МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

А. В. Абрамян, М. Э. Абрамян

Разработка пользовательского интерфейса на основе технологии Windows Presentation Foundation

Учебник

по курсу «Основы разработки пользовательского интерфейса» для студентов направления 02.03.02 «Фундаментальная информатика и информационные технологии» (бакалавриат)

Ростов-на-Дону – Таганрог
Издательство Южного федерального университета
2017

УДК [004.4:004.51](075.8)
ББК 32.973я73
А164

Печатается по решению учебно-методической комиссии Института математики, механики и компьютерных наук им. И. И. Воровича Южного федерального университета (протокол № 4 от 14 апреля 2017 г.)

Рецензенты:
профессор кафедры «Информатика» Ростовского государственного университета путей сообщения (РГУПС), доктор технических наук М. А. Бутакова;

доцент кафедры алгебры и дискретной математики Института математики, механики и компьютерных наук им. И. И. Воровича Южного федерального университета, кандидат физико-математических наук С. С. Михалкович

Абрамян А. В.

А 164 Разработка пользовательского интерфейса на основе системы Windows Presentation Foundation : учебник / А. В. Абрамян, М. Э. Абрамян ; Южный федеральный университет. – Ростов-на-Дону ; Таганрог : Издательство Южного федерального университета. 2017. – 301 с.
ISBN 978-5-9275-2375-7

В учебнике рассмотрены основные приемы разработки пользовательского интерфейса на основе технологии Windows Presentation Foundation (WPF), входящей в состав платформы .NET, начиная с версии 3.0. Учебный материал излагается в форме подробного описания 19 проектов для среды программирования Microsoft Visual Studio 2015, демонстрирующих различные аспекты технологии WPF. Описание проектов сопровождается многочисленными комментариями. Завершающий раздел содержит 48 учебных заданий, предназначенных для закрепления изученного материала.

Для студентов бакалавриата, обучающихся по направлению подготовки 02.03.02 «Фундаментальная информатика и информационные технологии».

УДК [004.4:004.51](075.8)
ББК 32.973я73

ISBN 978-5-9275-2375-7

Оглавление

Предисловие

Книга знакомит читателя с основными приемами разработки пользовательского интерфейса на основе системы Windows Presentation Foundation (WPF), входящей в платформу .NET, начиная с версии 3.0. Данная система была создана с целью снять ряд серьезных ограничений, имевшихся у ее предшественницы – системы разработки интерфейса Windows Forms, изначально входившей в состав платформы .NET. При этом наряду с сохранением концепций, лежащих в основе Windows Forms (в частности, механизма обработки событий), система WPF была дополнена новыми технологиями, позволяющими разрабатывать интерфейсы, имеющие существенно более широкие графические возможности и, что не менее важно в современном мире планшетов и смартфонов, допускающими автоматическую адаптацию к особенностям устройств, на которых запущено приложение.

Новой важной чертой системы WPF для разработчиков приложений стало явное разграничение программного кода и макета приложения, которое было обеспечено включением в состав проекта XML-файлов, позволяющих описывать визуальный интерфейс на особом декларативном языке разметки XAML (eXtensible Application Markup Language). Следует отметить, что в настоящее время подобный подход реализован в большинстве систем разработки интерфейсов.

Разработчики приложений на основе технологии WPF получили в свое распоряжение новые концепции, основанные на свойствах зависимостей, маршрутизируемых событиях и привязке данных. Созданные в рамках WPF иерархии классов и визуальных компонентов, в том числе компонентов с содержимым и группирующих компонентов, обеспечивающих динамическую компоновку своих дочерних элементов, позволили гибко комбинировать интерфейсные элементы, обеспечивая подходящее визуальное представление на экране любого размера.

Принципиально новой по сравнению с Windows Forms стала графическая подсистема, основанная на технологии DirectX и позволяющая реализовывать качественную и быструю двумерную и трехмерную графику и анимацию. Был осуществлен переход на аппаратно-независимую систему единиц измерения (вместо экранных пикселов), обеспечивший одинаковый внешний вид приложения на экране с любым разрешением.

Не последнее место в списке преимуществ технологии WPF занимает удобство разработки WPF-приложений, которое обеспечивается средства-

ми среды программирования Microsoft Visual Studio (версии 2008–2015). Традиционные методы визуальной разработки интерфейса, имевшиеся уже в технологии Windows Forms, были дополнены методами, основанными на непосредственном редактировании xaml-файлов, определяющих макет графического приложения. Подобное редактирование (как и редактирование программного кода) существенно упрощается благодаря контекстным подсказкам, автоматической проверке синтаксиса, средствам автозавершения и другим средствам, встроенным в редакторы кода и xaml-файлов.

Учитывая перечисленные выше особенности системы WPF и принимая во внимание широкую распространенность технологий и языков программирования, основанных на платформе .NET, представляется вполне оправданным применение данной системы в качестве базовой при изучении приемов и методов разработки пользовательского интерфейса в рамках соответствующего университетского курса.

При этом, однако, возникают две проблемы. Первая обусловлена сложностью системы WPF и обширностью ее средств, которые невозможно охватить в рамках одного курса. Вторая проблема связана с недостатком учебной литературы. Если ограничиться изданиями, переведенными на русский язык, то можно отметить лишь [7–9] (при включении в рассмотрение английских книг их общее количество увеличится лишь в 2–3 раза). Из этих изданий на роль учебника может отчасти претендовать только книга Чарльза Петцольда [7], написанная с большим методическим мастерством (что характерно для всех работ этого автора) и содержащая много примеров законченных программ. Две другие книги являются, скорее, справочниками, включающими небольшие фрагменты иллюстративного кода.

Предлагаемая книга представляет собой попытку решения обеих проблем. С одной стороны, она посвящена лишь основным возможностям технологии WPF, которые вполне можно освоить за семестровый курс, а с другой – излагает материал в «практической» форме, упрощающей его усвоение. Изложение построено в виде подробного описания ряда законченных программных проектов, каждый из которых посвящен одному из аспектов технологии WPF. Многочисленные комментарии содержат дополнительные сведения, связанные с изучаемыми возможностями. Во многих случаях приводится сравнение рассматриваемых средств WPF с аналогичными средствами библиотеки Windows Forms. В процессе разработки проектов авторы намеренно допускают типичные ошибки, характерные для начинающих разработчиков интерфейсов, подробно объясняют их причины и приводят способы исправления. Подобный вариант изложения учебного материала «на примерах» ранее с успехом применялся авторами при чтении ими курсов по разработке пользовательских интерфейсов с использованием библиотеки VCL системы программирования Borland Delphi

[1, 2] и предшественницы WPF – библиотеки Windows Forms платформы .NET [3].

Учебник состоит из описаний 19 проектов. Его содержимое можно разбить на три части. В первой части рассматриваются базовые возможности библиотеки WPF: создание проекта для WPF-приложения, работа с xaml-файлами, использование группирующих компонентов, управление программой посредством обработчиков событий (проект EVENTS), приемы работы с окнами в WPF-приложении, организация взаимодействия между окнами, особенности диалоговых окон (проект WINDOWS), совместное использование обработчиков событий, события клавиатуры (проект CALC), таймеры в WPF-приложении (проект CLOCK), возможности полей ввода, организация проверки правильности введенных данных (проект TEXTBOXES), события мыши (проект MOUSE), механизм перетаскивания Drag & Drop (проект ZOO), работа с курсорами и иконками, создание ресурсов приложения, совместное использование средств WPF и Windows Forms (проект CURSORS).

Вторая часть содержит описание разработки одного большого проекта TEXTEDIT, разбитое на 5 этапов (версий проекта). В ней основное внимание уделяется особенностям тех стандартных интерфейсных элементов, без которых не обходится практически ни одно приложение: меню и различных видов его команд (версии 1 и 2), контекстных меню (версия 3), панели инструментов (версия 4), статусной панели (версия 5). Кроме того, в версии 1 подробно рассматриваются особенности организации работы с файлами (открытие, сохранение, контроль за сохранением внесенных изменений), а также описываются приемы работы с командами WPF; в версии 2 рассказывается о том, как создавать новые команды WPF; в версии 3 рассматриваются особенности реализации команд редактирования; а в версиях 4 и 5 особое внимание уделяется различным аспектам механизма привязки компонентов и, кроме того, описываются действия по созданию новых свойств зависимости.

Третья часть содержит проекты, связанные с дополнительными возможностями WPF – работа с цветами и кистями и использование компонентов TrackBar (проект COLORS), работа со списками и использование стилей при определении макета приложения (проект LISTBOXES), работа с флажками и наборами флажков (проект CHECKBOXES), работа с иерархическими списками, реализация дерева каталогов и списка файлов, использование компонентов GridSplitter и Image, применение реестра Windows для хранения настроек приложения (проект IMGVIEW), работа с табличными списками и использование заставок и градиентных кистей (проект TRIGFUNC), создание компонентов во время выполнения программы (проект HTOWERS).

Завершает книгу раздел с 48 учебными заданиями, разбитыми на 5 групп. Каждая группа содержит однотипные задания; первая группа включает задания на организацию взаимодействия между окнами приложения, вторая – на синхронизацию компонентов и совместное использование обработчиков событий, третья – на реализацию режима перетаскивания Drag & Drop, четвертая – на создание программ, управляемых таймером, пятая – на использование стандартных диалоговых окон и работу с реестром. Большое количество заданий позволяет легко формировать из них наборы индивидуальных заданий одинакового уровня сложности.

За рамками настоящей книги осталось большинство возможностей библиотеки WPF, связанных с ее графической подсистемой, поскольку детальное изучение этих возможностей более естественно отнести к курсу по компьютерной графике.

Предполагается, что читатель книги уже изучил базовый курс по программированию. Знание основ языка C#, а также языка XML, является желательным, но не обязательным, так как ознакомиться с конструкциями этих языков можно в процессе чтения книги. Тем не менее полезной может оказаться книга [5], освещающая практически все аспекты языка C# и стандартных библиотек платформы .NET, а также книга [4], содержащая описание основных типов стандартной библиотеки (в том числе классов, связанных с обработкой строк и файлов), объектной модели языка C# и технологии LINQ, в том числе интерфейса LINQ to XML.

В качестве среды программирования используется Microsoft Visual Studio 2015, однако все проекты можно реализовать и в более ранних версиях этой среды (начиная с версии 2008). Применение новых возможностей языка C#, появившихся в его версии 6.0 и доступных только в Visual Studio версий 2015 и выше, всегда особо отмечается и сопровождается альтернативными вариантами кода для предыдущих версий.

Соглашения по оформлению фрагментов программного кода и xaml-файлов приводятся в первом проекте EVENTS. В нем же описываются основные действия по созданию и редактированию проекта для WPF-приложения.

1. События: EVENTS

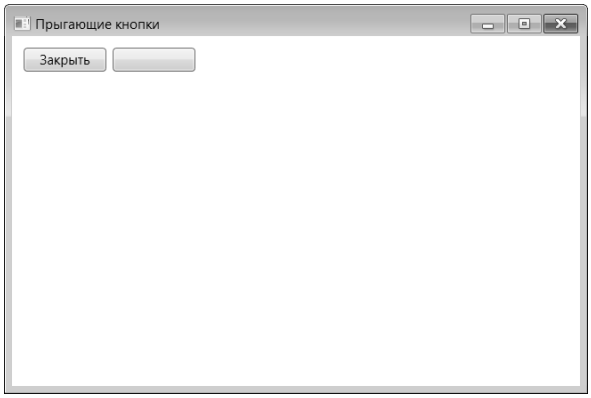


Рис. 1. Окно приложения EVENTS

1.1. Создание проекта для WPF-приложения

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

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

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

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

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

App.xaml:

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

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

App.xaml.cs:

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

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

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

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

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

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

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

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

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

Комментарий

На протяжении всей книги мы будем придерживаться следующих соглашений об отступах в текстах xaml- и cs-файлов. В xaml-файле каждый вложенный элемент набирается с *отступом в 2 пробела* относительно родительского элемента (причина столь небольшого отступа заключается в том, что глубина вложенности элементов в xaml-файлах может быть достаточно большой); если список атрибутов в открывающем теге элемента не умещается в одной строке, то он переносится на следующую строку с *отступом в 4 пробела* относительно начала открывающего тега. Для cs-файла ситуация обратная: вложенные конструкции набираются с *отступом в 4 пробела* (как в редакторе кода среды Visual Studio), а при переносе длинного оператора на новую строку используется *отступ в 2 пробела*.

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.

Комментарий

Большинство XML-атрибутов в xaml-файле относятся либо к *атрибутам свойств* (и определяют соответствующие свойства объектов), либо к *атрибутам событий* (и позволяют связать события с методами-обработчиками). XML-элементы тоже можно разбить на две категории: это *элементы-объекты*, имена которых совпадают с типом определяемого объекта, и *элементы-свойства*, имеющие составные имена вида *тип.свойство* (элементы-свойства используются в ситуации, когда свойство нельзя определить с помощью единственного атрибута).

Кроме того, для каждого типа компонентов WPF определено особое свойство, которое можно задать в xaml-файле, указав его в виде одного или нескольких *дочерних* элементов-объектов (примером такого свойства является свойство Content; в частности, в приведенном выше файле MainWindow.xaml свойство Content окна Window равно компоненту Grid). В подобной ситуации имя свойства вообще не указывается. Несколько дочерних элементов-объектов можно указывать, если определяемое свойство является свойством-коллекцией (примером такого свойства является свойство Children, имеющееся у всех *группирующих* компонентов-панелей, например, компонента Grid или используемого в следующем пункте компонента Canvas).

1.2. Добавление компонентов и настройка их свойств

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

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

Опишем первый способ.

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

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

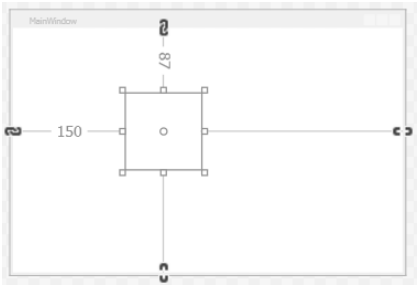


Рис. 2. Окно дизайнера после добавления компонента Canvas

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.

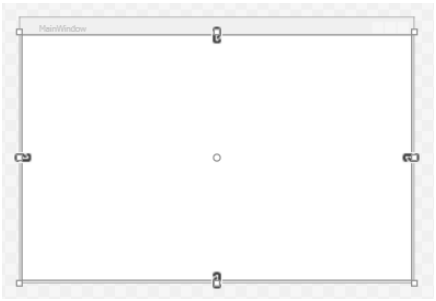


Рис. 3. Окно дизайнера после удаления атрибутов компонента Canvas

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, расположенной на расстоянии 10 единиц от левой и верхней границы клиентской области окна, – рис. 4).

