя события как атрибут соответствующего элемента в xaml-файле (в нашем случае в элемент Button надо ввести текст «Click=», причем достаточно набрать несколько начальных символов имени события и воспользоваться для завершения набора выпадающим списком) и после появления рядом с набранным атрибутом выпадающего списка с текстом «New Event Handler» выбрать этот текст (если он еще не выбран) и нажать клавишу Enter. При этом в xaml-файл будет добавлено имя обработчика (в нашем случае button1_Click), а в cs-файле будет создана заготовка для обработчика с этим именем, *хотя перехода к ней не произойдет*, чтобы дать возможность продолжить редактирование xaml-файла. Если в программе уже имеются обработчики, совместимые с тем событием, имя которого введено в xaml-файле, то в выпадающем списке наряду с вариантом «New Event Handler» будут приведены и имена всех таких обработчиков, что позволит быстро связать события для *нескольких* компонентов с *одним* обработчиком (хотя для подобного связывания имеется более удобная возможность, основанная на механизме *маршрутизируемых событий* и описанная в проекте CALC).

Если для какого-либо компонента предполагается определять обработчики, то рекомендуется предварительно задать *имя* для этого компонента, чтобы оно включалось в имена созданных обработчиков.

В дальнейшем вместо детального описания действий по созданию обработчиков событий мы будем просто приводить измененный фрагмент xaml-файла с новыми атрибутами и текст самого обработчика, выделяя добавленные (или измененные) фрагменты полужирным шрифтом:

```
<Button x:Name="button1" … Click="button1_Click" />
```

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    Close();
}
```

Текст button1_Click мы не только выделяем **полужирным** шрифтом, но и подчеркиваем, чтобы отметить то обстоятельство, что этот текст будет автоматически сгенерирован редактором xaml-файлов после ввода текста Click= и выбора из появившегося списка варианта «New Event Handler» (напомним, что при этом в cs-файле будет создан новый обработчик с указанным именем).

Добавим к нашему проекту еще один обработчик – на этот раз для компонента Canvas (обратите внимание на то, что если обработчик создается для компонента, не имеющего имени, то в имени обработчика по умолчанию используется имя класса этого компонента):

```
<Canvas MouseDown="Canvas_MouseDown" >
```

```
private void Canvas_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    Point p = e.GetPosition(this);
    Canvas.SetLeft(button1, p.X - button1.ActualWidth / 2);
    Canvas.SetTop(button1, p.Y - button1.ActualHeight / 2);
}
```

Кроме того, необходимо задать фон для компонента Canvas:

```
<Canvas MouseDown="Canvas_MouseDown" Background="White" >
```

Результат. Теперь после запуска программы при щелчке на любом месте окна кнопка «Закрыть» услужливо прыгает на указанное место. Нажатие на кнопку «Закрыть» приводит к завершению программы и возврату в среду Visual Studio.

1.4. Отсоединение обработчика от события

В начало описания класса MainWindow (перед конструктором public MainWindow()) добавьте новое поле:

```
Random r = new Random();
```

В окно добавьте новую кнопку button2, сделайте ее свойство Content пустой строкой и определите для этой кнопки два обработчика:

```
<Canvas MouseDown="Canvas_MouseDown" Background="White" >
    …
  <Button x:Name="button2" Content="" Canvas.Left="90"
      Canvas.Top="10" Width="75" MouseMove="button2_MouseMove"
      Click="button2_Click" />
</Canvas>
```

```
private void button2_MouseMove(object sender, MouseEventArgs e)
{
    if (Keyboard.IsKeyDown(Key.LeftCtrl)
      || Keyboard.IsKeyDown(Key.RightCtrl))
```

```
        return;
    Point p = e.GetPosition(this);
    Canvas.SetLeft(button2, r.NextDouble() *
      ((Content as Canvas).ActualWidth - 5));
    Canvas.SetTop(button2, r.NextDouble() *
      ((Content as Canvas).ActualHeight - 5));
}
private void button2_Click(object sender, RoutedEventArgs e)
{
    button2.Content = "Изменить";
    button2.MouseMove -= button2_MouseMove;
}
```

Результат. «Дикая» кнопка с пустым заголовком не дает на себя нажать, «убегая» от курсора мыши. Для того чтобы ее «приручить», надо переместить на нее курсор, держа нажатой клавишу Ctrl. После щелчка на дикой кнопке она приручается: на ней появляется заголовок «Изменить», и она перестает убегать от курсора мыши. Следует заметить, что приручить кнопку можно и с помощью клавиатуры, переместив на нее фокус с помощью клавиш со стрелками (или клавиши Tab) и нажав на клавишу пробела.

Недочет. Если попытаться «приручить» кнопку, переместив на нее фокус и нажав клавишу пробела, то перед приручением она прыгает по окну, пока не будет отпущена клавиша пробела. Причины такого поведения непонятны, поскольку нажатие клавиши пробела не должно приводить к активизации события, связанного с перемещением мыши. Следует, однако, отметить, что нажатие пробела обрабатывается в WPF особым образом, и по этой причине оно может приводить к таким странным эффектам.

Исправление. Дополните условие в методе button2_MouseMove:

