1. События: EVENTS

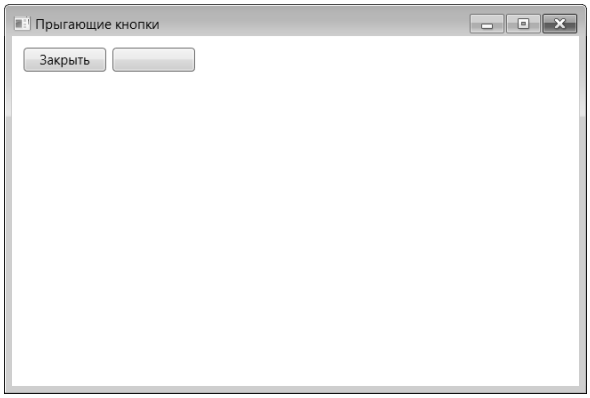

Рис. 1. Окно приложения EVENTS

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, содержащий 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 имеет вид </*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* <*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, значение которого 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 (в котором указываются свойства выделенного компонента). После выделения компонента надо его 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 надо 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 будет 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.

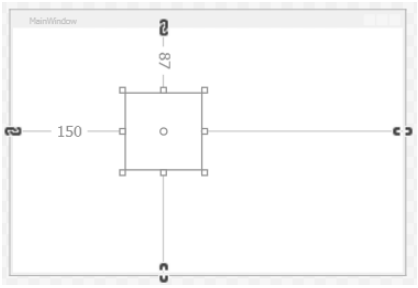

Рис. 2. Окно дизайнера после добавления компонента Canvas

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 окне следует перетащить ее в левый верхний угол окна (при подобном перетаскивании кнопка будет автоматически «притянута» к области, располо-

женной на расстоянии 10 единиц от левой и верхней границы клиентской области окна, — см. 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 мы всегда можем 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 являются типичным примером 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 позволит 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 мы 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» (см. рис. 5).

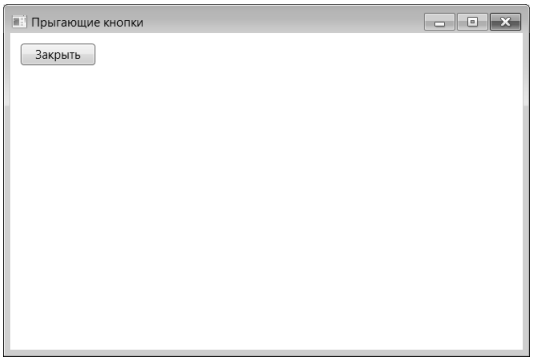

Рис. 5. Окно приложения EVENTS (первый вариант)

Комментарий

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

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).

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 на раздел 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, в которой 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. Надо 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 тек-

ста 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 об-

работчику 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.

2. Работа с несколькими окнами: WINDOWS

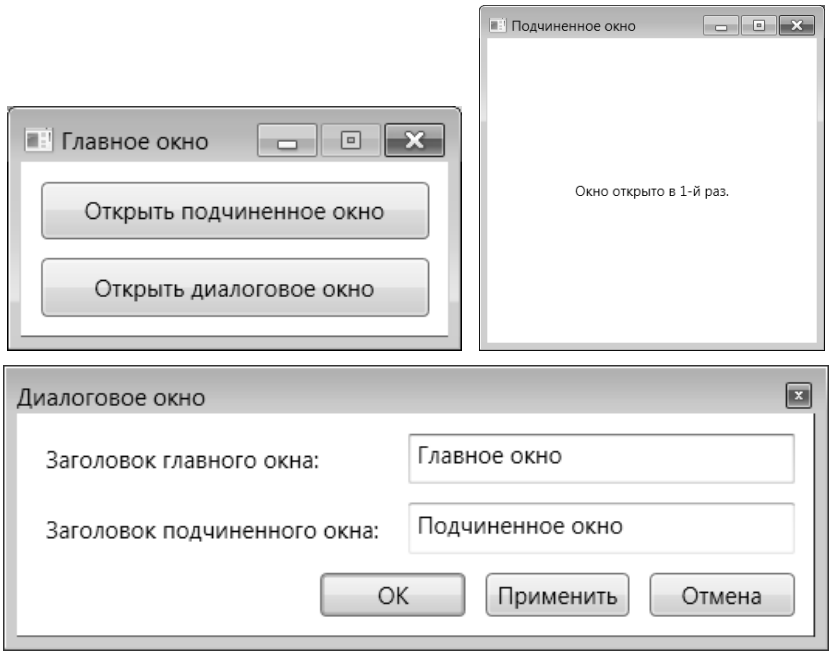

Рис. 7. Окна приложения WINDOWS

2.1. Настройка визуальных свойств окон. Открытие окон в обычном и диалоговом режиме

После создания проекта к нему необходимо добавить два дополнительных окна. Для этого требуется выполнить команду Project | Add Window… и в появившемся диалоговом окне указать имя класса, который будет связан с новым окном. Достаточно использовать имена, предлагаемые по умолчанию: Window1 для первого окна, Window2 для второго.

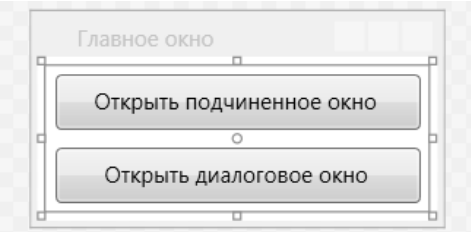

Рис. 8. Макет окна MainWindow приложения WINDOWS

MainWindow.xaml:

```
<Window x:Class="WINDOWS.MainWindow"
    …
    Title="Главное окно" SizeToContent="WidthAndHeight"
    ResizeMode="CanMinimize" Loaded="Window_Loaded" >
```

```
  <StackPanel Margin="5">
    <Button x:Name="button1" MinWidth="200" Margin="5"
        Content="Открыть подчиненное окно"
        Padding="5" Click="button1_Click" />
    <Button x:Name="button2" Margin="5"
        Content="Открыть диалоговое окно" Padding="5"
        Click="button2_Click" />
  </StackPanel>
</Window>
```

Window1.xaml:

```
<Window x:Class="WINDOWS.Window1"
    …
    Title="Подчиненное окно" Height="300" Width="300"
    ShowInTaskbar="False" >
  <Grid>

  </Grid>
</Window>
```

Window2.xaml:

```
<Window x:Class="WINDOWS.Window2"
    …
    Title="Диалоговое окно" Height="300" Width="300"
    WindowStartupLocation="CenterScreen" ResizeMode="NoResize"
    ShowInTaskbar="False" WindowStyle="ToolWindow" >
  <Grid>

  </Grid>
</Window>
```

В файле MainWindow.xaml.cs в начало описания класса MainWindow добавьте операторы:

```
Window1 win1 = new Window1();
Window2 win2 = new Window2();
```

Определите обработчики для класса MainWindow (эти обработчики указаны в файле MainWindow.xaml и поэтому их заготовки уже должны содержаться в классе MainWindow; напомним, что для большей наглядности мы подчеркиваем в xaml-файле имена подобных обработчиков):

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    win1.Owner = this;
    win2.Owner = this;
    win1.Left = this.Left + this.ActualWidth - 10;
```

```
    win1.Top = this.Top + this.ActualHeight - 10;
}
private void button1_Click(object sender, RoutedEventArgs e)
{
    win1.Show();
}
private void button2_Click(object sender, RoutedEventArgs e)
{
    win2.ShowDialog();
}
```

Результат. Программа включает три окна, демонстрирующие основные типы окон в графических Windows-приложениях: *окно фиксированного размера* (MainWindow), *окно переменного размера* (win1 типа Window1), *диалоговое окно* (win2 типа Window2). Главное окно MainWindow сразу отображается на экране при запуске приложения. Окна win1 и win2 (*подчиненные* окна) вызываются из главного окна нажатием соответствующей кнопки. При этом окно win1 отображается в обычном, а окно win2 — в *модальном* (*диалоговом*) режиме (если некоторое окно в приложении находится в диалоговом режиме, то до его закрытия нельзя переключаться на другие окна). Для завершения программы надо закрыть ее главное окно. При отображении главного окна место для его размещения выбирается операционной системой, окно win1 отображается около правого нижнего угла главного окна с небольшим наложением, окно win2 отображается в центре экрана.

Следует заметить, что полученная программа содержит серьезную ошибку, которая будет исправлена в следующем пункте.

2.2. Решение проблем, возникающих при повторном открытии подчиненных окон

Ошибка. После закрытия окна win1 или win2 попытка его повторного открытия приводит к исключению с диагностикой «*Нельзя задать Visibility или вызвать Show, ShowDialog или WindowInteropHelper.EnsureHandle после закрытия окна*»). Это связано с тем, что закрытие окна, открытого в *любом* режиме, приводит к его разрушению (заметим, что в библиотеке Windows Forms подобная ситуация имеет место только для окон, открытых в обычном режиме, а разрушения окон, открытых в диалоговом режиме, не происходит).

Исправление. Для классов Window1 и Window2 определите следующие *одинаковые* обработчики события Closing:

```
<Window x:Class="WINDOWS.Window1"
    …
    Closing="Window Closing" >
```

```
<Window x:Class="WINDOWS.Window2"
    …
    Closing="Window_Closing" >
```

Window1.xaml.cs и Window2.xaml.cs:

```
private void Window_Closing(object sender,
  System.ComponentModel.CancelEventArgs e)
{
    e.Cancel = true;
    Hide();
}
```

Результат. Теперь окна win1 и win2, можно многократно закрывать и открывать в ходе выполнения программы.

2.3. Контроль за состоянием подчиненного окна. Воздействие подчиненного окна на главное

Для окна MainWindow измените обработчик button1_Click:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    if (win1.IsVisible)
        win1.Close();
    else
        win1.Show();
}
```

Для окна Window1 определите обработчик события IsVisibleChanged:

```
<Window x:Class="WINDOWS.Window1"
    …
    IsVisibleChanged="Window_IsVisibleChanged" >
```

```
private void Window_IsVisibleChanged(object sender,
  DependencyPropertyChangedEventArgs e)
{
    (Owner.FindName("button1") as Button).Content = IsVisible ?
      "Закрыть подчиненное окно" : "Открыть подчиненное окно";
}
```

Результат. Заголовок кнопки button1 главного окна и действия при ее нажатии зависят от того, отображается на экране подчиненное окно win1 или нет. Подчиненное окно можно закрыть не только с помощью кнопки button1 главного окна, но и любым стандартным способом, принятым в Windows (например, с помощью комбинации клавиш Alt+F4); при *любом* способе закрытия подчиненного окна заголовок кнопки button1 будет изменен. Подчеркнем, что изменять надпись на кнопке button1 в обработчике

button1_Click не следует именно по той причине, что закрыть подчиненное окно можно не только с помощью этой кнопки.

2.4. Окно с содержимым в виде обычного текста

```
<Window x:Class="WINDOWS.Window1"
    … >
  <TextBlock x:Name="textBlock" HorizontalAlignment="Center"
      VerticalAlignment="Center"/>
</Window>
```

В начало описания класса Window1 добавьте поле

```
int count;
```

В имеющийся в классе Window1 обработчик Window_IsVisible-Changed добавьте следующий фрагмент:

```
if (IsVisible)
    textBlock.Text = "Окно открыто в " + (++count) + "-й раз.";
```

Результат. Текст подчиненного окна win1 содержит информацию о том, сколько раз оно было открыто. При изменении размеров подчиненного окна положение находящегося на нем текста изменяется так, чтобы он всегда оставался отцентрированным как по горизонтали, так и по вертикали относительно границ окна.

2.5. Модальные и обычные кнопки диалогового окна

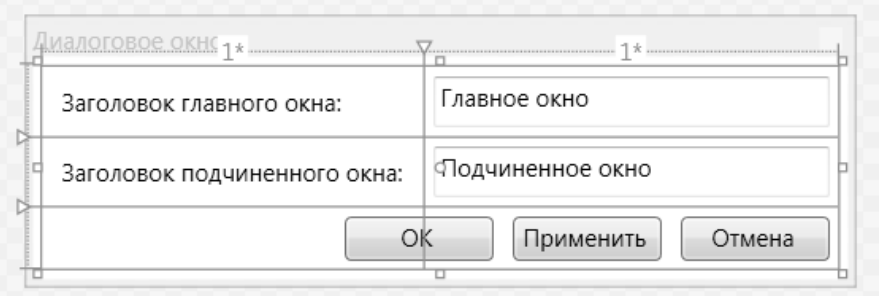

Рис. 9. Макет окна Window2 приложения WINDOWS

```
<Window x:Class="WINDOWS.Window2"
    …
    SizeToContent="WidthAndHeight"
    IsVisibleChanged="Window IsVisibleChanged" >
  <Grid Margin="5">
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
```

```
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Label Content="Заголовок главного окна:" Margin="5" />
    <Label Content="Заголовок подчиненного окна:" Margin="5"
        Grid.Row="1" />
    <TextBox x:Name="textBox1" Grid.Column="1" Margin="5"
        Text="Главное окно" MinWidth="200" />
    <TextBox x:Name="textBox2" Grid.Column="1" Margin="5"
        Grid.Row="1" Text="Подчиненное окно" />
    <StackPanel Grid.ColumnSpan="2"
        HorizontalAlignment="Right" Margin="0" Grid.Row="2"
        Orientation="Horizontal">
      <Button x:Name="button1" Content="OK" Width="75"
          Margin="5" IsDefault="True"
          Click="button1 Click" />
      <Button x:Name="textBox2" Content="Применить"
          Width="75" Margin="5" Click="button2 Click" />
      <Button Content="Отмена" Width="75" Margin="5"
          IsCancel="True" />
    </StackPanel>
  </Grid>
</Window>
```

В описание класса Window2 добавьте новое свойство, доступное только для чтения, и связанное с ним поле:

```
bool dialogRes;

public bool DialogRes
{
    get { return dialogRes; }
}
```

Определите три обработчика, которые уже указаны в xaml-файле:

```
private void Window_IsVisibleChanged(object sender,
  DependencyPropertyChangedEventArgs e)
{
    if (IsVisible)
        dialogRes = false;
}
private void button1_Click(object sender, RoutedEventArgs e)
{
```

```
    dialogRes = true;
    Close();
}
public void button2_Click(object sender, RoutedEventArgs e)
{
    Owner.Title = textBox1.Text;
    Owner.OwnedWindows[0].Title = textBox2.Text;
}
```

Обратите внимание на то, что обработчик button2_Click должен иметь модификатор public (он выделен в тексте полужирным шрифтом).

В классе MainWindow дополните обработчик button2_Click:

```
private void button2_Click(object sender, RoutedEventArgs e)
{
    win2.ShowDialog();
    if (win2.DialogRes)
        win2.button2_Click(null, null);
}
```

Результат. Диалоговое окно win2 позволяет изменить заголовки главного и подчиненного окна. Заголовки окон изменяются либо при нажатии обычной кнопки «Применить», либо при нажатии *модальной* кнопки «OK» (в последнем случае диалоговое окно закрывается). Окно также закрывается при нажатии модальной кнопки «Отмена»; в этом случае заголовки окон не изменяются. Вместо кнопки «OK» можно нажать клавишу Enter, вместо кнопки «Отмена» — клавишу Esc.

Недочет. При первом отображении диалогового окна в нем отсутствует активный компонент (т. е. элемент, имеющий фокус). В дальнейшем при закрытии и последующем открытии диалогового окна в нем будет активным тот компонент, который имел фокус в момент закрытия. Оба эти обстоятельства затрудняют работу с диалоговым окном. В частности, при повторном открытии диалогового окна его активным компонентом с большой долей вероятности будет кнопка «OK» или «Отмена» (если предыдущее закрытие окна было выполнено путем нажатия на эту кнопку), что потребует от пользователя лишних действий для перехода к тому полю ввода, которое он хочет изменить. Этот недочет будет исправлен в п. 2.6.

2.6. Установка активного компонента окна. Особенности работы с фокусом в библиотеке WPF

В классе Window2 добавьте в метод Window_IsVisibleChanged следующий оператор:

```
textBox1.Focus();
```

Результат. При первом открытии диалогового окна фокус ввода принимает компонент textBox1. Этот же компонент оказывается активным и

при последующих открытиях диалогового окна, независимо от того, какой компонент окна был активным в момент его закрытия. Таким образом, *диалоговое окно всегда отображается в одном и том же начальном состоянии*. Подобное поведение желательно обеспечивать для любых диалоговых окон.

2.7. Запрос на подтверждение закрытия окна

В классе Window1 измените обработчик Window_Closing:

```
private void Window_Closing(object sender,
  System.ComponentModel.CancelEventArgs e)
{
    e.Cancel = true;
    if (MessageBox.Show("Закрыть подчиненное окно?",
      "Подтверждение", MessageBoxButton.YesNo,
      MessageBoxImage.Question, MessageBoxResult.No) ==
      MessageBoxResult.Yes)
        Hide();
}
```

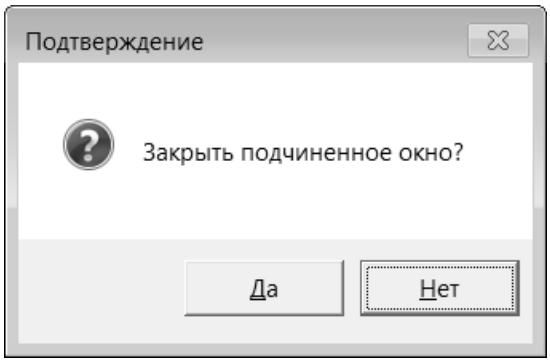

Рис. 10. Стандартное диалоговое окно

Результат. Перед закрытием подчиненного окна win1 отображается стандартное диалоговое окно «Подтверждение» с запросом на подтверждение закрытия (см. рис. 10). При выборе варианта «Нет» (который предлагается по умолчанию) закрытие подчиненного окна отменяется.

Недочет 1. При выборе в диалоговом окне варианта «Да» подчиненное окно закрывается, но главное окно не становится активным.