Рис. 4. Окно дизайнера после добавления компонента Button

Содержимое xaml-файла изменится следующим образом:

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

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

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

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

Следует также обратить внимание на то, что для кнопки не указано свойство Height (хотя свойство Width имеется). Если свойство Height отсутствует, то высота компонента определяется по размерам его содержимого, что в большинстве случаев является оптимальным. Можно было бы удалить и свойство Width, тогда все размеры кнопки будут подстроены под ее содержимое, однако обычно свойство Width указывается, поскольку желательно, чтобы все кнопки в приложении имели одинаковую ширину.

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

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

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

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

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

Отредактируем полученный xaml-файл: изменим заголовок окна на текст «Прыгающие кнопки», надпись на кнопке – на «Закрыть», ее имя – на button1 (поскольку в дальнейшем мы добавим к окну еще одну кнопку). Кроме того, укажем для окна свойство WindowStartupLocation, положив его равным CenterScreen (это значение обеспечивает автоматическое центрирование окна программы при ее запуске):

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

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

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

Комментарий

При редактировании xaml-файла оказываются удобными две возможности, связанные с автоматическим добавлением или удалением закрывающих тегов.

(1) Если ввести новый открывающий тег, включая все его атрибуты, то после ввода закрывающей угловой скобки к открывающему тегу будет добавлен закрывающий, а курсор разместится между тегами.

Пример. Предположим, что был введен следующий текст (позиция курсора помечена символом |):

```
<Button Content="Закрыть"|
```

Если теперь ввести символ «>», то текст изменится следующим образом (символ | по-прежнему указывает на позицию курсора):

```
<Button Content="Закрыть">|</Button>
```

(2) Если перед закрывающей угловой скобкой открывающего тега ввести символ «/» (превратив этим действием тег в *комбинированный*),

то соответствующий закрывающий тег будет удален из xaml-файла (при этом все дочерние элементы преобразованного элемента, если они имеются, станут элементами того же уровня, что и преобразованный элемент).

Пример. Если в тексте, полученном в предыдущем примере, перевести курсор на одну позицию влево

```
<Button Content="Закрыть"|></Button>
```

и ввести символ «/», то текст изменится следующим образом:

```
<Button Content="Закрыть"/>|
```

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

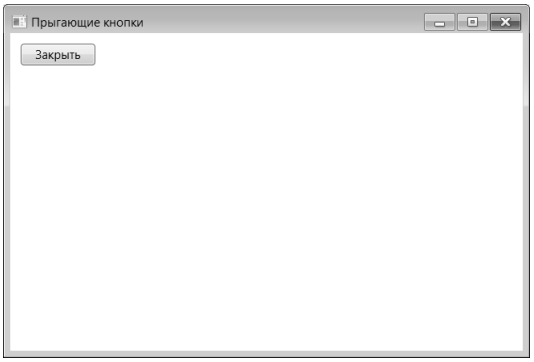


Рис. 5. Окно приложения EVENTS (первый вариант)

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

Комментарий

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

Рис. 6. Панель с дополнительными отладочными средствами XAML

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

1.3. Связывание события с обработчиком

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

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

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

}
```

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

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

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

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

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

Описанный выше способ создания нового обработчика события был реализован еще для библиотеки Windows Forms. Однако в WPF-проекте имеется более быстрый способ определения нового обработчика, не требующий использования окна Properties. Необходимо ввести имя события как атрибут соответствующего элемента в xaml-файле (в нашем случае в элемент Button надо ввести текст «Click=», причем достаточно набрать несколько начальных символов имени события и воспользоваться для завершения набора выпадающим списком) и после появления рядом с набранным атрибутом выпадающего списка с текстом «New Event Handler» выбрать этот текст (если он еще не выбран) и нажать клавишу Enter.

При этом в xaml-файл будет добавлено имя обработчика (в нашем случае button1_Click), а в cs-файле будет создана заготовка для обработчика с этим именем, *хотя перехода к ней не произойдет*, чтобы дать возможность продолжить редактирование xaml-файла. Если в программе уже имеются обработчики, совместимые с тем событием, имя которого введено в xaml-файле, то в выпадающем списке наряду с вариантом «New Event Handler» будут приведены и имена всех таких обработчиков, что позволит быстро связать события для *нескольких* компонентов с *одним* обработчиком (хотя для подобного связывания имеется более удобная возможность, основанная на механизме *маршрутизируемых событий* и описанная в проекте CALC).

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

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

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

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

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

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

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

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

Кроме того, необходимо задать фон для компонента Canvas:

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

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

Комментарии

1. Любой компонент WPF реагирует на нажатие мыши только в том случае, если имеет непустой фон. По умолчанию фон некоторых компонентов является пустым (в окне Properties в этом случае свойство Background имеет значение No Brush или вообще не содержит никакого текста). Следует заметить, что, несмотря на вид атрибута Background в xaml-файле, фон определяется не цветом, а особым классом WPF – *кистью* (имеется абстрактный класс Brush, от которого порождается несколько классов-потомков). Указание цветовой константы означает, что будет использоваться *сплошная* кисть типа SolidBrush с заливкой данного цвета. В последних проектах, описаннных в данной книге (TRIGFUNC и HTOWERS), мы познакомимся с *градиентной* кистью, имеющей две разновидности – LinearGradientBrush и RadialGradientBrush.

2. Важной характеристикой любого события, связанного с мышью, является позиция мыши. Для определения этой позиции в обработчиках мыши предусмотрен специальный метод, вызываемый для второго параметра обработчика e: e.GetPosition. Данный метод имеет обязательный параметр, задающий компонент, относительно которого определяется позиция мыши. Мы указали параметр this; это означает, что позиция будет определяться относительно левого верхнего угла клиентской области окна. Заметим, что все размеры в WPF задаются в так называемых *аппаратно-независимых единицах* (одна единица равна 1/96 дюйма) и представляются в виде вещественных чисел типа double (в то время как в библиотеке Windows Forms размеры задавались в экранно-зависимых *пикселах* и представлялись целыми числами).

3. Два последних оператора в обработчике Canvas_MouseDown демонстрируют способ, позволяющий задать в программе *присоединенные свойства* Left и Top, полученные компонентом от его родителя типа Canvas. Обратите внимание на то, что методы SetLeft и SetTop являются статическими и должны вызываться не для конкретного объекта типа Canvas, а для самого класса. Имеются парные методы Canvas.GetLeft(c) и Canvas.GetTop(c), позволяющие определить текущие значения свойств Left и Top для компонента c, расположенного на компоненте Canvas (эти методы будут использованы во фрагменте программы, добавленном в п. 1.5 при исправлении недочета).

Для задания присоединенных свойств можно также использовать «универсальный» метод SetValue, имеющийся у всех компонентов. Например, два последних оператора в обработчике Canvas_MouseDown можно изменить следующим образом:

```
button1.SetValue(Canvas.LeftProperty,
  p.X - button1.ActualWidth / 2);
button1.SetValue(Canvas.TopProperty,
  p.Y - button1.ActualHeight / 2);
```

Обратите внимание на то, что при указании присоединенного свойства в методе SetValue надо использовать его имя с суффиксом Property (на самом деле это и есть «настоящее» имя статического присоединенного свойства, поскольку подобный суффикс имеют *все* статические свойства зависимости). Интересно отметить, что с помощью метода SetValue с компонентом можно связывать *любые* свойства зависимости, определенные у *любых* типов компонентов (для получения значений этих свойств предназначен метод GetValue, пример использования которого приводится в последнем комментарии к п. 1.5).

4. Для определения *текущих размеров* компонента в программе надо обращаться к свойствам ActualWidth и ActualHeight. Свойства Width и Height для этого использовать нельзя, так как они обычно содержат лишь «рекомендованные» значения размеров, которые учитываются группирующими компонентами при компоновке своих дочерних компонентов (в частности, возможны рекомендованные значения «бесконечность» или NaN). В нашем случае свойства ActualWidth и ActualHeight кнопки используются для того, чтобы отцентрировать кнопку относительно курсора мыши.

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.

Комментарии

1. В данном пункте демонстрируется возможность *отсоединения* метода-обработчика от события, с которым он ранее был связан. Для этого используется операция −=, слева от которой указывается событие, а справа – тот обработчик, который надо отсоединить от события.

2. В обработчике button2_MouseMove определяются текущие размеры компонента Canvas, чтобы обеспечить случайное перемещение дикой кнопки только в пределах этого компонента (метод r.NextDouble() возвращает случайное вещественное число в полуинтервале [0; 1), при этом вычитание числа 5 гарантирует, что дикая кнопка будет видна на экране хотя бы частично). Заметим, что программа правильно реагирует на изменение размера окна: дикая кнопка всегда перемещается в пределах его текущего размера. Это обеспечивается благодаря тому, что панель Canvas по умолчанию занимает всю клиентскую область своего родителя-окна.

Поскольку мы не присвоили компоненту Canvas имя, нам пришлось обращаться к нему через его родителя, вызвав для окна его свойство Content и, кроме того, выполнив явное приведение типа с помощью операции as. Вместо приведения к типу Canvas можно было бы выполнить приведение к типу FrameworkElement – первому типу в иерархии наследования компонентов, в котором определены свойства, связанные с размерами. Можно было выполнить приведение к типу Panel – непосредственному потомку FrameworkElement, который является предком всех группирующих компонентов. Заметим, что выполнить приведение класса Canvas к типу Control не удастся, так как группирующие компоненты к данному типу не относятся (потомками Control являются, в частности, компоненты, имеющие свойство Content, например кнопки).

3. Следует обратить внимание на способ, с помощью которого в обработчике button2_MouseMove проверяется, нажата ли клавиша Ctrl. Обычно дополнительная информация о произошедшем событии передается в обработчик с помощью второго параметра e. Например, в обработчике button2_MouseMove с помощью данного параметра (типа MouseEventArgs) можно определить текущую позицию мыши (вызвав метод e.GetPosition) или состояние кнопок мыши (вызвав, например, свойство e.LeftButton и сравнив его с одним из его возможных значений – MouseButtonState.Pressed или MouseButtonState.Released). Однако информацию о нажатых в данный момент *клавишах* параметр e типа MouseEventArgs не содержит. Тем не менее подобную информацию можно получить с помощью статического свойства IsKeyDown класса Keyboard. Это же свойство мы использовали при исправлении обнару-

женного недочета, чтобы в случае, если нажата клавиша пробела, обработчик button2_MouseMove не выполнял никаких действий.

4. Обратите также внимание на то, что поле r в классе WainWinbdow не только описывается, но и сразу *инициализируется* (с помощью конструктора Random() без параметров). В какой момент выполняется данная инициализация? По правилам языка C#, *явно указанные операторы инициализации всех полей класса автоматически помещаются в начало любого конструктора данного класса*. Таким образом, поле r будет инициализировано в начале выполнения конструктора окна (перед выполнением оператора InitializeComponent(), указанного в теле данного конструктора). Разумеется, можно было бы поступить иначе: описать поле r типа Random без его инициализации

```
Random r;
```

и поместить в конструктор окна оператор

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

1.5. Присоединение к событию другого обработчика

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

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

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

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

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

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

В метод Canvas_MouseDown добавьте операторы:

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

Результат. Прирученная кнопка теперь выполняет полезную работу – щелчок на ней приводит к разворачиванию окна программы на весь экран, а новый щелчок восстанавливает первоначальное состояние окна. Если же щелкнуть мышью на окне (не на кнопке), то услужливая кнопка «Закрыть» прибежит на вызов, а прирученная кнопка «Изменить» снова одичает, потеряет текст своего заголовка и начнет убегать от мыши.

Комментарий

Приведенные тексты методов показывают, что при смене обработчика недостаточно присоединить к событию новый обработчик; необходимо также отсоединить от события обработчик, ранее связанный с ним. Данное обстоятельство обусловлено тем важным фактом, что *к одному и тому же событию можно последовательно присоединить несколько обработчиков* (для этого достаточно применить к этому событию несколько раз оператор +=). Следует отметить, что данная возможность для событий визуальных компонентов применяется крайне редко (достаточно отметить, что с помощью окна Properties или xaml-файла присоединить к одному событию несколько обработчиков *нельзя*). В то же время при явном присоединении обработчиков эта особенность может приводить к появлению трудно выявляемых ошибок, если, например, один и тот же обработчик будет присоединен к событию несколько раз. Подобные проблемы можно проиллюстрировать с помощью нашей программы, если закомментировать заголовок оператора if в обработчике Canvas_MouseDown:

```
//  if ((string)button2.Content == "Изменить")
```

Если теперь после запуска программы *несколько раз* щелкнуть мышью на окне, а затем приручить кнопку button2, то при ее последующем нажатии окно *несколько раз* последовательно перейдет из развернутого состояния в стандартное и обратно. Это объясняется тем, что теперь каждый щелчок на окне присоединяет к событию Click кнопки button2 *новый экземпляр* обработчика button2_Click, и при нажатии на эту кнопку каждый экземпляр обработчика последовательно запускается на выполнение. Ситуация осложняется еще тем обстоятельством, что в программе *невозможно* выяснить, сколько и какие обработчики присоединены в настоящий момент к данному событию (а не зная этого, нельзя и обеспечить отсоединение от события всех его обработчиков).

Итак, к действиям по явному присоединению обработчика к событию и его последующему отсоединению следует подходить *крайне осторожно*.

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.

Комментарии

1. В данном обработчике демонстрируется еще один способ доступа к компонентам окна, который удобен для организации перебора в цикле компонентов с похожими именами. Этот способ основан на применении метода FindName, который можно вызывать непосредственно для окна. Метод FindName возвращает компонент окна с указанным именем (или null, если компонент с таким именем в окне отсутствует).

2. Вместо статического метода GetLeft для получения значения присоединенного свойства Left можно было бы использовать более длинный, но и более универсальный вариант, использующий метод GetValue того

компонента, к которому ранее было присоединено свойство: (double)b.GetValue(Canvas.LeftProperty). Аналогичным образом можно получить значение свойства Top (и любых других свойств зависимости, присоединенных к данному компоненту).

2. Работа с несколькими окнами: WINDOWS

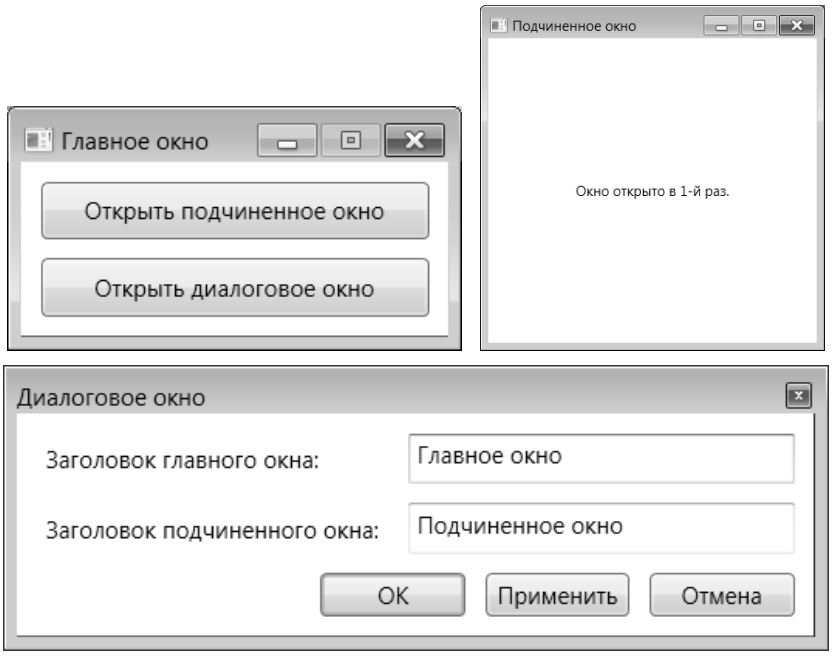


Рис. 7. Окна приложения WINDOWS

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.

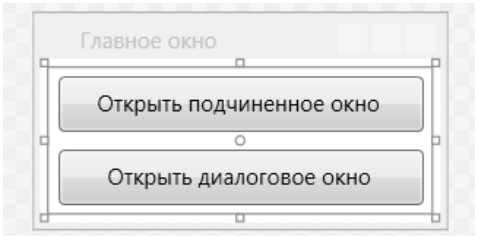


Рис. 8. Макет окна MainWindow приложения WINDOWS

MainWindow.xaml (рис. 8):

```
<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.