```
private void button2_MouseMove(object sender, MouseEventArgs e)
{
    if (Keyboard.IsKeyDown(Key.LeftCtrl)
      || Keyboard.IsKeyDown(Key.RightCtrl)
      || Keyboard.IsKeyDown(Key.Space))
        return;
    Point p = e.GetPosition(this);
    Canvas.SetLeft(button2, r.NextDouble() *
      ((Content as Canvas).ActualWidth - 5));
    Canvas.SetTop(button2, r.NextDouble() *
      ((Content as Canvas).ActualHeight - 5));
}
```

Прирученная кнопка пока ничего не делает. Это будет исправлено в следующем пункте.

1.5. Присоединение к событию другого обработчика

Для того чтобы прирученная кнопка при нажатии на нее выполняла какие-либо действия, можно добавить эти действия к уже имеющемуся обработчику button2_Click. Однако в этом случае обработчик должен проверять, в каком состоянии находится кнопка – диком или прирученном. Поступим по-другому: свяжем событие Click для прирученной кнопки *с новым обработчиком*. Такой подход позволит продемонстрировать в нашем проекте ряд особенностей, связанных с действиями по присоединению и отсоединению обработчиков.

Новый обработчик (назовем его button2_Click2) создадим «вручную», не прибегая к услугам окна Properties или xaml-файла. Для этого в конце описания класса MainWindow в файле MainWindow.xaml.cs (*перед* двумя последними скобками «}») добавим описание этого обработчика:

```
private void button2_Click2(object sender, RoutedEventArgs e)
{
    WindowState = WindowState == WindowState.Normal ?
      WindowState.Maximized : WindowState.Normal;
}
```

Чтобы подчеркнуть, что в данном случае никакая часть обработчика не создается автоматически, мы выделили весь текст обработчика полужирным шрифтом.

В метод button2_Click добавьте следующие операторы (здесь и далее в книге предполагается, что если место добавления не уточняется, то операторы надо добавлять в *конец* метода):

```
button2.Click -= button2_Click;
button2.Click += button2_Click2;
```

В метод Canvas_MouseDown добавьте операторы:

```
if ((string)button2.Content == "Изменить")
{
    button2.Content = "";
    button2.MouseMove += button2_MouseMove;
    button2.Click += button2_Click;
    button2.Click -= button2_Click2;
}
```

Результат. Прирученная кнопка теперь выполняет полезную работу – щелчок на ней приводит к разворачиванию окна программы на весь экран, а новый щелчок восстанавливает первоначальное состояние окна. Если же щелкнуть мышью на окне (не на кнопке), то услужливая кнопка «Закрыть»

прибежит на вызов, а прирученная кнопка «Изменить» снова одичает, потеряет текст своего заголовка и начнет убегать от мыши.

Недочет. При выполнении программы может возникнуть ситуация, когда одна или обе кнопки не будут отображаться в окне (если, например, кнопки были перемещены на новое место при развернутом окне, после чего окно возвращено в исходное состояние).

Исправление. Определите для окна обработчик события SizeChanged:

```
<Window x:Class="EVENTS.MainWindow"
    …
    Title="Прыгающие кнопки" Height="350" Width="525"
    WindowStartupLocation="CenterScreen"
    SizeChanged="Window SizeChanged" >
```

```
private void Window_SizeChanged(object sender,
  SizeChangedEventArgs e)
{
    var c = Content as Canvas;
    for (int i = 0; i < 2; i++)
    {
        var b = FindName("button" + (i + 1)) as Button;
        if (Canvas.GetLeft(b) > c.ActualWidth ||
          Canvas.GetTop(b) > c.ActualHeight)
        {
            Canvas.SetLeft(b, 10 + i * (b.ActualWidth + 10));
            Canvas.SetTop(b, 10);
        }
    }
}
```

Результат. Теперь в ситуации, когда при изменении размера окна его кнопки оказываются вне клиентской части, происходит перемещение этих кнопок на исходные позиции около левого верхнего угла окна.