Данный недочет объясняется тем обстоятельством, что «владельцем» диалогового окна MessageBox является то окно, которое было активным в момент отображения на экране окна MessageBox (в нашем случае это подчиненное окно win1), и именно это окно должно активизироваться при закрытии окна MessageBox. Однако при выборе варианта «Да» окно win1 закрывается, и поэтому его активизация оказывается невозможной. В подобной ситуации *ни одно окно на экране не будет активным*, а главное окно нашей программы, скорее всего, будет скрыто окном среды Visual Studio.

Одним из вариантов исправления подобного недочета является *явное указание* владельца окна MessageBox в дополнительном параметре, который должен располагаться *первым* в списке параметров. Например, в качестве этого параметра можно указать Owner. В этом случае при выборе варианта «Да» будет успешно активизировано главное окно. Однако это же окно будет активизироваться и при выборе варианта «Нет» (когда подчиненное окно останется на экране), что является неестественным.

Исправление. Замените оператор Hide() в методе Window_Closing класса Window1 на следующий составной оператор:

```
{
  Hide();
  Owner.Activate();
}
```

Недочет 2. Если в программе ни разу не отображалось подчиненное окно, то при закрытии главного окна выводится запрос на подтверждение закрытия подчиненного окна, хотя это окно на экране отсутствует.

Исправление. Добавьте *в начало* метода Window_Closing класса Window1 следующий фрагмент:

```
if (!IsVisible)
    return;
```

3. Совместное использование обработчиков событий и работа с клавиатурой: CALC

Рис. 11. Окно приложения CALC

3.1. Настройка коллективного обработчика событий

Рис. 12. Макет окна MainWindow

```
<Window x:Class="CALC.MainWindow"
    …
    Title="Calculator" SizeToContent="WidthAndHeight"
    WindowStartupLocation="CenterScreen"
    ResizeMode="CanMinimize" >
  <StackPanel MinWidth="500">
    <StackPanel Margin="10,10,10,0" Orientation="Horizontal" >
      <TextBox x:Name="textBox1" MinWidth="120" Text="0" />
      <Label x:Name="label1" Width="35" Content="+"
          HorizontalContentAlignment="Center" />
      <TextBox x:Name="textBox2" MinWidth="120" Text="0"/>
      <Label x:Name="label2" Content="=" Margin="10,0,10,0" />
    </StackPanel>
    <StackPanel Orientation="Horizontal" Margin="5"
        HorizontalAlignment="Right">
      <Button x:Name="button1" Content="+" Width="35"
          Padding="5" Margin="5" />
      <Button x:Name="button2" Content="-" Width="35"
          Padding="5" Margin="5" />
```

```
      <Button x:Name="button3" Content="x" Width="35"
          Padding="5" Margin="5" />
      <Button x:Name="button4" Content="/" Width="35"
          Padding="5" Margin="5" />
      <Button x:Name="button5" Content="=" Width="75"
          Padding="5" Margin="5" />
    </StackPanel>
  </StackPanel>
</Window>
```

Для кнопки button1 создайте обработчик события Click (напомним, что для этого достаточно ввести в xaml-файле текст Click= и в появившемся выпадающем списке выбрать вариант «New Event Handler»:

```
<Button x:Name="button1" Content="+" Width="35"
    Padding="5" Margin="5" Click="button1_Click" />
```

Дополните созданный в cs-файле обработчик следующим образом:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    label1.Content = (e.Source as Button).Content;
}
```

После этого *переместите* текст Click="button1_Click" в открывающий тег *родителя* кнопки button1 (т. е. ближайшего к ней компонента StackPanel), дополнив имя Click префиксом Button:

```
<StackPanel Orientation="Horizontal" Margin="5"
    HorizontalAlignment="Right" Button.Click="button1_Click" >
  <Button x:Name="button1" Content="_+" Width="35"
      Padding="5" Margin="5" Click="button1_Click" />
```

Результат. Нажатие на *любую* кнопку приводит к отображению текста, указанного на этой кнопке, в метке label1 между полями ввода textBox1 и textBox2.

3.2. Организация вычислений

Определите обработчик события Click для кнопки button5:

```
<Button x:Name="button5" Content="=" Width="75"
    Padding="5" Margin="5" Click="button5_Click" />
```

```
private void button5_Click(object sender, RoutedEventArgs e)
{
    double x = 0, x1, x2;
    if (!double.TryParse(textBox1.Text, out x1) ||
      !double.TryParse(textBox2.Text, out x2))
    {
        label2.Content = "= ERROR";
```

```
        return;
    }
    switch (label1.Content as string)
    {
        case "+":
            x = x1 + x2; break;
        case "-":
            x = x1 - x2; break;
        case "x":
            x = x1 * x2; break;
        case "/":
            x = x1 / x2; break;
    }
    label2.Content = "= " + x;
}
```

Результат. При нажатии кнопки «=» указанное выражение вычисляется и отображается на экране (в метке label2). В качестве операнда при любой операции можно указывать число 0; при делении на 0 результатом является «–бесконечность» или «бесконечность» (в зависимости от знака первого операнда) или «NaN» («не число»), если первый операнд также равен 0. Если поля ввода содержат текст, который нельзя преобразовать в вещественное число, то выводится результат «ERROR».

Ошибка. Отмеченный в конце предыдущего пункта недочет теперь приводит к неправильной работе программы. После нажатия на кнопку «=» символ «=» указывается между полями ввода; таким образом, информация о выбранной операции стирается, и при последующем нажатии кнопки «=» всегда выводится нулевой результат (для восстановления нормальной работы надо повторно выбрать требуемую операцию, нажав на связанную с ней кнопку). Обратите внимание на то, что в данном варианте программы при наступлении события Click для кнопки «=» выполняются два обработчика: button5_Click, который связан непосредственно с этой кнопкой, и button1_Click, который связан с ее родительским компонентом StackPanel. Поскольку событие Click является пузырьковым, вначале выполняется обработчик button5_Click.

Исправление. В начало метода button5_Click добавьте оператор

```
e.Handled = true;
```

3.3. Простейшие приемы ускорения работы с помощью клавиатуры

```
<Window x:Class="CALC.MainWindow"
    … >
```

```
  …
    <Button x:Name="button1" Content="_+" … />
    <Button x:Name="button2" Content="_-" … />
    <Button x:Name="button3" Content="_x" … />
    <Button x:Name="button4" Content="_/" … />
    <Button x:Name="button5" … IsDefault="True" />
  …
</Window>
```

Обратите внимание на добавленные символы подчеркивания в свойствах Content.

Рис. 13. Окно приложения CALC с подчеркнутыми символами в подписях кнопок

Результат. Кнопка «=» (button5) сделана *кнопкой по умолчанию* и отображается в окне особым образом (см. рис. 13); эквивалентом ее нажатия является нажатие на клавишу Enter. Символы, указанные на кнопках, подчеркиваются; это является признаком того, что с каждой кнопкой связана *клавиша-ускоритель* Alt+«*подчеркнутый символ*». Следует иметь в виду, что в последних версиях Windows символы, с которыми связываются клавиши-ускорители, подчеркиваются только в случае, если предварительно нажать клавишу Alt.

Ошибка. После нажатия на любую кнопку с арифметической операцией все последующие вычисления возвращают значение, равное 0 (поскольку первым символом метки label1 теперь является символ подчеркивания '_', не предусмотренный в операторе switch). Кроме того, символ операции, изображенный между полями ввода теперь тоже подчеркивается.

Исправление. Измените оператор в методе button1_Click следующим образом:

```
label1.Content = ((e.Source as Button).Content
  as string).TrimStart('_');
```

Недочет. Теперь, когда наша программа содержит средства для быстрого выполнения действий с помощью клавиатуры, более наглядно проявляется недочет, который имелся в ней с самого начала: при запуске данной программы в ней отсутствует компонент, имеющий фокус. Для того чтобы фокус появился на первом поле ввода (и при этом в нем отобразился вертикальный курсор), необходимо либо щелкнуть мышью на этом поле, либо

нажать клавишу Tab. Было бы удобнее, если бы фокус устанавливался на первое поле ввода сразу после запуска программы.

Исправление. Добавьте в конструктор класса MainWindow оператор:

```
textBox1.Focus();
```

3.4. Использование обработчика событий от клавиатуры

Определите обработчик события PreviewTextInput для MainWindow:

```
<Window x:Class="CALC.MainWindow"
    … PreviewTextInput="Window_PreviewTextInput" >
```

```
private void Window_PreviewTextInput(object sender,
  TextCompositionEventArgs e)
{
    char c = e.Text[0];
    switch (c)
    {
        case '+':
            button1_Click(button1, null); break;
        case '_':
            button1_Click(button2, null); break;
        case 'x':
        case '*':
            button1_Click(button3, null); break;
        case '/':
            button1_Click(button4, null); break;
    }
    e.Handled = !(char.IsDigit(c) || c == '-' ||
      c == '\b' || c == ',');
}
```

Кроме того, измените метод button1_Click следующим образом:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    var s = sender as Button ?? e.Source as Button;
    label1.Content = (s.Content as string).TrimStart('_');
}
```

Результат. Теперь для ввода любой операции достаточно нажать соответствующую клавишу (поскольку клавиша «–» может использоваться для ввода отрицательных чисел, в качестве ускорителя для кнопки «–» выбрана комбинация Shift+«–», соответствующая символу подчеркивания «_»). При вводе чисел игнорируются все клавиши, кроме цифровых, «−», «,» и Backspace (для обозначения символа, генерируемого клавишей Backspace, в C# можно использовать управляющую последовательность

'\b'; нажатие этой клавиши обеспечивает удаление символа, расположенного *слева* от курсора в активном поле ввода).