Комментарии

1. Благодаря явному заданию значения false для свойств ShowInTaskbar подчиненных окон, кнопки для этих окон не отображаются на панели задач в нижней части экрана.

2. За возможность изменения размеров окна и отображение кнопок минимизации/максимизации на его заголовке отвечает свойство ResizeMode, которое может принимать следующие значения: NoResize (размер окна фиксирован, кнопки не отображаются), CanMinimize (размер окна фиксирован, доступна кнопка минимизации), CanResize (значение по умолчанию: окно может менять размер, доступны обе кнопки), CanResizeWithGrip (то же, что и CanResize, но в правом нижнем углу окна дополнительно отображается треугольный маркер; благодаря этому маркеру увеличивается область, которую можно зацепить мышью для изменения размеров окна). Для диалоговых окон дополнительно следует установить свойство WindowStyle равным ToolWindow; это обеспечивает

скрытие иконки на заголовке окна (отображать иконки в диалоговых окнах не принято).

3. Присваивание свойству Owner некоторого окна w1, значения какого-либо другого окна w0 делает окно w1 *подчиненным* по отношению к *главному* окну w0. Подчиненное окно всегда отображается поверх главного (даже если главное окно является активным). Кроме того, при минимизации или закрытии главного окна его подчиненные окна также минимизируются (или, соответственно, закрываются). Следует заметить, что свойству Owner можно присвоить значение только такого окна, которое уже отображено на экране, поэтому указанные действия мы выполняем в обработчике события Loaded, которое возникает при *первом* отображении окна.

4. За начальное расположение окна на экране отвечает свойство WindowStartupLocation, равное по умолчанию значению Manual. При этом позицию окна можно задать явно с помощью свойств Left и Top или не задавать эти свойства, оставив определение начальной позиции на усмотрение операционной системы. В последнем случае «истинные» значения свойств Left и Top будут доступны только в момент первого отображения окна на экране. Как уже было отмечено в предыдущем комментарии, с этой ситуацией связано событие окна Loaded, поэтому начальное положение окна win1 определяется нами в обработчике данного события для главного окна.

5. Содержимое файла MainWindow.xaml демонстрирует традиционный для технологии WPF *динамический способ размещения компонентов в окне*, при котором их размеры и положение (а также иногда и размеры окна) определяются автоматически на основе указанных настроек. В данном случае в окне надо разместить две кнопки по вертикали. Такой способ размещения проще всего обеспечить с помощью группирующего компонента StackPanel (данный компонент имеет свойство Orientation с вариантами значений Vertical и Horizontal, причем первый вариант является значением по умолчанию).

Для указания *полей* – промежутков между компонентами – используется свойство Margin, которое может состоять из 1, 2 или 4 значений. Единственное значение определяет одинаковое поле (в *аппаратно-независимых единицах*, равных 1/96 дюйма) во всех направлениях, при наличии двух значений первое определяет поле слева и справа, а второе – сверху и снизу, при наличии четырех значений поля определяются в следующем порядке: левое, верхнее, правое, нижнее. Обратите внимание на то, что для того, чтобы обеспечить одинаковые промежутки (равные 10 единицам) как между компонентами, так и между компонентом и границей окна, следует задать поля, равные 5, как для группирующего

(невидимого) компонента, так и для содержащихся в нем видимых компонентов-кнопок.

Помимо «внешних полей» (margins) для компонентов можно задавать «внутренние поля» (paddings), определяющие расстояние от границы компонента до его содержимого. Внутренние поля определяются свойством Padding, которое задается по тем же правилам, что и свойство Margin.

Следует также обратить внимание на значение свойства MinWidth, которое задано только для первой кнопки. Оно определяет минимальную ширину данного компонента и тем самым минимальную ширину всей панели StackPanel, причем все остальные компоненты на этой панели будут иметь такую же ширину. Таким образом, реальные размеры как кнопок, так и панели будут определяться размером шрифта, используемого для надписей на кнопках. Если шрифт велик настолько, что текст по ширине будет превосходить указанную минимальную ширину в 200 единиц, то свойство MinWidth будет проигнорировано и ширина кнопки станет больше 200 единиц; при этом кнопка по-прежнему будет иметь указанные внутренние и внешние поля.

В окнах, подобных главному окну из нашего проекта, желательно, чтобы их размер подстраивался под размер содержимого (в данном случае – панели StackPanel). Для этого предусмотрено свойство окна SizeToContent, которое мы положили равным WidthAndHeight (можно также подстраивать под размер содержимого только ширину или только высоту окна). По умолчанию данное свойство равно Manual, в этом случае не окно подстраивается под свое содержимое, а наоборот – компоненты подстраиваются под размер окна. Заметим, что если оставить в xaml-файле атрибуты Width и Height для окна, то в *окне дизайнера* окно будет иметь указанные размеры даже при наличии атрибута SizeToContent, равного WidthAndHeight, однако при *выполнении программы* явно указанные размеры окна будут игнорироваться.

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, разрушения же 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.

Комментарии

1. Событие Closing относится к группе событий, которые возникают *перед* выполнением некоторого действия и позволяют *отменить* его (имена этих событий оканчиваются на -ing). Второй параметр e у обработчиков подобных событий имеет изменяемое свойство Cancel, которому следует присвоить значение true, если требуется отменить соответствующее действие. В приведенном обработчике отменяется *закрытие окна*; вместо этого оно просто *удаляется с экрана* методом Hide (аналогичного результата можно добиться, установив значение его свойства Visibility равным значению Visibility.Hidden). Заметим, что сделанное изменение не препятствует «настоящему» закрытию подчиненных окон при закрытии главного окна приложения.

2. Избежать выявленной в данном пункте ошибки можно было бы, создавая подчиненные окна *заново* каждый раз перед их отображением. Однако такой способ требует дополнительных действий, если при повторном отображении окна необходимо *восстанавливать* его в том виде, который оно имело в момент закрытия, в то время как способ, использованный в нашем проекте, подобных действий не требует.

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.

Комментарий

В то время как главное окно для доступа к подчиненному может просто обратиться к нему по имени, подчиненное окно так сделать не может, поскольку имя главного окна ей неизвестно (главное окно в нашем проекте имени вообще не имеет). Однако подчиненное окно может обратиться к главному, используя свое свойство Owner. Для доступа к конкретному компоненту главного окна, имеющему имя, мы воспользовались методом FindName. Можно было поступить по-другому: выполнить явное приведение объекта Owner к типу MainWindow и после этого обратиться к его свойству button1:

```
(Owner as MainWindow).button1.Content = …
```

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.

Комментарии

1. Добавленное в описание класса Window1 поле count при создании окна автоматически инициализируется нулем; в дальнейшем это поле можно вызывать из любого метода класса. Новые поля позволяют хранить дополнительную информацию о состоянии окна.

2. Напомним, что при использовании *операции инкремента* вида ++i вначале происходит увеличение значения переменной i на 1, а затем данная переменная используется в выражении. Для операции i++ действия выполняются в обратном порядке: вначале прежнее значение i используется в выражении, а затем это значение увеличивается на 1.

3. В данном случае содержимым окна является не группирующий, а «обычный» компонент. Особенностью использованного компонента TextBlock является то, что его содержимым может быть только строка (этот компонент не имеет свойства Content, зато имеет свойство Text типа string). Для обеспечения центрирования текста по обоим измерениям достаточно установить соответствующие значения свойств HorizontalAlignment и VerticalAlignment компонента TextBlock.

2.5. Модальные и обычные кнопки диалогового окна

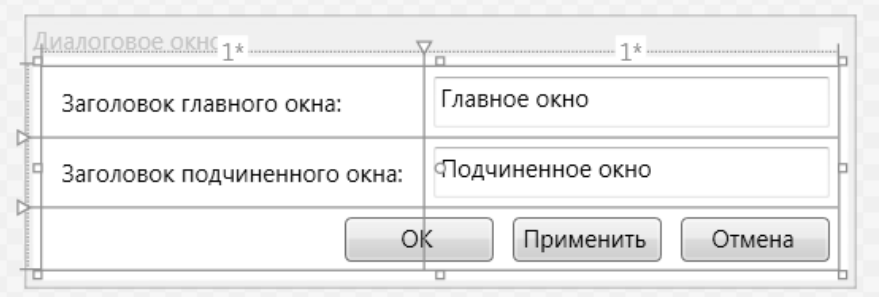


Рис. 9. Макет окна Window2 приложения WINDOWS

```
<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.

Комментарии

1. Для того чтобы нажатие на кнопку «Отмена» приводило к закрытию диалогового окна (а также чтобы нажатие клавиши Esc интерпретировалось как нажатие на эту кнопку), для данной кнопки надо установить равным true свойство IsCancel. Для того чтобы кнопка «OK» считалась кнопкой по умолчанию (и нажатие клавиши Enter интерпретировалось как нажатие на эту кнопку), для данной кнопки надо установить равным true свойство IsDefault. Заметим, что хотя кнопка «OK» сделана кнопкой по умолчанию, для нее все равно необходимо определить обработчик события Click (для кнопки «Отмена» обработчик определять не требуется).

2. Доступ из окна win2 к свойству Title окна win1 возможен благодаря тому, что эти окна имеют общего владельца (Owner), который хранит список своих подчиненных окон (в порядке их добавления) в свойстве-коллекции OwnedWindows.

3. Включенное в класс Window2 свойство DialogRes позволяет определить *способ закрытия* диалогового окна: если окно было закрыто по нажатию кнопки «ОК», то свойство равно true, если окно было закрыто по нажатию кнопки «Отмена» (или каким-либо другим способом, например, по нажатию кнопки закрытия на заголовке окна), то свойство равно false. Это свойство проверяется в главном окне после возврата из метода ShowDialog.

Следует сказать, что метод ShowDialog из библиотеки WPF тоже возвращает значение, позволяющее определить, каким образом было закрыто диалоговое окно (это значение имеет тип bool?, т. е. может быть равно true, false и null, и определяется по значению стандартного свойства окна DialogResult того же типа), однако данный механизм корректно работает только в ситуации, когда диалоговое окно действительно закрывается, а не просто удаляется с экрана, как в нашем случае.

4. Явный вызов метода button2_Click класса Window2 в обработчике button2_Click класса MainWindow обеспечивает выполнение действий, связанных с нажатием на кнопку «Применить» (таким образом, данный вызов *имитирует нажатие на кнопку*). При вызове этого метода в качестве параметров указаны константы null, так как значения параметров в методе button2_Click класса Window2 не используются.

Для возможности вызова метода button2_Click класса Window2 из класса MainWindow модификатор доступа для данного метода необходимо изменить с private на public или internal (модификатор internal обеспечивает доступ к данному методу *в пределах создаваемого проекта*).

Заметим, что в данном случае можно было бы обойтись без модификации метода button2_Click класса MainWindow: достаточно просто вызывать метод button2_Click класса Window2 в уже имеющемся обработчике button1_Click *этого же класса* Window2:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    button2_Click(null, null);
    dialogRes = true;
    Close();
}
```

При этом отпадает необходимость в изменении модификатора метода button2_Click с private на public, и, кроме того, можно вообще обойтись без свойства DialogRes.

5. Макет окна Window2 демонстрирует те же особенности компоновки, что и ранее обсуждавшийся макет главного окна, только в более сложном варианте. В нем, как и в главном окне, все компоненты размещаются с учетом их «истинных» размеров, причем размер окна подстраивается под размер компонентов. В данном случае вместо панели StackPanel используется более сложный группирующий компонент Grid, позволяющий размещать данные по строкам и столбцам. Следует обратить внимание на способ задания количества строк и столбцов (мы использовали простейший способ; в более сложных ситуациях можно явно указывать размеры некоторых строк и столбцов или настраивать их размеры с соблюдением требуемых пропорций – см. далее проект IMGVIEW).

Номер ячейки, которую должен занимать компонент, определяется присоединенными свойствами Grid.Row и Grid.Column (которые «делегируются» дочерним компонентам таким же образом, как и рассмотренные в проекте EVENTS свойства Left и Top компонента Canvas). Если для компонента не указывать свойства Grid.Row или Grid.Column, то их значение считается равным 0, т. е. соответствует первой строке или первому столбцу компонента Grid. Настраивая присоединенное свойство Grid.ColumnSpan, можно обеспечить «захват» компонентом нескольких столбцов (имеется также парное свойство Grid.RowSpan). Мы использовали свойство ColumnSpan для размещения набора кнопок по всей ширине нижней строки компонента Grid, сгруппировав их с помощью вспомогательной горизонтальной панели StackPanel и выровняв эту панель по правой границе родительского компонента Grid.

Обратить внимание на интересную особенность полученного макета. Поскольку для одного из полей ввода мы задали свойство MinWidth, ширина полей ввода не может стать меньше значения этого свойства, но *может увеличиваться*, если ее минимального размера недостаточно для отображения введенного текста. При этом будет пропорционально увеличиваться и ширина компонента Grid, и ширина всего окна, причем кнопки будут по-прежнему выровнены по правой границе.

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 «ОК» 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.

Комментарии

1. Отметим, что указанное действие по установке фокуса происходит при *скрытии* окна. В этом можно убедиться, если добавить перед оператором установки фокуса условие:

```
if (!IsVisible)
    textBox1.Focus();
```

В то же время, если использовать вариант

```
if (IsVisible)
    textBox1.Focus();
```

то фокус на первом поле ввода при последующих открытиях окна устанавливаться *не будет*.

Подобное странное поведение объясняется двумя такими же странными особенностями библиотеки WPF. Во-первых, метод Focus обеспечивает установку фокуса для указанного компонента только в случае, если *в момент вызова метода окно отображается на экране* (хотя более естественным было бы реализовать метод таким образом, чтобы он *в любом случае* сохранял информацию об установке фокуса и учитывал ее при отображении окна). Заметим, что метод Focus возвращает логическое значение, которое равно true, если вызов метода действительно обеспечил успешную установку фокуса на данном элементе.

Во-вторых, при установке свойства IsVisible равным true событие IsVisibleChanged наступает *до того момента*, как окно появится на экране, и наоборот, при установке свойства IsVisible равным false событие IsVisibleChanged выполняется, когда окно *еще отображается* на экране. Подобное поведение тоже представляется нелогичным, поскольку не позволяет связать некоторые действия (например, установку фокуса) с тем моментом, когда окно в очередной раз отображается на экране.

Тем не менее при *первом* отображении диалогового окна поле ввода textBox1 все же получает фокус, хотя это, казалось бы, противоречит ска-

занному выше. Это связано с особенностями реализации механизма настройки фокуса в WPF: если в окне *еще не установлен активный компонент*, то *первый* вызов метода Focus обеспечит установку фокуса на требуемый компонент даже при невидимом окне, несмотря на то, что этот вызов вернет значение false. Если же активный компонент уже был установлен ранее, когда окно еще отображалось на экране, то последующие вызовы метода Focus при скрытом окне не позволят изменить фокус.

Описанные особенности демонстрируют сложность и некоторую непоследовательность реализации механизма работы с фокусом в WPF. Отметим, что в WPF имеются два вида фокуса: *клавиатурный* и *логический*. Для обработки клавиатурного фокуса можно использовать методы класса Keyboard. В частности, свойство Keyboard.FocusedElement, доступное только для чтения, позволяет определить элемент приложения, имеющий в данный момент фокус, а метод Keyboard.Focus(comp) позволяет установить фокус на компонент comp, *но только в случае, если этот компонент отображается на экране* и находится в активном в данный момент окне. Для работы с логическим фокусом предназначен класс FocusManager. В частности, он позволяет устанавливать различные *области фокусировки*, в каждой из которых может быть определен свой логический фокус, а также получать и изменять логический фокус для каждой области фокусировки fscope методами GetFocusedElement(fscope) и SetFocusedElement(fscope, comp). Если какая-либо область фокусировки теряет клавиатурный фокус, то, тем не менее, в ней сохраняется информация о том ее компоненте, который имеет логический фокус. Поэтому в дальнейшем данный компонент автоматически получит клавиатурный фокус, если фокус примет сама область. К сожалению, вся эта красивая схема работает только в случае, когда окно отображается на экране. Изменить активный компонент для *скрытого* окна, *если в нем уже имеется активный компонент*, описанными выше средствами невозможно. В частности, даже если для скрытого окна попытаться установить логический фокус на другой компонент, при отображении этого окна фокус получит тот компонент, который имел фокус в момент скрытия окна, а не тот, для которого (при скрытом окне) вызывался метод SetFocusedElement. Единственная ситуация, при которой возможна установка логического фокуса для скрытого окна, – это ситуация, при кото*рой в окне ранее еще не было компонента, имеющего фокус*. Мы уже отмечали, что в этой ситуации установить фокус можно проще: обычным вызовом метода Focus().

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();
}
```

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

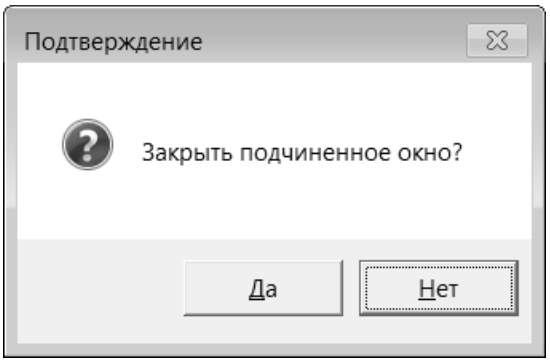


Рис. 10. Стандартное диалоговое окно

Комментарий

В обработчике использован один из наиболее полных вариантов метода Show класса MessageBox, позволяющий указать (1) текст запроса, (2) заголовок окна запроса, (3) набор кнопок для данного окна, (4) иконку в окне и (5) кнопку, предлагаемую по умолчанию. Любой параметр, кроме первого, может отсутствовать; при этом должны отсутствовать и все следующие за ним параметры. Если отсутствует второй параметр, то заголовок окна является пустым, если третий, то в окне отображается единственная кнопка «ОК», если четвертый – иконка в окне не отображается, если пятый – кнопкой по умолчанию является первая кнопка.

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

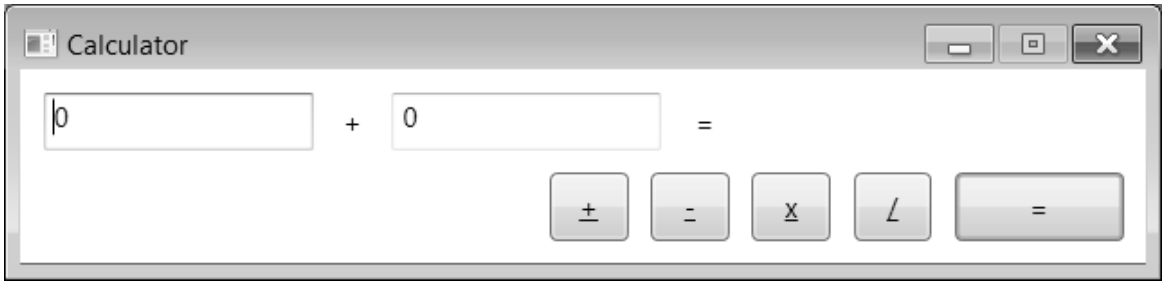


Рис. 11. Окно приложения CALC

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.

Комментарии

1. Поскольку при нажатии на любую из кнопок с обозначением арифметической операции следует выполнять однотипные действия, создавать для каждой кнопки особый обработчик события Click нецелесообразно. В приложениях Windows Forms в подобной ситуации создается один обработчик, который затем связывается с соответствующими событиями *всех* требуемых компонентов. Такой подход возможен и в WPF-приложениях. В нашем случае его можно реализовать, определив обработчик button1_Click для кнопки button1 и указав в xaml-файле атрибут Click="button1_Click" *для всех четырех кнопок* button1–button4. При этом

оператор в обработчике button1_Click можно изменить, указав вместо e.Source параметр sender (оба варианта будут работать одинаково):

```
label1.Content = (sender as Button).Content;
```

Однако в случае WPF-приложений можно использовать другой подход, который позволяет избежать явного связывания обработчика с событиями для нескольких компонентов. Подход основан на механизме *маршрутизируемых событий* (routed events), благодаря которому информация о возникших событиях может передаваться *по цепочке компонентов*. В WPF почти все стандартные события являются маршрутизируемыми. При этом все маршрутизируемые события делятся на три категории: *прямые* (direct events), которые ведут себя как обычные события .NET и не передаются по цепочке наследования (примером такого события является MouseEnter); *туннелируемые* (tunneling events), которые возникают в компоненте верхнего уровня и «спускаются» по цепочке его дочерних компонентов к компоненту, в котором фактически произошло действие, вызвавшее данное событие (например, событие PreviewTextInput, которое будет использовано далее в нашем проекте), и *пузырьковые* (bubbling events), которые «поднимаются» от компонента, где произошло событие, вверх по цепочке его родительских компонентов (например, событие MouseDown или использованное в данном пункте событие Click). Заметим, что в названиях всех туннелируемых событий используется префикс Preview и событие с таким префиксом наступает *до* наступления одноименного события без этого префикса.

На всем пути прохождения события оно может приводить к запуску связанных с ним обработчиков. Замечательной чертой механизма маршрутизируемых событий в WPF является то, что обработчик для маршрутизируемого события можно связать даже с тем родителем, для которого соответствующее событие не определено! Именно такая ситуации имеет место в нашем случае, поскольку для компонента StackPanel не предусмотрено событие Click. Тем не менее мы смогли связать с ним обработчик для события Click, которое может возникать в его дочерних компонентах (для этого нам потребовалось уточнить имя Click именем того компонента, для которого событие Click определено: Button.Click). Подобное поведение похоже на поведение присоединенных свойств (подробно рассмотренных в проекте EVENTS), поэтому в данной ситуации говорят о *присоединенных событиях* (attached events).

Теперь при возникновении события Click у любой из кнопок панели StackPanel оно «поднимется» к родителю-панели и приведет к вызову связанного с ним обработчика button1_Click. При этом в параметре sender обработчика будет указан компонент, в котором был вызван обработчик (в нашем случае панель StackPanel), а в свойстве Source второго парамет-

ра e будет указан компонент, в котором фактически произошло событие (в нашем случае одна из кнопок).

Описанный механизм имеет одну особенность, которую необходимо учитывать: при связывании события в родительском компоненте с некоторым обработчиком мы не можем указать ту *часть* набора дочерних компонентов, для которой надо использовать обработчик. Обработчик будет вызываться для *всех* дочерних компонентов, для которых предусмотрено соответствующее событие (а также и для самого родительского компонента, если для него тоже предусмотрено это событие). Не следует думать, что указание префикса Button при определении обработчика в компоненте StackPanel ограничит действие обработчика только компонентами Button. Обработчик будет вызван для дочернего компонента *любого* типа, если в нем произойдет событие Click.

В нашем случае указанная особенность приведет к недочету в программе (он описывается в конце данного пункта).

2. В макете нашего приложения нельзя естественным образом распределить компоненты по столбцам. В такой ситуации использование группирующего компонента Grid нецелесообразно; вместо него мы используем вложенный набор панелей StackPanel (внешняя панель с вертикальной ориентацией содержит две горизонтально ориентированные панели, причем для второй горизонтальной панели дополнительно устанавливается выравнивание по правой границе).

Недочет. При нажатии на кнопку «=» между полями ввода выводится знак равенства, что не имеет смысла. Этот недочет будет исправлен в следующем пункте.

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. В случае если поля ввода содержат текст, который нельзя преобразовать в вещественное число, то выводится результат «ERROR».

Комментарии

1. При выполнении операций над числами типа double *ошибок времени выполнения не возникает*, однако результатом может быть одно из «особых» значений: double.NegativeInfinity (–∞), double.PositiveInfinity (+∞) и double.NaN («не число»).

2. Для преобразования строки в число можно использовать методы Parse и TryParse соответствующего числового типа. Метод TryParse следует применять, если возможна ситуация, когда требуемое преобразование окончится неудачей (при использовании метода Parse такая ситуация приведет к возбуждению исключения, для обработки которого потребуется писать дополнительный код; кроме того, обработка исключения требует существенно больше времени, чем обычная проверка с помощью условного оператора).

3. В методе button5_Click демонстрируются важные особенности, связанные с типом string. Во-первых, тип string можно использовать в качестве переключателя в операторе switch, во-вторых, для типа string определена операция +, в которой в качестве другого операнда (причем не обязательно второго) можно указывать *выражение любого типа*; при этом данное выражение автоматически преобразуется к типу string с помощью метода ToString, определенного для любого типа платформы .NET.

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, связанный 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;
```

Комментарий

Добавленный оператор помечает событие как *обработанное*, поэтому при передаче информации о данном событии вверх по иерархии родительских компонентов остальные обработчики не запускаются.

Итак, если для одного и того же туннелируемого или пузырькового события определен обработчик и в дочернем, и в родительском компоненте, то будут выполнены оба эти обработчика (причем порядок их вызова определяется категорией события). Однако имеется возможность прервать цепочку вызовов обработчиков данного события – для этого достаточно в одном из обработчиков пометить событие как обработанное описанным выше способом. Следует также заметить, что если туннелируемое событие (например, PreviewTextInput) помечено как обработанное, то не вызываются и все обработчики парного к нему пузырькового события (для события PreviewTextInput парным является TextInput). Соответствующий пример будет приведен в п. 3.4.

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.

Результат. Кнопка «=» (button5) сделана *кнопкой по умолчанию* и отображается в окне особым образом (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.

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

Комментарий

В WPF-проектах для выделения символов, с которыми требуется связать клавишу-ускоритель, необходимо указать перед ними символ подчеркивания «_» (в той редкой ситуации, когда символ подчеркивания требуется использовать в надписи на компоненте, надо ввести этот символ дважды). Заметим, что в проектах Windows Forms для связи символа с клавишей-ускорителем использовался символ «&». Он был заменен на символ «_», поскольку в xaml-файле (как и в любом XML-файле) символ «&» интерпретируется особым образом.

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, тоже подчеркивается.

Исправление. Измените оператор в методе button1_Click следующим образом:

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

Комментарий

Для удаления одного или нескольких начальных символов строки достаточно вызвать метод TrimStart, указав удаляемые символы в качестве параметров (если параметры не указывать, то удаляются пробельные символы). Имеется также метод TrimEnd, удаляющий конечные символы, и метод Trim, удаляющий как начальные, так и конечные символы. Чтобы в нашем случае можно было использовать данный метод, необходимо выполнить явное преобразование свойства Content к типу string.

Недочет. Теперь, когда 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).

Комментарии

1. При реализации описанных возможностей мы воспользовались тем, что событие PreviewTextInput является *туннелируемым*, т. е. вначале оно обрабатывается в родительском компоненте верхнего уровня (окне), а затем «спускается» по иерархии подчинения к тому компоненту, в котором возникло. Это позволило уже на уровне окна проанализировать введенный текст и выполнить требуемые действия по изменению арифметической операции (и, кроме того, «не пропустить» дальше те символы, которые не имеет смысла использовать в арифметических операндах). Напомним, что для прекращения последующих вызовов обработчиков данного события необходимо пометить событие как обработанное, положив свойство e.Handled равным true. Следует сказать, что пометка события как обработанного позволяет отменить вызов и *стандартных* обработчиков событий, связанных с компонентами (в нашем случае был отменен вызов стандартного обработчика события TextInput, обеспечивающего добавление набранного на клавиатуре символа в поле ввода).