Недочет 1. Если нажать клавишу пробела, находясь на одном из полей ввода, то пробел будет введен в это поле.

Это связано с тем, что пробел в WPF-приложениях обрабатывается особым образом: несмотря на то, что он является отображаемым символом и, казалось бы, нажатие на него должно приводить к возникновению события TextInput (и предшествующего ему события PreviewTextInput), этого не происходит. Таким образом, если мы хотим заблокировать ввод пробелов, это придется сделать с помощью дополнительного обработчика.

Исправление. Определите для компонента StackPanel, содержащего поля ввода, обработчик события PreviewKeyDown:

```
<StackPanel MinWidth="500">
  <StackPanel … PreviewKeyDown="StackPanel_PreviewKeyDown" >
    <TextBox x:Name="textBox1" MinWidth="120" Text="0"/>
```

```
private void StackPanel_PreviewKeyDown(object sender,
  KeyEventArgs e)
{
    e.Handled = e.Key == Key.Space;
}
```

Недочет 2. В нашей программе предполагается, что десятичным разделителем является *запятая*, тогда как при других региональных настройках в системе Windows может использоваться другой разделитель.

Исправление. Измените фрагмент последнего оператора в методе Window_PreviewTextInput:

```
c == ','
c == System.Globalization.CultureInfo.CurrentCulture.
  NumberFormat.NumberDecimalSeparator[0]
```

3.5. Контроль за изменением исходных данных

Добавьте в метод button1_Click следующий оператор:

```
label2.Content = "=";
```

Кроме того, определите для поля ввода textBox1 обработчик события TextChanged, а также свяжите этот обработчик с полем ввода textBox2:

```
<TextBox x:Name="textBox1" …
    TextChanged="textBox1_TextChanged" />
<Label … />
<TextBox x:Name="textBox2" …
    TextChanged="textBox1_TextChanged" />
```

```
private void textBox1_TextChanged(object sender,
  TextChangedEventArgs e)
```

```
{
    if (label2 != null)
        label2.Content = "=";
}
```

Результат. При изменении операции или содержимого текстовых полей результат предыдущего вычисления стирается. Это важная возможность, позволяющая предотвратить *рассогласование отображаемых данных*. При ее отсутствии возможна ситуация, когда после выполнения, например, вычислений вида 3 + 2 (с результатом 5) пользователь изменит первый операнд на 2, получив на экране текст 2 + 2 = 5.

4. Работа с датами и временем: CLOCK

Рис. 14. Окно приложения CLOCK

4.1. Отображение текущего времени

```
<Window x:Class="CLOCK.MainWindow"
    …
    Title="Clock" ResizeMode="CanMinimize"
    SizeToContent="WidthAndHeight"
    WindowStartupLocation="CenterScreen">
  <StackPanel>
    <Border Margin="10" BorderBrush="Black" BorderThickness="1" >
      <TextBlock x:Name="label1" Text="00:00:00" FontSize="100"
         Padding="50" FontFamily="Arial" TextAlignment="Center"/>
    </Border>
  </StackPanel>
</Window>
```

В список директив using в начале файла MainWindow.xaml.cs добавьте директиву:

```
using System.Windows.Threading;
```

В описание класса MainWindow добавьте поле

```
DispatcherTimer timer1 = new DispatcherTimer();
```

В конструктор класса добавьте следующие операторы:

```
timer1.Interval = TimeSpan.FromMilliseconds(1000);
timer1.Tick += timer1_Tick;
timer1.Start();
```

Опишите в классе MainWindow обработчик события Tick для таймера (этот обработчик придется ввести полностью, вместе с его заголовком, так как заготовку для него нельзя создать с помощью окна Properties или xaml-файла):

```
void timer1_Tick(object sender, EventArgs e)
{
    label1.Text = DateTime.Now.ToLongTimeString();
}
```

Результат. При работе программы в ее окне отображается текущее время (см. рис. 15).

Рис. 15. Окно приложения CLOCK (первый вариант)

Недочет. В течение первой секунды после запуска программы в метке сохраняется исходный текст «00:00:00», так как событие Tick возникает первый раз только через промежуток времени timer1.Interval, равный в нашем случае 1000 миллисекундам.

Исправление. Добавьте вызов обработчика для таймера в конструктор окна MainWindow:

```
timer1_Tick(null, null);
```

4.2. Реализация возможностей секундомера

```
<Window x:Class="CLOCK.MainWindow"
    … >
  <StackPanel>
    <Border Margin="10,10,10,0" … >
      …
    </Border>
    <StackPanel Margin="5" Orientation="Horizontal"
        HorizontalAlignment="Center">
      <CheckBox x:Name="checkBox1" Margin="5" Content="_Timer"
          VerticalContentAlignment="Center"
          VerticalAlignment="Center" Padding="10,0"
```

```
            Click="checkBox1_Click" />
        <Button x:Name="button1"  MinWidth="75" Padding ="5"
            Margin="5" Content="_Start/Stop" IsEnabled="False"
            Click="button1_Click" />
        <Button x:Name="button2"  MinWidth="75" Padding ="5"
            Margin="5" Content="_Reset" IsEnabled="False"
            Click="button2_Click" />
      </StackPanel>
    </StackPanel>
</Window>
```

В классе MainWindow определите обработчики, уже добавленные в него в результате указания атрибутов Click в xaml-файле:

```
private void checkBox1_Click(object sender, RoutedEventArgs e)
{
    if ((bool)checkBox1.IsChecked)
    {
        t = -1;
        timer1.Interval = TimeSpan.FromMilliseconds(100);
    }
    else
        timer1.Interval = TimeSpan.FromMilliseconds(1000);
    timer1_Tick(null, null);
    button1.IsEnabled = button2.IsEnabled =
      (bool)checkBox1.IsChecked;
    timer1.IsEnabled = true;
}
private void button1_Click(object sender, RoutedEventArgs e)
{
    timer1.IsEnabled = !timer1.IsEnabled;
}
private void button2_Click(object sender, RoutedEventArgs e)
{
    timer1.IsEnabled = false;
    t = 0;
    label1.Text = "0:0";
}
```

Кроме того, добавьте в класс MainWindow новое поле

```
int t;
```

а также дополните обработчик timer1_Tick:

```
void timer1_Tick(object sender, EventArgs e)
{
```

```
    if ((bool)checkBox1.IsChecked)
    {
        t++;
        label1.Text = string.Format("{0}:{1}",
          t / 10, t % 10);
    }
    else
        label1.Text = DateTime.Now.ToLongTimeString();
}
```

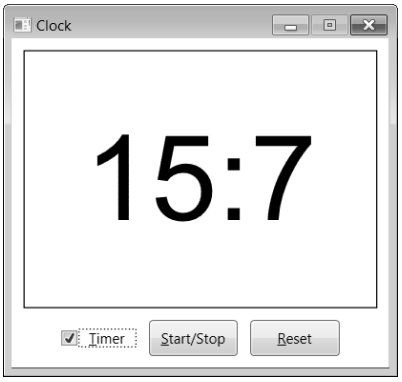

Рис. 16. Окно приложения CLOCK в режиме секундомера

Результат. При установке флажка Timer во включенное состояние программа переходит в *режим секундомера*, причем секундомер сразу запускается, отображая на экране секунды и десятые доли секунд (см. рис. 17). Запуск и остановка секундомера осуществляются по нажатию кнопки Start/Stop, сброс секундомера — по нажатию кнопки Reset. Доступны клавиши-ускорители: Alt+T (смена режима «часы/секундомер»), Alt+S (старт/остановка секундомера), Alt+R (сброс секундомера).

Недочет. При изменении режима изменяется ширина окна, «подстраиваясь» под текущий размер текста, выводимого на метке. Однако в данном случае изменение размеров окна не представляется оправданным. В частности, оно нарушит выравнивание окна по центру экрана. Кроме того, в режиме секундомера окно будет изменять размер во многих ситуациях, например, при переходе от 9 секунд к 10, от 99 секунд к 100, а также при сбросе значения секундомера.

Исправление. Добавьте к элементу Border в xaml-файле новый атрибут:

```
<Border … MinWidth="600">
```

Результат. Теперь ширина окна остается неизменной в любом режиме.

Ошибка. Кажущаяся правильность работы секундомера обманчива. В этом можно убедиться, если не останавливать секундомер в течение некоторого времени (выполняя при этом другие действия на компьютере), по-

сле чего сравнить результат с точным временем. Причина заключается в том, что событие Tick наступает *примерно* через каждые 100 мс; кроме того, надо учитывать, что данное событие наступает только при отсутствии других событий, которые требуется обработать программе. Если программа выполняет какой-либо обработчик длительное время, то в течение этого времени информация секундомера не будет обновляться, а затем отсчет времени продолжится с прежнего значения. Для правильной реализации секундомера надо связать его с *часами компьютера* (используя метод Now).