2. Для имитации возникновения события, связанного с нажатием кнопок 1–4, в методе Window_PreviewTextInput вызывается обработчик данного события button1_Click. При этом необходимо указать нужную кнопку. Проще всего передать ее в качестве первого параметра обработчика, однако такой подход требует корректировки действий, содержащихся в методе button1_Click.

В операторе, добавленном в метод button1_Click, делается попытка привести параметр sender к типу Button. Если эта попытка успешна, то соответствующая кнопка помещается в переменную s. Если же указанное преобразование нельзя выполнить, то операция as возвращает значение null. Это означает, что обработчик был вызван родительским компонентом, а «истинный» адресат события содержится в свойстве e.Source, которое в этом случае приводится к типу Button и сохраняется в переменной s. Все описанные действия удалось реализовать в единственном операторе благодаря операции a ?? b, которая возвращает значение a, если оно не равно null, и b в противном случае.

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;
}
```

Комментарий

При нажатии пробела возникают только события KeyDown и KeyUp (и связанные с ними события PreviewKeyDown и PreviewKeyUp), которые реагируют на нажатие *любых* клавиш, в том числе и не приводящих к генерации отображаемых символов. Мы перехватываем это событие на уровне родителя обоих полей ввода, поэтому оно не доходит до них и пробелы в полях ввода не отображаются.

Заметим, что перехватывать событие на более высоком уровне (на уровне вертикальной панели StackPanel или на уровне окна), не следует, так как в этом случае нажатие пробела не дойдет и до других компонентов окна, в частности, до кнопок. В результате станет недоступной возможность нажать кнопку, выделив ее и нажав клавишу пробела.

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]
```

Комментарий

Статическое свойство CurrentCulture класса CultureInfo, определенного в пространстве имен System.Globalization, позволяет получить инфор-

мацию о *региональных настройках*, используемых операционной системой, в частности о *числовых форматах*. Необходимость в указании индекса [0] обусловлена тем, что свойство NumberDecimalSeparator имеет *строковый* тип, который не совместим по присваиванию с символьным типом. Заметим, что свойство NumberDecimalSeparator доступно только для чтения, однако имеется возможность изменить региональные настройки *в целом* для конкретного приложения (см. по этому поводу комментарий в проекте CLOCK, п. 4.1).

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.

Комментарии

1. Здесь мы использовали более традиционный способ связывания одного обработчика события с несколькими компонентами – путем указания этого обработчика в xaml-файле в описании *каждого* компонента.

Впрочем, в данном случае тоже можно было воспользоваться механизмом маршрутизируемых событий и после создания обработчика textBox1_TextChanged для компонента textBox1 не копировать соответствующий атрибут в компонент textBox2, а *переместить* его в родительский компонент StackPanel, снабдив префиксом TextBox:

```
<StackPanel Margin="10,10,10,0" …
```

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

2. Указание обработчиков событий, подобных событию TextChanged, непосредственно в xaml-файле может приводить к неожиданным ошибкам. Например, если закомментировать условный оператор в методе textBox1_TextChanged

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

то при запуске программы возникнет исключение NullReferenceException («Ссылка на объект не указывает на экземпляр объекта»). Это связано с тем, что в WPF событие TextChanged возникает *сразу* после конструирования поля ввода (при присваивания свойству Text начального значения). Но в момент создания поля ввода textBox1 метка label2 еще не существует, поскольку компоненты создаются в порядке их указания в xaml-файле, что и приводит к возникновению исключения.

Мы избежали этой ошибки, добавив проверку в обработчик. Исправить подобную ошибку можно и другим способом: не добавляя проверку в метод textBox1_TextChanged, *удалить оба атрибута* TextChanged="textBox1_TextChanged" в xaml-файле и вместо этого добавить *в конец* конструктора MainWindow операторы

```
textBox1.TextChanged += textBox1_TextChanged;
textBox2.TextChanged += textBox1_TextChanged;
```

Благодаря этим операторам связывание события TextChanged с обработчиками будет происходить уже *после* создания всех компонентов окна, и попытки обращения к неинициализированной метке не произойдет.

При анализе данного исправления возникает естественный вопрос: можно ли *в программном коде* связать требуемый обработчик с общим родителем обоих полей ввода – панелью StackPanel (подобно тому, как это делается в xaml-файле – см. комментарий 1)? Этому препятствуют два обстоятельства: во-первых, данная панель не имеет имени, с помощью которого к ней можно было бы обратиться в коде, и, во-вторых, в компоненте StackPanel отсутствует событие TextChanged. Первую проблему легко решить, добавив к описанию панели в xaml-файле атрибут x:Name. Вторая проблема решается благодаря наличию у любого компонента метода AddHandler, позволяющего связать с компонентом обработчик события даже в случае, если это событие для компонента не предусмотрено. Между прочим, первую из отмеченных проблем можно вообще не решать, если связать обработчик с родителем более высокого уровня – окном MainWindow. Таким образом, вместо указанных выше двух операторов достаточно добавить в конструктор окна следующий вызов метода AddHandler:

```
    AddHandler(TextBox.TextChangedEvent,
      new TextChangedEventHandler(textBox1_TextChanged));
```

Обратите внимание на необходимость указания суффикса Event в первом параметре метода AddHandler и на более сложный способ определения второго параметра, требующий вызова конструктора класса, связанного с данным типом обработчиков событий.

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 (рис. 15).

Рис. 15. Окно приложения CLOCK (первый вариант)

Комментарии

1. Для работы с датами и временем в библиотеке .NET предусмотрена структура DateTime. Ее статическое свойство Now, доступное только для чтения, возвращает *текущую дату и время* (по системным часам компьютера). Текущую дату без времени (время соответствует полуночи) можно получить с помощью статического свойства Today. Для преобразования даты/времени к их стандартным строковым представлениям можно использовать следующие методы структуры DateTime:

- ToShortDateString – дата в кратком формате «d», например «27.01.1756»;
- ToLongDateString – дата в полном формате «D», например «27 января 1756 г.»);
- ToShortTimeString – время в кратком формате «t», например «10:55»;
- ToLongTimeString – время в полном формате «T», например «10:55:15».

Метод ToString без параметров возвращает дату/время в формате «G» (дата в кратком формате, время в полном). Формат отображения даты/времени можно явно указать в методе ToString; например, в нашей программе можно было бы использовать такой вариант: DateTime.Now.ToString("T").

Упомянем еще некоторые форматы для даты/времени: «g» – дата и время в кратком формате, «F» – дата и время в полном формате, «f» – дата в полном формате, время в кратком, «M» или «m» – формат «месяц, день», «Y» или «y» – формат «месяц, год».

При форматировании дат используются текущие *региональные настройки* (в нашем случае – настройки для России), хотя имеется перегруженный вариант метода ToString, где можно явно указать требуемую региональную настройку. Можно также сменить региональную настройку для приложения в целом; для этого достаточно установить новое значение свойства CurrentCulture для объекта Thread.CurrentThread из пространства имен System.Threads. Например, для того чтобы установить для нашего приложения региональные настройки, соответствующие американскому варианту английского языка, достаточно добавить в конструктор следующий оператор:

```
System.Threading.Thread.CurrentThread.CurrentCulture =
  new System.Globalization.CultureInfo("en-US");
```

При этом вариант отображения текущего времени в окне изменится (рис. 16).

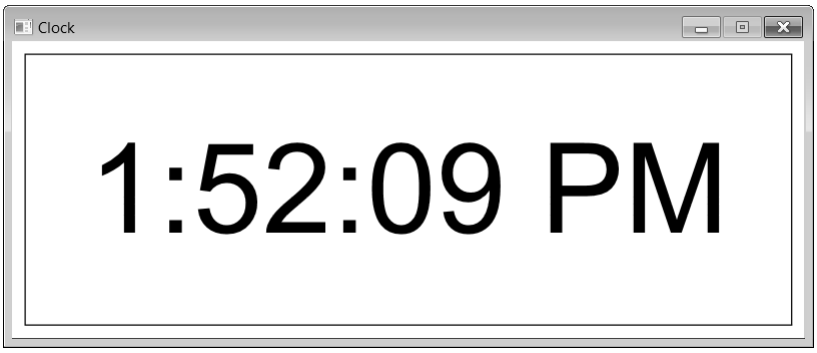


Рис. 16. Окно приложения CLOCK с измененными региональными настройками

Заметим, что настройки для России имеют имя «ru-RU».

2. В отличие от библиотеки Windows Forms, где предусмотрен специальный *невизуальный компонент* Timer, в библиотеке WPF приходится использовать «обычный» объект типа DispatcherTimer (из пространства имен System.Windows.Threading), явным образом задавая в программе все его свойства и события. Обратите внимание на то, что свойство Interval (время между срабатываниями таймера) имеет тип TimeSpan (этот тип подробно описывается в последнем комментарии к следующему пункту).

3. Особенностью макета данного приложения является использование *рамки* Border вокруг метки с текстом текущего времени. Для отображения времени мы использовали специальную текстовую метку TextBlock, содержимым которой (в отличие от «обычной» метки Label) может быть только текст. В качестве имени для этого компонента мы выбрали label1 как более кратное и наглядное по сравнению с textBlock1.

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();
}
```

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 (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).

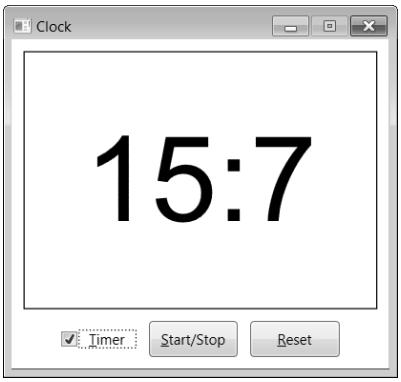


Рис. 17. Окно приложения CLOCK в режиме секундомера

Комментарии

1. Свойство IsChecked компонента CheckBox имеет в WPF тип bool?, т. е. может принимать три значения: true, false и null (последний вариант используется для флажков с тремя состояниями), поэтому в условии оператора if приходится выполнять приведение свойства checkBox1.IsChecked к типу bool (вместо этого можно использовать сравнение checkBox1.IsChecked == true).

2. Для формирования текста метки в режиме секундомера используется метод Format класса string, возвращающий строку, которая содержит фиксированные фрагменты и строковые представления различных объектов, отформатированные требуемым образом. Первым параметром данного метода является *форматная строка*, содержащая как обычный текст, так и *форматные настройки* для остальных параметров (количество подобных форматируемых параметров может быть произвольным). Форматные настройки заключаются в фигурные скобки {}; в нашем случае использованы простейшие форматные настройки, в которых задается только порядковый номер параметра, выводимого в указанной позиции форматной строки (в подобной простейшей ситуации для форматирования данного параметра автоматически вызывается его метод ToString). Параметры нумеруются от 0.

В версии C# 6.0, используемой в Visual Studio 2015, для формирования строк с различными «внешними» параметрами вместо метода Format удобнее применять так называемые *интерполированные строки*. В интерполированной строке перед открывающей ее двойной кавычкой указывается символ $; а параметры задаются в ней в фигурных скобках. С использованием интерполированной строки оператор задания текста метки в режиме секундомера можно представить в следующем виде:

```
label1.Text = $"{t / 10}:{t % 10}";
```

Таким образом, интерполированная строка представляет собой форматную строку метода Format, в которой вместо порядкового номера выводимого параметра указывается сам этот параметр.

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), после 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 программу 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();
}
```

Комментарий

Структура TimeSpan предназначена для хранения *относительных* промежутков времени. Промежутки времени измеряются в днях, часах, минутах, секундах и миллисекундах, причем для получения значения каждого из этих компонентов можно использовать соответствующие свойства структуры TimeSpan: Day, Hour, Minute, Second, Millisecond (заметим, что эти же свойства имеются и у структуры DateTime; кроме того, у структуры DateTime есть свойства Year и Month). Для задания нулевого промежутка времени проще всего воспользоваться полем TimeSpan.Zero, доступным только для чтения. И структура DateTime, и структура TimeSpan имеют также поля для чтения, определяющие их наименьшие и наибольшие возможные значения – MinValue и MaxValue.

Операции сложения и вычитания определяются для структур DateTime и TimeSpan следующим образом: сумма или разность значений типа TimeSpan имеет тип TimeSpan; сумма или разность значений типа DateTime и TimeSpan (*в указанном порядке*) имеет тип DateTime; разность значений типа DateTime имеет тип TimeSpan; складывать значения типа DateTime нельзя. Поскольку для промежутков времени допускаются отрицательные значения, для структуры TimeSpan определена также операция «унарный минус».

Для создания объектов типа DateTime и TimeSpan с требуемыми значениями проще всего воспользоваться одним из предусмотренных для них конструкторов. Конструктор без параметров возвращает для DateTime минимальную дату (полночь 1 января 1 года н. э.), а для TimeSpan – нулевой промежуток времени. В остальных конструкторах DateTime необходимо указывать год, месяц, день (и можно указать дополнительно время в часах, минутах и секундах, возможно, дополненное параметром со значением миллисекунд). В конструкторах TimeSpan необходимо указывать час, минуту и секунду. В качестве дополнительного *начального* параметра можно указать число дней. Если указано число дней, то можно указать дополнительный *последний* параметр – количество миллисекунд.

О других методах и свойствах структур DateTime и TimeSpan можно прочесть, например, в [4, гл. 2] и [5, гл. 6].

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.

Комментарии

1. «Объединить» в одном обработчике действия при однократном и двойном щелчке мышью нам удалось благодаря наличию свойства e.ClickCount. Интересно, что данное свойство может принимать значения, б*о*льшие 2, позволяя программе реагировать на *тройные* щелчки мышью.

2. При связывании некоторых действий с однократным и двойным щелчком мыши необходимо учитывать, что при выполнении двойного щелчка система *вначале* регистрирует *одинарный щелчок* (при котором наступает событие MouseDown с параметром e.ClickCount, равным 1, а затем событие MouseUp с параметром e.ClickCount, равным 1), и только потом, после второго нажатия кнопки мыши, *если промежуток времени между нажатиями был достаточно мал*, регистрируется двойной щелчок (при котором наступают события MouseDown с e.ClickCount = 2 и MouseUp с e.ClickCount = 2). Поэтому *необходимо, чтобы действие, выполняющееся при однократном щелчке, не конфликтовало с действием, которое связывается с двойным щелчком*. В нашей программе в этом отношении все обстоит благополучно: хотя в режиме секундомера предусмотрены действия и при одинарном, и при двойном щелчке, действие при одинарном щелчке (запуск, или остановка, или сброс секундомера) никак не конфликтует с действием, связанным с двойным щелчком (т. е. с переходом в режим часов).

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.

Комментарии

1. Событие StateChanged происходит при изменении *состояния* окна, т. е. при его минимизации, максимизации и возврате в нормальное состояние. В этих ситуациях, как и при срабатывании таймера, наша программа изменяет текст на кнопке приложения (заметим, что до указанного исправления текст на кнопке изменялся только при очередном срабатывании таймера, и поэтому в режиме часов до этого события могла пройти целая секунда).

2. При определении обработчика для события окна StateChanged возникает соблазн связать данное событие с уже имеющимся обработчиком timer1_Tick, что позволило бы не дублировать программный код. К сожалению, подобный вариант исправления тоже неправильно работает в режиме остановленного секундомера (объясните, почему). Вместе с тем избежать дублирования текста все же можно, если, наоборот, в обработчике timer1_Tick вместо добавленного в данном пункте текста поместить вызов обработчика Window_StateChanged:

```
Window_StateChanged(null, null);
```

5. Поля ввода: TEXTBOXES

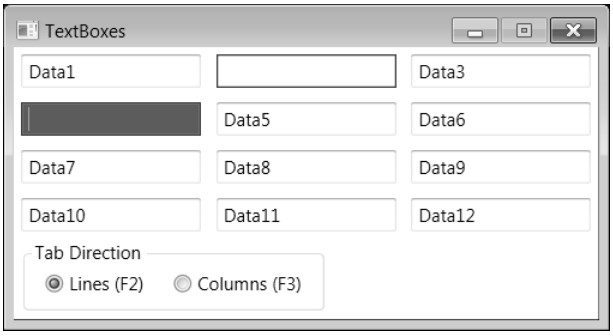


Рис. 18. Окно приложения TEXTBOXES

5.1. Дополнительное выделение активного поля ввода

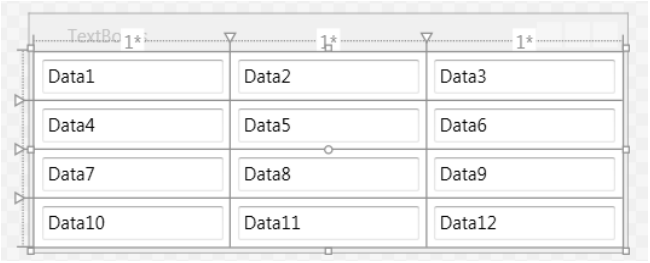


Рис. 19. Макет окна приложения TEXTBOXES (первый вариант)

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.

Комментарии

1. Перемещение фокуса обеспечивается щелчком мыши на компоненте или клавишами Tab и Shift+Tab. При использовании клавиш Tab и Shift+Tab порядок обхода тех компонентов, которые могут получать фокус, определяется значением их свойства TabIndex, которое по умолчанию полагается равным int.MaxValue (т. е. максимальному значению типа int). Если у некоторых компонентов значения свойств TabIndex совпадают, то порядок обхода соответствует порядку следования этих компонентов в xaml-файле. Для перемещения фокуса часто бывает достаточно использовать *клавиши со стрелками*, однако для компонентов TextBox это невозможно, так как в них нажатия на клавиши со стрелками обрабатываются особым образом. С обходом компонентов связано также свойство IsTabStop: если IsTabStop равно false, то при обходе с помощью клавиш Tab и Shift+Tab данный компонент пропускается. Еще одним свойством, связанным с фокусом, является Focusable: компонент может полу-

чать фокус (любым способом) только в случае, если это свойство равно true.

2. Благодаря связыванию созданных обработчиков с компонентом Grid – родительским компонентом всех полей ввода – мы смогли обеспечить вызов этих обработчиков для всех полей ввода при наступлении событий получения или потери фокуса. Обратите внимание на то, что при перемещении атрибутов GotFocus и LostFocus нам не пришлось добавлять к ним префикс TextBox. Это связано с тем, что данные события имеются и у компонента Grid (хотя они для него не наступают, так как значением по умолчанию для свойства Focusable компонента Grid является false).

3. Свойства Background и Foreground, задающие цвет фона и символов, имеют тип Brush, что позволяет указывать для них не только обычные цвета, но и более сложные цветовые настройки (например, градиентную заливку – см. проекты TRIGFUNC и HTOWERS). Исходные значения этих свойств для полей ввода сохраняем во вспомогательных переменных backgr и foregr. Важно, чтобы это сохранение было выполнено до того, как поле ввода textBox1 получит фокус, так как при получении фокуса запускается обработчик textBox1_GotFocus, изменяющий исходные значения цветов фона и символов.

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.

Комментарий

Для выделения фрагмента текста в поле ввода предусмотрен метод Select, имеющий два параметра, – позицию *начала выделения* (нумеруется от 0; значение 0 соответствует позиции *перед* первым символом) и *длину выделения*, т. е. количество выделенных символов. Если длина выделения равна 0, то позиция начала выделения определяет позицию каретки. Для начала выделения и его длины предусмотрены также свойства SelectionStart и SelectionLenght, которые доступны и для чтения, и для записи. В нашем случае вместо вызова метода Select было достаточно изменить свойство SelectionStart:

```
tb.SelectionStart = tb.Text.Length;
```

(свойство SelectionLength изменять не требуется).

Если в поле ввода желательно выделять весь текст при получении им фокуса, то для этого можно использовать следующий оператор:

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

Отметим также свойство SelectedText, которое позволяет получить и *изменить* выделенный текст: присваивание данному свойству новой строки приводит к тому, что выделенный фрагмент заменяется указанной строкой (если поле ввода не содержало выделения, то указанная строка *вставляется* в позицию каретки), причем для вставленного фрагмента сохраняется выделение. Если присвоить свойству SelectedText пустую строку, то выделенный в поле ввода фрагмент будет удален.

5.2. Управление порядком обхода полей на форме

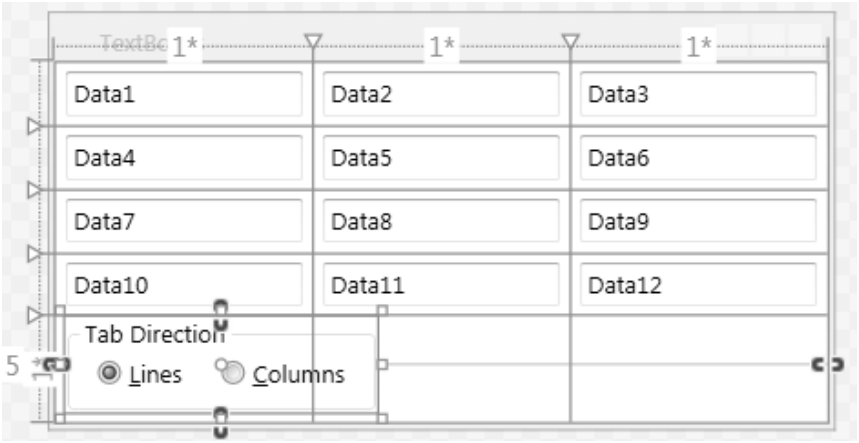


Рис. 20. Макет окна приложения TEXTBOXES (второй вариант)

```
<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 свой-

ству для «пустой» переменной tb возбуждается исключение NullReference-Exception.

Исправление. Дополните методы textBox1_GotFocus и textBox1_Lost-Focus:

```
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;
}
```

Комментарии

1. Для перебора дочерних компонентов группирующего компонента Grid мы использовали его свойство-коллекцию Children (это свойство имеется у любого группирующего компонента).

В обработчике radioButton1_Checked дочерние компоненты перебираются в цикле foreach; проверка tb != null позволяет отбросить дочерний компонент GroupBox, не являющийся полем ввода. Как было отмечено в комментарии 1 предыдущего пункта, для перебора компонентов в исходном порядке достаточно установить для них *одинаковые* значения свойства TabIndex.

В обработчике radioButton2_Checked дочерние компоненты перебираются в цикле for, поскольку в данном случае при вычислении новых значений свойств TabIndex используется *порядковый номер* компонентов. В этом обработчике для отбрасывания «лишнего» компонента типа GroupBox достаточно уменьшить на 1 число итераций цикла, так как лишний компонент является *последним* элементом коллекции Children. Обратите внимание на то, что радиокнопки не входят в коллекцию Children компонента Grid, так как их непосредственным родителем является группирующий компонент StackPanel, который, в свою очередь, яв-

ляется *содержимым* компонента GroupBox (компонент GroupBox, несмотря на свое название, *не является* группирующим компонентом).

Формула, использованная в обработчике radioButton2_Checked, позволяет получить следующее распределение значений TabIndex для различных полей ввода (обратите внимание на то, что значения TabIndex здесь увеличиваются *по столбцам*):

```
 0*1   1*10   2*100
 3*1   4*10   5*100
 6*1   7*10   8*100
 9*1  10*10  11*100
```

2. Возможность выбора радиокнопки с помощью клавиатурных комбинаций обеспечивается благодаря указанию символов «_» в начале подписей к радиокнопкам (ранее эту возможность мы использовали в п. 3.3 проекта CALC).

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.

Комментарии

1. Необходимость в дополнительном изменении толщины рамки связана с тем, что при исходной толщине, в случае активного поля ввода, зеленый фон «заслоняет» красную рамку. Небольшое увеличение толщины позволяет решить эту проблему. Толщина представлена особым классом Thickness; благодаря этому с помощью варианта конструктора класса Thickness с *четырьмя* параметрами можно установить разную толщину для левой, верхней, правой и нижней части рамки. Следует заметить, что свойства Margin и Padding также имеют тип Thickness.

2. В библиотеке WPF имеется специальный механизм проверки правильности (validation) введенных данных (см., например, [8, гл. 19]), однако он основан на привязке данных и требует создания дополнительных объектов. Описанный в данном пункте вариант организации проверки достоверности предоставляет функциональность, аналогичную функциональности компонента ErrorProvider из библиотеки Windows Forms (отсутствующего в WPF) и требует минимального объема программного кода. Если для некоторых полей требуется проверка достоверности специального вида (например, проверка возможности преобразования введенного текста в число), то можно пометить такие поля с помощью некоторого значения свойства Tag и дополнить обработчик события TextChanged требуемыми вариантами проверки, выполняемыми для полей с соответствующими значениями свойства Tag (использование свойства Tag рассматривается в проекте ZOO, п. 7.3).

3. Мы связали созданный обработчик с родительским компонентом grid1 не в xaml-файле, а в конструкторе окна с помощью метода AddHandler. Это необходимо для того, чтобы *первое* наступление события TextChanged (которое произойдет сразу после создания каждого поля ввода при начальном изменении его свойства Text) *не привело* к вызову обработчика textBox1_TextChanged. В противном случае свойствам BorderBrush и BorderThickness будут присвоены значения null, так как поля bordbr и bordth определяются в конструкторе окна уже *после* создания полей ввода (которые, как и все компоненты окна, определяемые в xaml-файле, создаются в его конструкторе при вызове метода InitializeComponent).

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.

Комментарии

1. В обработчике Window_Closing мы не выполняем отбор тех дочерних компонентов группирующего компонента Grid, которые являются полями ввода (подобный тому, который использовался в обработчике radioButton1_Checked из п. 5.2). Вместо этого все компоненты приводятся к типу Control – их общему предку, у которого уже имеется свойство BorderBrush.

2. Используя реализованную в .NET, начиная с версии 3.5, технологию LINQ (Language Integrated Query – технология *запросов, интегрированных в язык программирования*, – см. [4, гл. 10], [5, гл. 8]), тело данного обработчика можно реализовать в виде *единственного* оператора:

```
e.Cancel = grid1.Children.OfType<TextBox>()
    .Any(e1 => e1.BorderBrush == Brushes.Red);
```

В этом операторе к коллекции Children компонента grid1 применяется запрос OfType, отбирающий в ней только те компоненты, которые могут быть приведены к типу TextBox, после чего к полученной коллекции компонентов TextBox применяется запрос Any, возвращающий значение true, если хотя бы для одного элемента этой коллекции e1 выполняется условие e1.BorderBrush == Brushes.Red.

6. Обработка событий от мыши: MOUSE

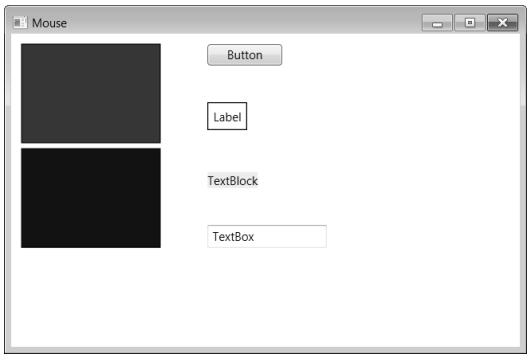


Рис. 21. Окно приложения MOUSE

6.1. Перетаскивание панели с помощью мыши

Рис. 22. 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

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

Комментарии

1. Напомним, что компонент Canvas будет реагировать на события, связанные с мышью, только в том случае, если он имеет непустой фон (т. е. его свойство Background не должно быть равно null). Поэтому мы явно настроили в xaml-файле белый фон для данного компонента. В случае *прозрачного* фона (Transparent) компонент также будет реагировать на события мыши.

Заметим, что в WPF-приложениях можно отключить компонент (и все его дочерние компоненты) от любых событий, связанных с мышью, положив его свойство IsHitTestVisible равным false (подобным образом, с помощью свойства Focusable, можно отключить компонент от событий, связанных с клавиатурой, – см. проект TEXTBOXES, п. 5.1).

2. Связывание созданных обработчиков с компонентом Canvas обеспечивает их вызов для всех его дочерних компонентов (при наступлении в них соответствующего события), а также *и для самого компонента Canvas*. По этой причине в начале каждого обработчика особым образом обрабатываем ситуацию, когда событие произошло в самом компоненте Canvas.

3. Для формирования заголовка окна при перетаскивании прямоугольника используется метод Format класса string, подробно описанный в проекте CLOCK (см. комментарий 2 в п. 4.2). Обратите внимание на то, что в строковом представлении объекта типа Point автоматически добавляется разделитель «;» между координатами, например «80,4;59,6».

4. Кажущееся постоянство значений координат, отображаемых на экране при перетаскивании прямоугольника, объясняется тем, что немедленно, после изменения координат, происходит корректировка положения прямоугольника в окне, а вместе с тем и пересчет локальных координат мыши относительно прямоугольника. Если перемещать прямоугольник достаточно быстро, то можно заметить, что в заголовке окна на короткое время отобразятся другие значения координат.

5. При вычислении нового положения прямоугольника к его прежней позиции (определяемой с помощью свойств Left и Top, полученных от родителя Canvas) прибавляется *смещение*, вычисленное по текущему значению q позиции курсора мыши и значению p той позиции курсора, которая была сохранена в момент нажатия кнопки мыши.

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 пе-

рехватывает 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);
```

Комментарий

Характеристика, определяющая, в каком порядке перекрывающиеся компоненты будут отображаться в окне, называется *Z-порядком*, поскольку ее можно интерпретировать как координату компонента на оси Z, ориентированной перпендикулярно плоскости экрана. В библиотеке Windows Forms для управления Z-порядком имелись методы BringToFront и SendToBack. В WPF такие методы отсутствуют, однако любой группирующий компонент имеет присоединенное свойство ZIndex, которое, подобно свойствам Left и Top компонента Canvas, «передается» всем дочерним компонентам этого группового компонента (для получения свойства используется статический метод GetZIndex, а для изменения – метод SetZIndex). Дочерний компонент, у которого свойство ZIndex *больше*, располагается на экране *над* компонентами с меньшими значениями свойства ZIndex, т. е. «ближе» к пользователю (таким образом, если считать, что ZIndex является координатой компонента на оси Z, то эта ось Z направлена к пользователю). Если же значения свойства ZIndex у компонентов одинаковые, то их Z-порядок определяется очередностью добавления к групповому компоненту (тот дочерний компонент, который добавлен позже всех, располагается над всеми). Дочерние компоненты, определяемые в макете окна, добавляются к групповому компоненту в том порядке, в котором они указываются в xaml-файле. По умолчанию свойства ZIndex всех дочерних компонентов полагаются равными 0.