Исправление. В описании класса MainWindow удалите описание поля t и добавьте описание новых полей:

```
~~int t;~~
DateTime startTime, pauseTime;
TimeSpan pauseSpan;
```

Поле startTime будет содержать время начального запуска секундомера; поле pauseTime — время последней остановки секундомера, а поле pauseSpan — суммарную длительность всех остановок, выполненных после начального запуска.

Метод checkBox1_Click измените следующим образом (приведен только тот фрагмент метода, который требует изменения):

```
~~t = -1;~~
startTime = DateTime.Now;
pauseSpan = TimeSpan.Zero;
```

Аналогичные изменения внесите в метод button2_Click:

```
~~t = 0;~~
pauseTime = startTime;
pauseSpan = TimeSpan.Zero;
```

Добавьте в метод button1_Click операторы:

```
if (timer1.IsEnabled)
    pauseSpan += DateTime.Now - pauseTime;
else
    pauseTime = DateTime.Now;
```

И откорректируйте метод timer1_Tick:

```
private void timer1_Tick(object sender, EventArgs e)
{
    if ((bool)checkBox1.IsChecked)
    {
        ~~t++;~~
        ~~label1.Text = string.Format("{0}:{1}",~~
        ~~t / 10, t % 10);~~
        TimeSpan s = DateTime.Now - startTime - pauseSpan;
```

```
        label1.Text = string.Format("{0}:{1}",
          s.Minutes * 60 + s.Seconds, s.Milliseconds / 100);
    }
    else
      label1.Text = DateTime.Now.ToLongTimeString();
}
```

4.3. Альтернативные варианты выполнения команд с помощью мыши

```
<TextBlock x:Name="label1" … MouseDown="label1_MouseDown" />
```

```
private void label1_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    if (e.ClickCount == 1)
    {
        if (!button1.IsEnabled)
            return;
        if (e.ChangedButton == MouseButton.Left)
            button1_Click(null, null);
        else
            if (e.ChangedButton == MouseButton.Right)
                button2_Click(null, null);
    }
    else
    if (e.ClickCount == 2)
    {
        checkBox1.IsChecked = !(bool)checkBox1.IsChecked;
        checkBox1_Click(null, null);
    }
}
```

Результат. Двойной щелчок любой кнопкой мыши на метке приводит к смене режима «часы/секундомер», однократный щелчок левой кнопкой в режиме секундомера запускает или останавливает секундомер, однократный щелчок правой кнопкой мыши приводит к сбросу секундомера.

4.4. Отображение текущего состояния часов и секундомера на панели задач

В метод timer1_Tick добавьте следующий фрагмент:

```
Title = WindowState == WindowState.Minimized ?
  label1.Text : "Clock";
```

Результат. Если минимизировать окно приложения CLOCK, то на его кнопке, расположенной на панели задач (которая обычно размещается у нижней границы экрана), будет отображаться, в зависимости от режима, текущее время или данные секундомера. Если окно приложения находится в обычном состоянии, то на кнопке приложения отображается текст, совпадающий с заголовком окна: «Clock».

Недочет. Если минимизировать окно в режиме остановленного секундомера, то текст кнопки приложения не изменится.

Исправление. Определите обработчик события StateChanged для окна:

```
<Window x:Class="CLOCK.MainWindow"
    … StateChanged="Window_StateChanged" >
```

```
private void Window_StateChanged(object sender, EventArgs e)
{
    Title = WindowState == WindowState.Minimized ?
      label1.Text : "Clock";
}
```

Результат. Теперь текст на кнопке приложения правильно корректируется в любой ситуации; кроме того, корректировка этого текста выполняется быстрее.

5. Поля ввода: TEXTBOXES

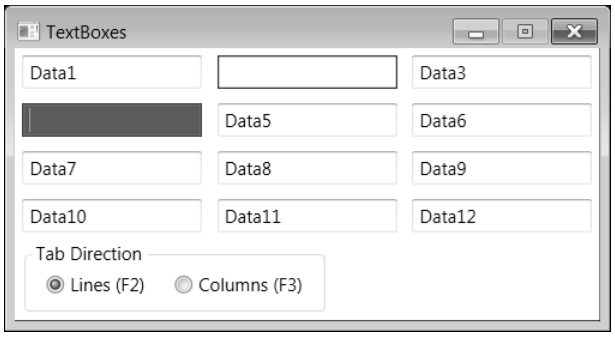

Рис. 17. Окно приложения TEXTBOXES

5.1. Дополнительное выделение активного поля ввода

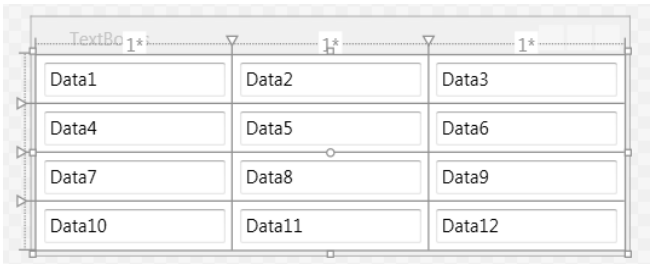

Рис. 18. Макет окна приложения TEXTBOXES (первый вариант)

При определении полей ввода в xaml-файле удобно вначале полностью задать первое поле, скопировать его в буфер обмена, а затем выполнять его вставку, корректируя имя поля, свойство Text и, при необходимости, значения свойств Grid.Row и Grid.Column (напомним, что в случае равенства 0 эти свойства можно не указывать).

```
<Window x:Class="TEXTBOXES.MainWindow"
    …
    Title="TextBoxes" SizeToContent="WidthAndHeight"
    WindowStartupLocation="CenterScreen"
    ResizeMode="CanMinimize" >
  <Grid x:Name="grid1" >
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
```

```
      <ColumnDefinition />
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <TextBox x:Name="textBox1" Grid.Row="0" Grid.Column="0"
        Margin="5" Text="Data1" MinWidth="120"/>
    <TextBox x:Name="textBox2" Grid.Row="0" Grid.Column="1"
        Margin="5" Text="Data2" MinWidth="120"/>
    <TextBox x:Name="textBox3" Grid.Row="0" Grid.Column="2"
        Margin="5" Text="Data3" MinWidth="120"/>
    <TextBox x:Name="textBox4" Grid.Row="1" Grid.Column="0"
        Margin="5" Text="Data4" MinWidth="120"/>
    <TextBox x:Name="textBox5" Grid.Row="1" Grid.Column="1"
        Margin="5" Text="Data5" MinWidth="120"/>
    <TextBox x:Name="textBox6" Grid.Row="1" Grid.Column="2"
        Margin="5" Text="Data6" MinWidth="120"/>
    <TextBox x:Name="textBox7" Grid.Row="2" Grid.Column="0"
        Margin="5" Text="Data7" MinWidth="120"/>
    <TextBox x:Name="textBox8" Grid.Row="2" Grid.Column="1"
        Margin="5" Text="Data8" MinWidth="120"/>
    <TextBox x:Name="textBox9" Grid.Row="2" Grid.Column="2"
        Margin="5" Text="Data9" MinWidth="120"/>
    <TextBox x:Name="textBox10" Grid.Row="3" Grid.Column="0"
        Margin="5" Text="Data10" MinWidth="120"/>
    <TextBox x:Name="textBox11" Grid.Row="3" Grid.Column="1"
        Margin="5" Text="Data11" MinWidth="120"/>
    <TextBox x:Name="textBox12" Grid.Row="3" Grid.Column="2"
        Margin="5" Text="Data12" MinWidth="120"/>
  </Grid>
</Window>
```

Для поля ввода textBox1 создайте обработчики событий GotFocus и LostFocus:

```
    <TextBox x:Name="textBox1" … GotFocus="textBox1_GotFocus"
        LostFocus="textBox1_LostFocus" />
```

После этого *переместите* полученные атрибуты в компонент Grid:

```
  <Grid x:Name="grid1" GotFocus="textBox1_GotFocus"
      LostFocus="textBox1_LostFocus" >
    …
    <TextBox x:Name="textBox1" … GotFocus="textBox1_GotFocus"
        LostFocus="textBox1_LostFocus" />
```

В файле MainWindow.xaml.cs в описание класса MainWindow добавьте поля

```
Brush backgr;
Brush foregr;
```

В конструктор класса MainWindow добавьте следующие операторы:

```
backgr = textBox1.Background;
foregr = textBox1.Foreground;
textBox1.Focus();
```

Наконец, определите в классе MainWindow ранее созданные обработчики:

```
private void textBox1_GotFocus(object sender, RoutedEventArgs e)
{
    TextBox tb = e.Source as TextBox;
    tb.Foreground = Brushes.White;
    tb.Background = Brushes.Green;
}
private void textBox1_LostFocus(object sender, RoutedEventArgs e)
{
    TextBox tb = e.Source as TextBox;
    tb.Foreground = foregr;
    tb.Background = backgr;
}
```

Результат. При получении фокуса любым полем ввода (т. е. при *активизации* поля ввода) изменяется его фон и цвет символов; при потере фокуса восстанавливается исходная цветовая настройка.