Теперь должен быть понятен смысл сделанных изменений. В поле maxz хранится *максимальный* Z-индекс компонентов (вначале он равен 0). При щелчке на некотором компоненте поле maxz увеличивается на 1 и его новое значение определяет Z-индекс данного компонента. Тем самым этот компонент поднимается над всеми остальными дочерними компонентами. Итак, с помощью поля maxz мы реализовали функциональность отсутствующего метода BringToFront.

Аналогичным образом можно реализовать функциональность метода SendToBack: для этого достаточно описать поле minz и присваивать значение --minz свойству ZIndex того компонента, который требуется расположить *под* всеми остальными.

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»). В данной ситуации мышь захватывается *компонентом Canvas* (поскольку для него тоже вызывается обработчик rect1_MouseDown), поэтому при перемещении мыши над прямоугольником событие MouseMove передается не прямоугольнику, а захватившему мышь компоненту Canvas. Стоит в этой ситуации отпустить мышь над прямоугольником, как любое ее последующее перемещение будет перехвачено этим прямоугольником, что проявится в изменении заголовка окна.

Комментарий

Чтобы обеспечить захват мыши (и ее последующее освобождение) для *любого* компонента, для которого возникло событие MouseDown (или, соответственно, MouseUp), мы выполнили приведение свойства e.Source к типу FrameworkElement – общему предку *всех* компонентов, в том числе и групповых.

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).

Комментарии

1. При перетаскивании компонентов за правую или нижнюю границу окна проблем не возникает, так как для доступа к этим компонентам достаточно *увеличить размеры самого окна*.

2. Ограничить размеры компонента можно с помощью его свойств MinWidth, MinHeight, MaxWidth и MaxHeight. Если, например, указать в xaml-файле для компонента rect1 атрибуты MinWidth="50" и MinHeight="50", то размеры первого прямоугольника не удастся сделать меньшими 50 × 50. Однако такой способ требует настройки свойств для *каждого* компонента, размеры которого можно изменять.

3. Функция Max, возвращающая максимальный из двух своих параметров числового типа, является статическим (т. е. классовым) методом класса Math. Все прочие стандартные математические функции также реализованы в виде статических методов класса Math.

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.

Комментарий

Свойство Cursor, имеющееся у любого визуального компонента, а также класс Cursors будут подробно рассмотрены в проекте CURSORS. Отметим лишь, что если свойство Cursor равно null, то компонент будет использовать курсор своего родителя.

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 колеси-

ком, которое также можно нажимать): нажатие и отпускание этой кнопки никак не влияет на текущий режим программы.

6.5. Перетаскивание компонентов любого типа

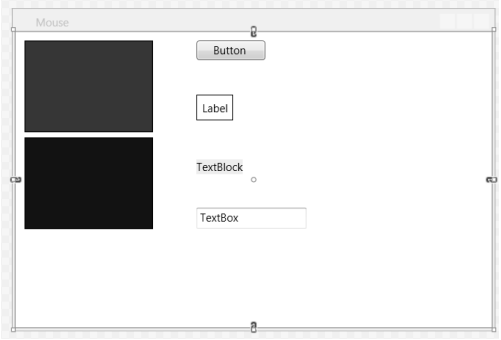


Рис. 23. Макет окна приложения MOUSE (второй вариант)

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).

Комментарий

Методы rect1_MouseDown, rect1_MouseMove и rect1_MouseUp позволяют единообразно обрабатывать компоненты различного типа благодаря приведению свойства e.Source к типу FrameworkElement – *общему предку всех визуальных компонентов*. Заметим, что в этих методах использованы только те свойства и методы, которые имеются у класса FrameworkElement (иначе произошла бы ошибка компиляции). Обработчики событий не потребовалось явным образом присоединять к новым

компонентам, поскольку эти обработчики уже связаны с *родителем* Canvas всех добавленных компонентов.

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 установить поле ввода TextBox у левой границы окна и продолжить сдвигать 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.

Комментарий

Присваивание свойству Handled значения true означает, что данное событие *обработано* (см. проект CALC, п. 3.2). В частности, если это действие выполняется в обработчиках событий PreviewMouseDown и PreviewMouseUp, то события MouseDown и MouseUp не возникают.

7. Перетаскивание (Drag & Drop): ZOO

Рис. 24. Окно приложения ZOO

7.1. Перетаскивание меток в окне

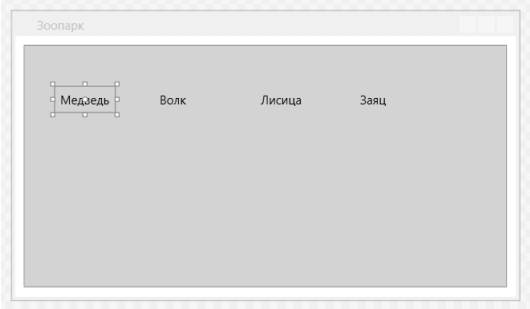


Рис. 25. Макет окна приложения ZOO (первый вариант)

```
<Window x:Class="ZOO.MainWindow"
    …
    Title="Зоопарк" SizeToContent="WidthAndHeight"
    WindowStartupLocation="CenterScreen"
    ResizeMode="CanMinimize" >
  <StackPanel >
    <Canvas x:Name="canvas1" Margin="10" Width="480" Height="240"
        Background="LightGray" AllowDrop="True"
        Drop="canvas1_Drop" >
      <TextBlock x:Name="label1" Canvas.Left="30" Text="Медведь"
          Canvas.Top="41" Padding="5" />
      <TextBlock x:Name="label2" Canvas.Left="130" Text="Волк"
          Canvas.Top="41" Padding="5" />
      <TextBlock x:Name="label3" Canvas.Left="230" Text="Лисица"
          Canvas.Top="41" Padding="5" />
      <TextBlock x:Name="label4" Canvas.Left="330" Text="Заяц"
          Canvas.Top="41" Padding="5" />
```

```
    </Canvas>
  </StackPanel>
</Window>
```

```
private void canvas1_Drop(object sender, DragEventArgs e)
{
    if (!(e.Source is Canvas))
        return;
    TextBlock src = e.Data.GetData(typeof(TextBlock))
      as TextBlock;
    Point p = e.GetPosition(canvas1);
    Canvas.SetLeft(src, p.X - src.ActualWidth / 2);
    Canvas.SetTop(src, p.Y - src.ActualHeight / 2);
}
```

Для компонента label1 создайте обработчик события MouseDown, после чего переместите соответствующий атрибут в компонент Canvas:

```
<Canvas x:Name="canvas1" … MouseDown="label1_MouseDown" >
  <TextBlock x:Name="label1" … MouseDown="label1_MouseDown" />
```

```
private void label1_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    TextBlock t = e.Source as TextBlock;
    if (t == null)
        return;
    if (e.ChangedButton == MouseButton.Left)
        DragDrop.DoDragDrop(t, t, DragDropEffects.Move);
}
```

Результат. Метки с названиями зверей можно перетаскивать с помощью левой кнопки мыши по компоненту Canvas. В процессе перетаскивания метки-«источника» (source) она остается на месте, однако вид курсора мыши изменяется, что является признаком режима перетаскивания. В качестве «приемника» (target) пока определен лишь компонент Canvas: при отпускании над ним кнопки мыши происходит перемещение метки зверя на указанную позицию.

В дальнейшем в текстах обработчиков, связанных с перетаскиванием, будем использовать имена src и trg для объектов-*источников* и объектов-*приемников* соответственно.

Комментарии

1. В данном комментарии дается обзор основных возможностей режима Drag & Drop. Средства для реализации этого режима имеются и в библиотеке Windows Forms, и в библиотеке WPF, причем, несмотря

на многие общие черты, в них есть ряд важных отличий, которые необходимо учитывать при программировании режима Drag & Drop в приложениях Windows Forms и WPF. В дальнейшем мы будем особо оговаривать важнейшие особенности, описывая в основном тексте вариант для WPF и указывая в скобках вариант для Windows Forms.

Для запуска режима перетаскивания в WPF предусмотрен статический метод DoDragDrop класса DragDrop (*в библиотеке Windows Forms* одноименный метод имеется у любого компонента). Первый параметр метода указывает тот объект, который инициирует перетаскивание (обычно это какой-либо компонент), второй параметр определяет *источник* перетаскивания, т. е. тот объект, который будет приниматься *приемником* перетаскивания (в нашем случае объект, инициирующий перетаскивание, совпадает с объектом-источником), а третий параметр типа DragDropEffects определяет разрешенный результат (*эффект*) успешного перетаскивания (*в библиотеке Windows Forms* метод DoDragDrop имеет не три, а два параметра – источник перетаскивания и эффект). Предусмотрено три основных эффекта – Copy, Move и Link; их названия связаны с вариантами действий при перетаскивании *файлов*. Если в ходе перетаскивания может возникнуть один из *нескольких* эффектов (например, Copy *или* Move), то в качестве последнего параметра надо указать все эти эффекты, объединяя их операцией | («побитовое ИЛИ»), например: DragDropEffects.Copy | DragDropEffects.Move. Имеется также член перечисления DragDropEffects.All, объединяющий в WPF эффекты Move и Copy (*в библиотеке Windows Forms* вариант All объединяет все три основных эффекта).

Для того чтобы компонент мог выступать в роли приемника, необходимо установить значение его свойства AllowDrop равным true. Только в этом случае компонент сможет реагировать на события, связанные с перетаскиванием (DragEnter, DragOver, DragLeave и Drop). В WPF свойство AllowDrop, установленное в true для компонента-родителя, распространяет свое действие на все дочерние компоненты; таким образом, в WPF-приложениях надо явно задавать свойство AllowDrop равным false для тех дочерних компонентов приемника-родителя, которые *не должны* выступать в роли приемника (*в библиотеке Windows Forms* ситуация иная: для *каждого* компонента, который должен выступать в роли приемника, свойство AllowDrop должно быть явно установлено в true; компоненты с AllowDrop, равным false, считаются «невидимыми» для режима Drag & Drop).

Событие DragEnter возникает при *появлении* над компонентом источника перетаскивания, а событие DragOver – при последующем *перемещении* источника над компонентом. В обработчиках этих событий компонент-приемник указывает, может ли он принять источник, присвоив свой-

ству e.Effects значение соответствующего эффекта. При этом приемник может определить, какие эффекты разрешены источником, обратившись к свойству e.AllowedEffects, доступному только для чтения (*в библиотеке Windows Forms* аналогичные свойства имеют имена Effect и AllowedEffect). Приемник может также установить свойство e.Effects равным None; это будет означать, что он отказывается принимать источник перетаскивания. Эффект, выбранный приемником, определяет вид его курсора в режиме перетаскивания; в частности, если приемник выбрал эффект None, то его курсор будет иметь вид запрещающего знака (обычно это перечеркнутая окружность). Если обработчики DragEnter и DragOver не указаны, то приемник будет принимать источник с любым эффектом (*в библиотеке Windows Forms* при отсутствии этих обработчиков приемник не будет принимать перетаскиваемые объекты).

В WPF следует определять обработчики для обоих событий (и DragEnter, и DragOver), причем если при перемещении источника над приемником доступность приемника не может измениться (как в нашей программе, а также в большинстве других программ), то можно определить *один* обработчик, связав его с обоими событиями (*в библиотеке Windows Forms* при наличии обработчика события DragEnter можно не связывать его с событием DragOver, если эти события должны обрабатываться одинаково).

В WPF в обработчиках событий DragEnter и DragOver необходимо указывать оператор e.Handled = true, иначе значение, присвоенное свойству e.Effects, может игнорироваться.

Событие Drop возникает при *отпускании* источника над компонентом-«приемником», причем только в том случае, когда компонент *может* принять источник (*в библиотеке Windows Forms* аналогичное событие имеет имя DragDrop).

В любом из перечисленных событий можно получить доступ к объекту-источнику с помощью метода GetData объекта e.Data, указав в качестве параметра *формат* источника (выражение типа Type или string). Результат, возвращаемый методом GetData, имеет тип object, поэтому его требуется явным образом преобразовать к фактическому типу источника. В ситуации, когда тип источника заранее не известен, приемник может запросить соответствующую информацию, используя метод GetFormats объекта e.Data (без параметров), который возвращает *массив* строк – имен форматов данных, имеющихся в источнике (поскольку источник перетаскивания может предоставлять данные в *нескольких* форматах). В нашем случае вызов метода GetFormats вернул бы массив из одного элемента – строки «System.Windows.Controls.TextBlock», т. е. *полного имени типа* источника. Заметим, что это имя можно указать в качестве параметра метода GetData (вместо typeof(TextBlock)).

2. Для перемещения метки на новое место необходимо знать позицию курсора мыши при завершении режима перетаскивания. Для получения данной позиции следует использовать метод e.GetPosition (напомним, что такой же метод доступен в любых обработчиках, связанных с событиями мыши).

Недочет 1. Если перетащить одну метку на другую и отпустить кнопку мыши, то ничего не произойдет. В подобной ситуации (при перемещении источника над недопустимым приемником) желательно, чтобы курсор имел вид запрещающего знака.

Исправление. Для компонента canvas1 создайте обработчик события DragEnter и укажите *этот же обработчик* для события DragOver:

```
<Canvas x:Name="canvas1" … DragEnter="canvas1_DragEnter"
    DragOver="canvas1_DragEnter" >
```

```
private void canvas1_DragEnter(object sender, DragEventArgs e)
{
    if (e.Source is Canvas)
        return;
    e.Handled = true;
    e.Effects = DragDropEffects.None;
}
```

Результат. Теперь при перетаскивании одной метки на другую курсор мыши принимает вид запрещающего знака.