Недочет. При получении фокуса вертикальный курсор (*каретка*, caret), имеющий вид вертикальной линии, появляется в начале текста, тогда как удобнее, чтобы он располагался в его конце.

Исправление. Добавьте в метод textBox1_GotFocus оператор:

```
tb.Select(tb.Text.Length, 0);
```

Результат. Теперь при получении фокуса клавиатурный курсор располагается за последним символом текста.

5.2. Управление порядком обхода полей на форме

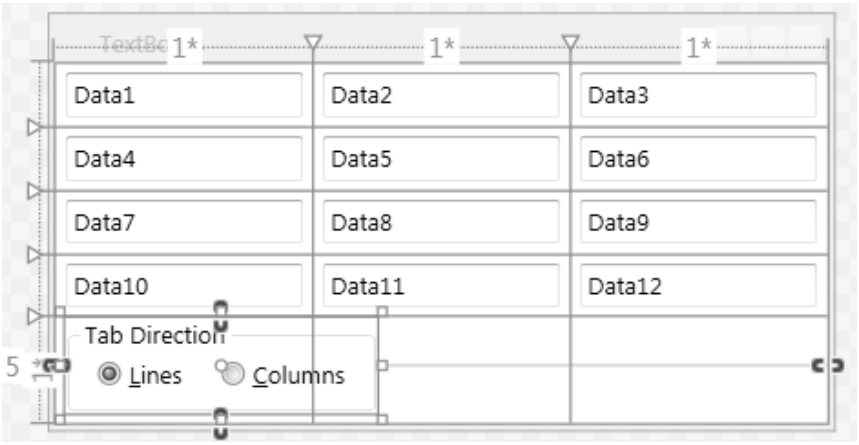

Рис. 19. Макет окна приложения TEXTBOXES (второй вариант)

```
<Window x:Class="TEXTBOXES.MainWindow"
        … >
  <Grid x:Name="grid1" GotFocus="textBox1_GotFocus"
      LostFocus="textBox1_LostFocus" >
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    …
    <TextBox x:Name="textBox12" Grid.Row="3" Grid.Column="2"
         Margin="5" Text="Data12" MinWidth="120"/>
    <GroupBox Grid.ColumnSpan="3" Grid.Row="4" Grid.Column="0"
        Header="Tab Direction" HorizontalAlignment="Left"
        Margin="5,0,5,5" VerticalAlignment="Center">
      <StackPanel Orientation="Horizontal">
        <RadioButton x:Name="radioButton1" Content="_Lines"
            Margin="10,5" IsChecked="True"
            Checked="radioButton1_Checked" />
        <RadioButton x:Name="radioButton2" Content="_Columns"
            Margin="10,5" Checked="radioButton2_Checked" />
      </StackPanel>
    </GroupBox>
  </Grid>
</Window>
```

```
private void radioButton1_Checked(object sender,
```

```
  RoutedEventArgs e)
{
    foreach (var e1 in grid1.Children)
    {
        TextBox tb = e1 as TextBox;
        if (tb != null)
            tb.TabIndex = int.MaxValue;
    }
}
private void radioButton2_Checked(object sender,
  RoutedEventArgs e)
{
    for (int i = 0; i < grid1.Children.Count - 1; i++)
    {
        TextBox tb = grid1.Children[i] as TextBox;
        tb.TabIndex = i * (int)Math.Pow(10, i % 3);
    }
}
```

Результат. Добавленные в окно радиокнопки предназначены для изменения порядка обхода полей ввода (теперь возможны два варианта обхода с помощью клавиш Tab и Shift+Tab: по строкам и по столбцам). Для переключения порядка обхода можно также использовать клавиатурные комбинации Alt+L и Alt+C.

Ошибка. При попытке выбрать радиокнопку (любым способом — с помощью щелчка мыши или путем нажатия клавиатурной комбинации) в программе возникает исключение. Это связано с тем, что для радиокнопок, как и для полей ввода, предусмотрены события GotFocus и LostFocus, при возникновении которых компонент Grid запускает обработчики textBox1_GotFocus и textBox1_LostFocus (эти обработчики запускаются для *любых* его дочерних компонентов, если для них предусмотрены данные события). При вызове обработчиков для радиокнопок свойство e.Source будет иметь тип RadioButton, который *нельзя* привести к типу TextBox. В такой ситуации операция as возвращает значение null, которое записывается в переменную tb, и при попытке обращения к любому свойству для «пустой» переменной tb возбуждается исключение NullReferenceException.

Исправление. Дополните методы textBox1_GotFocus и textBox1_LostFocus:

```
private void textBox1_GotFocus(object sender, RoutedEventArgs e)
{
    TextBox tb = e.Source as TextBox;
```

```
    if (tb == null)
        return;
    tb.Foreground = Brushes.White;
    tb.Background = Brushes.Green;
    tb.Select(tb.Text.Length, 0);
}
private void textBox1_LostFocus(object sender, RoutedEventArgs e)
{
    TextBox tb = e.Source as TextBox;
    if (tb == null)
        return;
    tb.Foreground = foregr;
    tb.Background = backgr;
}
```

Недочет. При любом из реализованных способов изменения порядка обхода полей текущее поле ввода теряет фокус (поскольку фокус принимает одна из радиокнопок).

Исправление. Определите для компонента grid1 обработчик события PreviewKeyDown и дополните текст радиокнопок:

```
<Grid x:Name="grid1" GotFocus="textBox1_GotFocus"
      LostFocus="textBox1_LostFocus"
      PreviewKeyDown="grid1 PreviewKeyDown" >
  …
      <RadioButton x:Name="radioButton1" Content="_Lines (F2)"
          Margin="10,5" IsChecked="True"
          Checked="radioButton1_Checked" />
      <RadioButton x:Name="radioButton2" Content="_Columns (F3)"
          Margin="10,5" Checked="radioButton2_Checked" />
```

```
private void grid1_PreviewKeyDown(object sender, KeyEventArgs e)
{
    switch (e.Key)
    {
        case Key.F2:
            radioButton1.IsChecked = true;
            break;
        case Key.F3:
            radioButton2.IsChecked = true;
            break;
    }
}
```

Результат. Теперь для настройки порядка обхода полей по строкам достаточно нажать клавишу F2, а по столбцам — F3, причем текущее поле ввода сохраняет фокус.

5.3. Проверка правильности введенных данных

В файле MainWindow.xaml.cs в описание класса MainWindow добавьте поля

```
Brush bordbr;
Thickness bordth;
```

Для поля ввода textBox1 создайте обработчик события TextChanged:

```
<TextBox x:Name="textBox1" …
    TextChanged="textBox1_TextChanged" />
```

```
private void textBox1_TextChanged(object sender,
  TextChangedEventArgs e)
{
    TextBox tb = e.Source as TextBox;
    if (tb.Text == "")
    {
        tb.BorderBrush = Brushes.Red;
        tb.BorderThickness = new Thickness(1.01);
    }
    else
    {
        tb.BorderBrush = bordbr;
        tb.BorderThickness = bordth;
    }
}
```

После создания обработчика textBox1_TextChanged *удалите* связанный с ним атрибут в xaml-файле:

```
<TextBox x:Name="textBox1" …
    TextChanged="textBox1_TextChanged" />
```

Наконец, дополните конструктор класса MainWindow следующим образом:

```
public MainWindow()
{
    InitializeComponent();
    grid1.AddHandler(TextBox.TextChangedEvent,
      new TextChangedEventHandler(textBox1_TextChanged));
    backgr = textBox1.Background;
    foregr = textBox1.Foreground;
    bordbr = textBox1.BorderBrush;
```

```
    bordth = textBox1.BorderThickness;
    textBox1.Focus();
}
```

Результат. Если активное поле ввода является пустым, то вокруг него рисуется красная рамка, которая сохраняется и при потере фокуса этим полем. Такой способ позволяет наглядно информировать пользователя о том, что введенное им значение является недопустимым.

Недочет. Причина, по которой пустое поле выделяется как ошибочное, может быть непонятна пользователю.

Исправление. Дополните текст метода textBox1_TextChanged:

```
private void textBox1_TextChanged(object sender,
  TextChangedEventArgs e)
{
    TextBox tb = e.Source as TextBox;
    if (tb.Text == "")
    {
        tb.BorderBrush = Brushes.Red;
        tb.BorderThickness = new Thickness(1.01);
        tb.ToolTip = "Поле не должно быть пустым";
    }
    else
    {
        tb.BorderBrush = bordbr;
        tb.BorderThickness = bordth;
        tb.ToolTip = null;
    }
}
```

Результат. Теперь при наведении курсора мыши на поле, обведенное красной рамкой (не обязательно активное), возникает *всплывающая подсказка* (tool tip) «Поле не должно быть пустым». Для заполненных полей подсказка не отображается.

5.4. Блокировка окна с ошибочными данными

Как правило, программы, в которые введены ошибочные данные, не запрещают пользователю перемещаться по различным полям ввода, но при этом блокируют выполнение действий, связанных с окончательной обработкой всего введенного набора данных (например, сохранение данных в файле или их пересылка по сети). Реализуем аналогичное поведение для нашего проекта.

```
<Window x:Class="TEXTBOXES.MainWindow"
    … Closing="Window Closing" >
  …
```

```
</Window>
```

```
private void Window_Closing(object sender,
  System.ComponentModel.CancelEventArgs e)
{
    bool res = false;
    foreach (var e1 in grid1.Children)
        if ((e1 as Control).BorderBrush == Brushes.Red)
        {
            res = true;
            break;
        }
    e.Cancel = res;
}
```

Результат. Наличие пустого поля ввода не препятствует переходу в другие поля, однако пустое поле помечается как ошибочное. При наличии хотя бы одного ошибочного поля окно нельзя закрыть.

6. Обработка событий от мыши: MOUSE

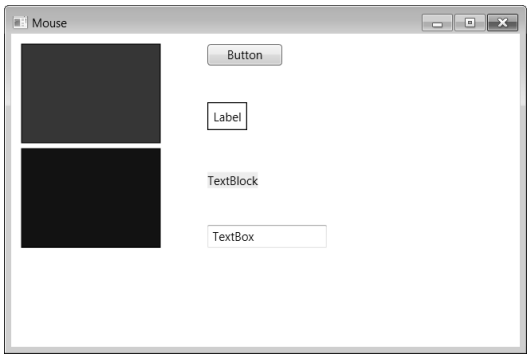

Рис. 20. Окно приложения MOUSE

6.1. Перетаскивание панели с помощью мыши

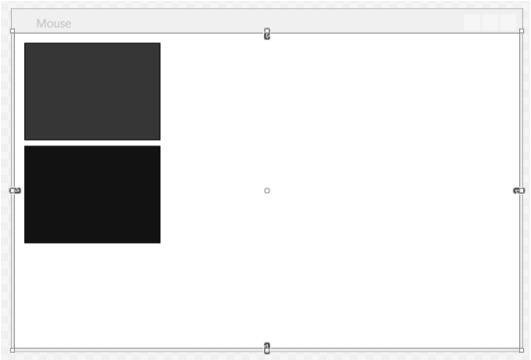

Рис. 21. Макет окна приложения MOUSE (первый вариант)

```
<Window x:Class="MOUSE.MainWindow"
    …
    Title="Mouse" Height="350" Width="525"
    WindowStartupLocation="CenterScreen" >
  <Canvas x:Name="canvas1" Background="White" >
    <Rectangle x:Name="rect1" Fill="Red" Height="100"
        Canvas.Left="10" Stroke="Black" Canvas.Top="10"
        Width="140" />
    <Rectangle x:Name="rect2" Fill="Blue" Height="100"
        Canvas.Left="10" Stroke="Black" Canvas.Top="115"
        Width="140" />
  </Canvas>
</Window>
```

В описание класса MainWindow добавьте поле

```
Point p;
```

Для компонента rect1 создайте обработчики событий MouseDown и MouseMove:

```
<Rectangle x:Name="rect1" … MouseDown="rect1_MouseDown"
    MouseMove="rect1_MouseMove" />
```

```
private void rect1_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    if (e.Source == canvas1)
        return;
    var a = e.Source as Rectangle;
    p = e.GetPosition(a);
}
private void rect1_MouseMove(object sender, MouseEventArgs e)
{
    if (e.Source == canvas1)
    {
        Title = "Mouse";
        return;
    }
    var a = e.Source as Rectangle;
    Point q = e.GetPosition(a);
    Title = string.Format("Mouse - {0} {1}", a.Name, q);
    if (e.LeftButton == MouseButtonState.Pressed)
    {
        Canvas.SetLeft(a, Canvas.GetLeft(a) + q.X - p.X);
        Canvas.SetTop(a, Canvas.GetTop(a) + q.Y - p.Y);
    }
}
```

После создания обработчиков *переместите* связанные с ними атрибуты MouseDown="rect1_MouseDown" и MouseMove ="rect1_MouseMove" в компонент Canvas:

```
<Canvas x:Name="canvas1" … MouseDown="rect1_MouseDown"
    MouseMove="rect1_MouseMove" >
  <Rectangle x:Name="rect1" … MouseDown="rect1_MouseDown"
      MouseMove="rect1_MouseMove" />
```

Результат. При перемещении курсора мыши над любым прямоугольником в заголовке окна выводится имя прямоугольника и текущие значения *локальных* координат мыши (относительно прямоугольника). Если при этом удерживается нажатой левая кнопка мыши, то прямоугольник пере-

мещается по окну (*перетаскивается* мышью). При перемещении курсора мыши на свободную часть окна восстанавливается исходный заголовок «Mouse».

Недочет. Прямоугольник rect1 при перетаскивании может «заслоняться» прямоугольником rect2. При этом если курсор мыши окажется над прямоугольником rect2, то курсор будет «захвачен» прямоугольником rect2 (поскольку этот прямоугольник располагается сверху и поэтому перехватывает события от мыши). Было бы удобнее всегда делать «верхним» тот компонент, который перетаскивается в данный момент.

Исправление. Добавьте в класс MainWindow новое поле

```
int maxz;
```

В метод rect1_MouseDown добавьте оператор:

```
Canvas.SetZIndex(a, ++maxz);
```

6.2. Изменение размеров компонента с помощью мыши. Захват мыши и его особенности

В описание класса MainWindow добавьте поле

```
Size s;
```

В метод rect1_MouseDown добавьте оператор

```
s = new Size(a.ActualWidth, a.ActualHeight);
```

В метод rect1_MouseMove добавьте фрагмент

```
else
if (e.RightButton == MouseButtonState.Pressed)
{
    a.Width = s.Width + q.X - p.X;
    a.Height = s.Height + q.Y - p.Y;
}
```

Результат. При перемещении мыши над любым прямоугольником с нажатой *правой* кнопкой происходит изменение *размеров* прямоугольника (левая кнопка по-прежнему используется для изменения положения прямоугольника в окне).

Недочет. Для того чтобы получить прямоугольник малого размера, необходимо «зацепить» его правой кнопкой мыши около правого нижнего угла, поскольку изменение размеров прекращается, как только курсор мыши покинет область прямоугольника.

Отмеченный недочет можно исправить, если использовать возможность *захвата мыши* (mouse capture). Следует заметить, что в библиотеке Windows Forms захват мыши выполняется автоматически, тогда как в библиотеке WPF им надо управлять явным образом.

Явление захвата состоит в том, что если нажать над компонентом какую-либо кнопку мыши, то этот компонент «захватит» мышь и «заставит» передавать ему все сообщения от мыши (даже если курсор мыши покинет

компонент) до тех пор, пока кнопка мыши не будет отпущена (причем это событие тоже будет обработано компонентом, ранее захватившим мышь). Используя захват мыши, мы получаем более полный контроль над действиями, связанными с перетаскиванием компонента и изменением его размеров, однако при этом могут возникнуть особые ситуации, для которых потребуется предусматривать специальную обработку.

Исправление. Добавьте новый оператор *в начало* метода rect1_MouseDown:

```
private void rect1_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    (e.Source as FrameworkElement).CaptureMouse();
    if (e.Source == canvas1)
        return;
    var a = e.Source as Rectangle;
    p = e.GetPosition(a);
    Canvas.SetZIndex(a, ++maxz);
    s = new Size(a.ActualWidth, a.ActualHeight);
}
```

Для компонента rect1 создайте обработчик события MouseUp, после чего *переместите* связанный с ним атрибут в компонент Canvas:

```
<Canvas x:Name="canvas1" … MouseUp="rect1_MouseUp" >
  <Rectangle x:Name="rect1" … MouseUp="rect1_MouseUp" />
```

```
private void rect1_MouseUp(object sender, MouseButtonEventArgs e)
{
    (e.Source as FrameworkElement).ReleaseMouseCapture();
}
```

Результат. Теперь прямоугольник можно перетащить даже на область вне окна приложения: если при этом не отпускать кнопку мыши, то прямоугольник можно вернуть обратно на видимую часть окна. Аналогично, уменьшив размеры прямоугольника практически до нулевых (так, что курсор мыши покинет область прямоугольника, а одна или обе координаты, отображаемые в заголовке окна, станут отрицательными), можно затем восстановить их.

Эффект захвата может проявиться в нашей программе и по-другому: если нажать кнопку мыши на свободной части окна (т. е. на компоненте Canvas), а затем переместить курсор мыши на один из прямоугольников, то заголовок окна не изменится (он по-прежнему будет иметь вид «Mouse»). Это происходит потому, что в данной ситуации мышь захвачена *компонентом Canvas* (поскольку для него тоже вызывается обработчик rect1_MouseDown), и при перемещении мыши над прямоугольником собы-

тие MouseMove передается не этому прямоугольнику, а захватившему мышь компоненту Canvas. Стоит в этой ситуации отпустить мышь, находясь на прямоугольнике, как любое ее перемещение будет перехвачено этим прямоугольником, что проявится в изменении заголовка окна.

Ошибка. Если попытаться сделать размеры прямоугольника меньше нулевых, то возникнет исключение, связанное с тем, что свойства Width и Height *не могут принимать отрицательные значения*. Кроме того, если перетащить прямоугольник целиком за левую или верхнюю границу окна и *отпустить кнопку мыши*, то доступ к прямоугольнику окажется невозможным.

Исправление. Измените завершающую часть метода rect1_MouseMove следующим образом:

```
if (e.LeftButton == MouseButtonState.Pressed)
{
    Canvas.SetLeft(a, Math.Max(0, Canvas.GetLeft(a) + q.X - p.X));
    Canvas.SetTop(a, Math.Max(0, Canvas.GetTop(a) + q.Y - p.Y));
}
else
if (e.RightButton == MouseButtonState.Pressed)
{
    a.Width = Math.Max(20, s.Width + q.X - p.X);
    a.Height = Math.Max(20, s.Height + q.Y - p.Y);
}
```

Результат. Теперь перетаскивать прямоугольник за пределы левой или верхней границы окна невозможно, и, кроме того, определен *минимальный* размер прямоугольника, равный 20 × 20 (напомним, что размеры в WPF задаются в *аппаратно-независимых единицах*, равных 1/96 дюйма).

6.3. Использование дополнительных курсоров

Добавьте в метод rect1_MouseDown следующие операторы:

```
if (e.ChangedButton == MouseButton.Left)
    a.Cursor = Cursors.Hand;
else
if (e.ChangedButton == MouseButton.Right)
    a.Cursor = Cursors.SizeNWSE;
```

Добавьте в метод rect1_MouseUp оператор:

```
(e.Source as FrameworkElement).Cursor = null;
```

Результат. В режиме изменения размеров курсор мыши принимает вид диагональной двунаправленной стрелки, а в режиме перетаскивания — «указывающей руки». После выхода из этих режимов восстанавливается стандартный вид курсора.

6.4. Обработка ситуации с одновременным нажатием двух кнопок мыши

Наша программа будет прекрасно работать до тех пор, пока пользователю не придет в голову мысль нажать левую кнопку мыши *при нажатой правой* (или наоборот). Для определенности опишем поведение программы в ситуации, когда при нажатой на прямоугольнике левой кнопке мыши была дополнительно нажата правая: в момент нажатия второй кнопки вид курсора изменится на диагональную стрелку, однако при последующем перемещении мыши по-прежнему будет выполняться перемещение окна (а не изменение его размеров). Если в этой ситуации отпустить левую кнопку мыши (оставив нажатой правую), то курсор примет стандартный вид, но при перемещении мыши будут *изменяться размеры* прямоугольника, причем при выходе курсора мыши за границу прямоугольника режим перемещения будет отменен (так как при ранее выполненном отпускании левой кнопки был отменен режим захвата мыши).

Чтобы избежать подобных нежелательных эффектов, следует более детально анализировать состояние кнопок мыши в обработчиках, связанных с их нажатием и отпусканием. При этом следует учитывать, что в обработчике события MouseMove мы *вначале* анализируем левую кнопку мыши, а *затем* правую. Таким образом, естественно считать, что левая кнопка *имеет более высокий приоритет*: если она нажата (даже одновременно с правой), то должно выполняться *перемещение* прямоугольника, а не изменение его размеров (и вид курсора должен учитывать эту особенность).

Измените методы rect1_MouseDown и rect1_MouseUp:

```
private void rect1_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    (e.Source as UIElement).CaptureMouse();
    if (e.Source == canvas1)
        return;
    var a = e.Source as Rectangle;
    p = e.GetPosition(a);
    Canvas.SetZIndex(a, ++maxz);
    s = new Size(a.ActualWidth, a.ActualHeight);
    if (e.LeftButton == MouseButtonState.Pressed)
        a.Cursor = Cursors.Hand;
    else
    if (e.RightButton == MouseButtonState.Pressed)
        a.Cursor = Cursors.SizeNWSE;
}
```

```
private void rect1_MouseUp(object sender, MouseButtonEventArgs e)
{
    var a = e.Source as FrameworkElement;
    if (e.LeftButton == MouseButtonState.Pressed)
    {
        if (a != canvas1)
            a.Cursor = Cursors.Hand;
    }
    else
    if (e.RightButton == MouseButtonState.Pressed)
    {
        if (a != canvas1)
            a.Cursor = Cursors.SizeNWSE;
    }
    else
    {
        (e.Source as FrameworkElement).ReleaseMouseCapture();
        (e.Source as FrameworkElement).Cursor = null;
    }
}
```

Результат. Внесенные исправления полностью исключают описанное выше аномальное поведение. При нажатии и отпускании любых кнопок мыши в любом порядке на компоненте Canvas курсор не изменяется. При нажатии и отпускании любых кнопок мыши в любом порядке на прямоугольниках вид курсора всегда соответствует правильному режиму: это режим перетаскивания, если нажата левая кнопка (даже если одновременно нажата правая), или режим изменения размера, если нажата правая кнопка, а левая отпущена. В любом случае захват мыши отменяется только при отпускании *всех* ее кнопок. Отметим также, что наша программа корректно реагирует и на действия со *средней* кнопкой мыши (т. е. с колесиком, которое также можно нажимать): нажатие и отпускание этой кнопки никак не влияет на текущий режим программы.

6.5. Перетаскивание компонентов любого типа

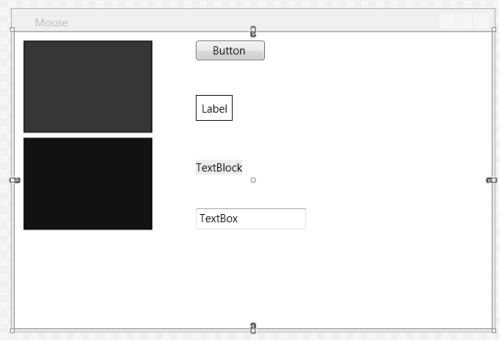

Рис. 22. Макет окна приложения MOUSE (второй вариант)

Добавьте к компоненту Canvas новые компоненты различного типа:

```
<Canvas x:Name="canvas1" … >
  …
  <Button x:Name="button1" Content="Button" Canvas.Left="197"
      Canvas.Top="10" Width="75"/>
  <Label x:Name="label1" BorderBrush="Black" BorderThickness="1"
      Content="Label" Canvas.Left="197" Canvas.Top="69"/>
  <TextBlock x:Name="textBlock1" Background="Yellow"
      Canvas.Left="197" Text="TextBlock" Canvas.Top="139"/>
  <TextBox x:Name="textBox1" Height="23" Canvas.Left="197"
      Text="TextBox" Canvas.Top="192" Width="120"/>
</Canvas>
```

В каждый из методов rect1_MouseDown и rect1_MouseMove внесите следующее изменение:

```
var a = e.Source as Rectangle;
var a = e.Source as FrameworkElement;
```

Результат. Добавленные компоненты (кнопка, два вида меток и поле ввода) обрабатываются *почти* так же, как и прямоугольники: их размер и положение можно изменять с помощью мыши (особенности, связанные с кнопкой и полем ввода, будут рассмотрены далее).

Недочет. Кнопку и поле ввода нельзя перемещать при нажатой левой кнопке мыши, подобно другим компонентам. Это связано с тем, что для данных компонентов нажатие левой кнопки мыши обрабатывается особым образом, а действия, определенные в обработчиках событий MouseDown, MouseMove и MouseUp, игнорируются. Тем не менее, для этих компонентов тоже можно обеспечить выполнение требуемых действий, если вместо событий MouseDown, MouseMove и MouseUp обрабатывать *предшествующие им* события PreviewMouseDown, PreviewMouseMove и PreviewMouseUp.

Исправление. Измените три атрибута для компонента Canvas в xaml-файле (к имени каждого атрибута надо добавить префикс Preview):

```
<Canvas x:Name="canvas1" Background="White"
    PreviewMouseDown="rect1_MouseDown"
    PreviewMouseMove="rect1_MouseMove"
    PreviewMouseUp="rect1_MouseUp">
```

Результат. Теперь режим перемещения и изменения размеров работает одинаково для всех компонентов окна, причем для кнопки и поля ввода это не препятствует выполнению *дополнительных действий*, определенных для данных компонентов. Например, при нажатии на кнопку Button левой кнопкой мыши она переходит в «нажатое» состояние, а если пере-

местить поле ввода TextBox к левой границе окна и продолжить перемещать курсор мыши влево так, что курсор будет перемещаться по тексту, содержащемуся в поле ввода, то произойдет *выделение* части текста.

Указанные дополнительные действия определены в обработчиках для событий мыши, которые «встроены» в компоненты Button и TextBox. Несмотря на то что эти обработчики встроены в код компонентов, *их можно отключить*; для этого достаточно добавить в каждый из методов rect1_MouseDown и rect1_MouseMove следующий оператор (метод rect1_MouseUp изменять не требуется):

```
e.Handled = true;
```

Теперь при нажатии левой кнопкой мыши на кнопке Button она не будет переходить в нажатое состояние.