Комментарии

1. Обработчик label1_DragEnter будет вызываться как для канвы Canvas, так и для любых ее дочерних компонентов-меток. Собственно, мы и создали этот обработчик для того, чтобы обрабатывать соответствующие события для меток. Если же данный обработчик вызывается для самого компонента Canvas, то он не должен выполнять никаких действий (в противном случае, если убрать указанный в начале обработчика условный оператор, то курсор перетаскивания будет *всегда* иметь запрещающий знак – и над метками, и над канвой Canvas). Имя свойства e.Source не должно вводить в заблуждение: оно не имеет никакого отношения к источнику перетаскивания (слово Source в данном случае означает *источник возникшего события*). С точки зрения механизма Drag & Drop тот объект, который содержится в свойстве e.Source, является *приемником* перетаскивания.

2. Если закомментировать оператор e.Handled = true, то курсор не будет изменяться на запрещающий. Таким образом, как и было отмечено ранее, этот оператор необходим для корректной работы обработчиков событий DragEnter и DragOver.

3. Если удалить в xaml-файле атрибут DragOver="canvas1_DragEnter", то при перетаскивании метки на другую метку курсор на долю секунды изменится на запрещающий, но затем опять восстановит исходный вид, соответствующий эффекту Move. Таким образом, как и было отмечено ранее, обработчик label1_DragEnter необходимо связать с *двумя* событиями – DragEnter и DragOver.

Недочет 2. Теперь в начальный момент перетаскивания курсор принимает вид запрещающего знака. Это нежелательно, так как может ввести в заблуждение пользователя, который решит, что он сделал что-то не так.

Исправление. Откорректируйте последний оператор в методе canvas1_DragEnter:

```
private void canvas1_DragEnter(object sender, DragEventArgs e)
{
    if (e.Source is Canvas)
        return;
    e.Handled = true;
    e.Effects = e.Data.GetData(typeof(TextBlock)) == e.Source ?
        DragDropEffects.Move : DragDropEffects.None;
}
```

Результат. В начальный момент перетаскивания курсор мыши остается разрешающим. Перетаскивание одной метки на другую по-прежнему запрещено.

Комментарий

При определении свойства e.Effects теперь проверяется, совпадают ли объект-источник (определяемый с помощью метода GetData) и объект-приемник (определяемый свойством e.Source). Обратите внимание на то, что в подобной ситуации не требуется приводить указанные объекты к типу TextBlock, так как для сравнения двух объектов на тождество не обязательно знать их фактический тип.

7.2. Перетаскивание меток в поля ввода

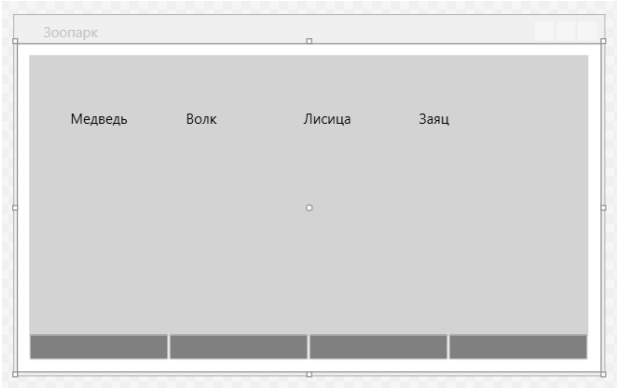


Рис. 26. Макет окна приложения ZOO (второй вариант)

```
<Window x:Class="ZOO.MainWindow"
    … >
  <StackPanel >
    <Canvas x:Name="canvas1" Margin="10,10,10,0" … >
      …
    </Canvas>
    <UniformGrid x:Name="grid1" Margin="10,0,10,10" Columns="4"
        AllowDrop="True">
      <TextBox Text="" IsReadOnly="True" Background="Gray" />
      <TextBox Text="" IsReadOnly="True" Background="Gray" />
      <TextBox Text="" IsReadOnly="True" Background="Gray" />
      <TextBox Text="" IsReadOnly="True" Background="Gray" />
    </UniformGrid>
  </StackPanel>
</Window>
```

Для компонента grid1 создайте обработчики событий PreviewDragEnter и Drop и задайте для события PreviewDragOver обработчик, уже созданный для события PreviewDragEnter:

```
<UniformGrid x:Name="grid1" …
    PreviewDragEnter="grid1 PreviewDragEnter"
    PreviewDragOver="grid1_PreviewDragEnter" Drop="grid1 Drop" >
```

```
private void grid1_PreviewDragEnter(object sender,
  DragEventArgs e)
{
    var trg = e.Source as TextBox;
    if (trg == null)
        return;
    e.Handled = true;
    e.Effects = trg.Text == "" ?
        DragDropEffects.Move : DragDropEffects.None;
}
private void grid1_Drop(object sender, DragEventArgs e)
{
    var trg = e.Source as TextBox;
    if (trg == null)
        return;
    var src = e.Data.GetData(typeof(TextBlock)) as TextBlock;
    trg.Text = src.Text;
    src.Visibility = Visibility.Hidden;
}
```

Результат. Теперь приемником может также служить любое *незаполненное* поле ввода («пустая клетка»). При перетаскивании метки на незаполненное поле ввода «зверь попадает в клетку» (текст метки отображается в поле ввода). Перетаскивание метки на уже заполненное поле ввода пока запрещено, хотя в следующем пункте это действие станет доступным.

Комментарии

1. Чтобы добавленный в поле ввода текст нельзя было редактировать, мы сделали все поля ввода доступными только для чтения, установив их свойство IsReadOnly равным true.

2. В макете использован группирующий компонент UniformGrid, который, в отличие от компонента Grid, всегда создает ячейки одинакового размера и размещает в них свои дочерние компоненты в порядке их указания в xaml-файле. Вместо определения строк и столбцов с помощью специальных элементов-свойств RowDefinition и ColumnDefinition (как в Grid) в UniformGrid достаточно указать атрибуты-свойства Rows и Columns (причем если один из этих атрибутов отсутствует, то размер таблицы подбирается по количеству дочерних компонентов; например, если бы в нашем случае было указано значение Columns="2", то UniformGrid содержал бы две строки, а при Columns="1" – четыре).

3. Обратите внимание на то, что для проверки доступности клеток мы использовали события PreviewDragEnter и PreviewDragOver. Это обусловлено особенностями реализации режима Drag & Drop для полей ввода TextBox. У этих компонентов события DragEnter и DragOver уже связаны со *стандартными* обработчиками, которые позволяют перетаскивать в поля ввода *только текстовые данные*. Таким образом, *переопределить* обработчики событий DragEnter и DragOver мы не можем, однако можем определить обработчики событий PreviewDragEnter и PreviewDragOver, которые возникают раньше и, при наличии в них оператора e.Handled = true, *отменяют* выполнение стандартных обработчиков.

3. Может возникнуть вопрос, почему для свойства Visibility разработчики WPF определили новый тип вместо того, чтобы использовать обычный тип bool. Это связано с тем, что при скрытии компоненты могут вести себя по-разному, резервируя или освобождая ранее занятую область своего родителя – группирующего компонента. Если свойство Visibility равно Visibility.Collapsed, то компонент освобождает ранее занятую область, а если равно Visibility.Hidden, то область сохраняется за компонентом, хотя в данный момент он и не отображается на экране. В нашем случае указанные варианты скрытия будут приводить к одинаковым результатам, однако при использовании других панелей (например, StackPanel) результаты будут различными, так как в случае скрытия с вариантом

Visibility.Collapsed освобожденная область будет занята другим дочерним компонентом, что потребует перекомпоновки всего содержимого панели.

7.3. Взаимодействие меток при их перетаскивании друг на друга

```
<Window x:Class="ZOO.MainWindow"
    … >
  <StackPanel >
    <Canvas … >
      <TextBlock x:Name="label1" … Tag="4" />
      <TextBlock x:Name="label2" … Tag="3" />
      <TextBlock x:Name="label3" … Tag="2" />
      <TextBlock x:Name="label4" … Tag="1" />
    </Canvas>
    <UniformGrid … >
      <TextBox … Tag="0" />
      <TextBox … Tag="0" />
      <TextBox … Tag="0" />
      <TextBox … Tag="0" />
    </UniformGrid>
  </StackPanel>
</Window>
```

Измените имеющиеся обработчики canvas1_DragEnter, canvas1_Drop и grid1_Drop следующим образом:

```
private void canvas1_DragEnter(object sender, DragEventArgs e)
{
    if (e.Source is Canvas)
        return;
    e.Handled = true;
    e.Effects = e.Data.GetData(typeof(TextBlock)) == e.Source ?
      DragDropEffects.Move : DragDropEffects.None;
    e.Effects = DragDropEffects.Move;
}
private void canvas1_Drop(object sender, DragEventArgs e)
{
    if (!(e.Source is Canvas))
        return;
    if (e.Source is Canvas)
    {
        TextBlock src = e.Data.GetData(typeof(TextBlock))
          as TextBlock;
```

```
        Point p = e.GetPosition(canvas1);
        Canvas.SetLeft(src, p.X - src.ActualWidth / 2);
        Canvas.SetTop(src, p.Y - src.ActualHeight / 2);
    }
    else
    {
        var trg = e.Source as TextBlock;
        var src = e.Data.GetData(typeof(TextBlock)) as TextBlock;
        if ((src.Tag as string)[0] > (trg.Tag as string)[0])
        {
            Canvas.SetLeft(src, Canvas.GetLeft(trg));
            Canvas.SetTop(src, Canvas.GetTop(trg));
            trg.Visibility = Visibility.Hidden;
        }
        else
            src.Visibility = Visibility.Hidden;
    }
}
private void grid1_Drop(object sender, DragEventArgs e)
{
    var trg = e.Source as TextBox;
    if (trg == null)
        return;
    var src = e.Data.GetData(typeof(TextBlock)) as TextBlock;
    if ((src.Tag as string)[0] >= (trg.Tag as string)[0])
    {
        trg.Text = src.Text;
        trg.Tag = src.Tag;
    }
    src.Visibility = Visibility.Hidden;
}
```

Измените атрибуты PreviewDragEnter и PreviewDragOver для компонента grid1 в xaml-файле:

```
<UniformGrid x:Name="grid1" …
    PreviewDragEnter="canvas1_DragEnter"
    PreviewDragOver="canvas1_DragEnter" Drop="grid1_Drop">
```

После сделанных изменений обработчик grid1_PreviewDragEnter уже не будет связан ни с одним событием, поэтому его описание в классе MainWindow можно удалить.

Результат. При перетаскивании одного зверя на другого более сильный «поедает» более слабого. То же самое происходит, если один из зве-

рей перетаскивается в клетку, уже занятую другим зверем. Заметим, что теперь события DragEnter и DragOver *всех* компонентов (и канвы, и меток, и полей ввода) связаны с одним и тем же обработчиком – canvas1_DragEnter.

Комментарии

1. С помощью свойства Tag, имеющегося у любого компонента, определяется относительная «сила» зверей. При помещении зверя в клетку информация о силе зверя сохраняется в свойстве Tag клетки (т. е. компонента TextBox). Для того чтобы сохранить возможность помещения зверя в пустую клетку, в начале программы свойства Tag всех полей ввода полагаются равными 0. Свойство Tag описано как object, поэтому в нем можно хранить данные любого типа. В нашем случае в свойствах Tag было бы удобно хранить данные *целого* типа, однако при задании этого свойства в xaml-файле ему присваивается *строка*, поэтому при обращении к свойству Tag из обработчиков событий оно приводится к типу string. Чтобы для сравнения значений этого свойства можно было использовать операцию «<», из полученных строк извлекаются первые символы.

2. При определении нового значения для позиции источника src в новом фрагменте обработчика canvas1_Drop можно было использовать текущие координаты мыши (как в старом фрагменте, соответствующем случаю, когда приемником является компонент Canvas), однако проще воспользоваться свойствами Canvas.Left и Canvas.Top приемника trg. Кроме того, подобный способ позволяет поместить метку-источник в точности на место метки-приемника (независимо от того, в какой части метки-приемника была отпущена кнопка мыши).

Ошибка. Если при перетаскивании метки отпустить ее над ней самой, то метка исчезнет. Таким образом, *зверь поедает самого себя*.

Исправление. В методе canvas1_Drop *перед* оператором

```
if ((src.Tag as string)[0] > (trg.Tag as string)[0])
```

вставьте следующий фрагмент:

```
if (src == trg)
    return;
```

7.4. Действия в случае перетаскивания на недопустимый приемник

Измените метод label1_MouseDown:

```
private void label1_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    var t = e.Source as TextBlock;
    if (t == null)
```

```
        return;
    if (e.ChangedButton == MouseButton.Left)
        if (DragDrop.DoDragDrop(t, t, DragDropEffects.All) ==
          DragDropEffects.None)
            t.Visibility = Visibility.Hidden;
}
```

Результат. Если перетаскивание метки-«зверя» завершается за пределами окна (в этом случае курсор перетаскивания имеет вид запрещающего знака), то зверь «убегает» из зоопарка, и его метка в окне исчезает.

Комментарий

Метод DoDragDrop, запускающий режим перетаскивания, не возвращает управление вызвавшей его программе до тех пор, пока режим перетаскивания не завершится; при этом он возвращает тот эффект, которым завершилось перетаскивание. В частности, если он вернул значение DragDropEffects.None, то это означает, что источник был отпущен над *недопустимым приемником*, т. е. в тот момент, когда курсор перетаскивания имел запрещающий знак. В нашей программе все компоненты на окне в любой момент времени являются допустимыми приемниками, поэтому метод DoDragDrop вернет значение DragDropEffects.None только при отпускании источника *за пределами окна*.

7.5. Дополнительное выделение источника и приемника в ходе перетаскивания

Измените методы label1_MouseDown и canvas1_DragEnter:

```
private void label1_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    var t = e.Source as TextBlock;
    if (t == null)
        return;
    t.Foreground = Brushes.Red;
    if (e.ChangedButton == MouseButton.Left)
        if (DragDrop.DoDragDrop(t, t, DragDropEffects.All) ==
          DragDropEffects.None)
            t.Visibility = Visibility.Hidden;
    t.Foreground = Brushes.Black;
}
private void canvas1_DragEnter(object sender, DragEventArgs e)
{
    e.Handled = true;
    e.Effects = DragDropEffects.Move;
```

```
    var trg = e.Source as TextBlock;
    if (trg == null)
        return;
    trg.Background = Brushes.Yellow;
}
```

В методе canvas1_Drop после оператора

```
var src = e.Data.GetData(typeof(TextBlock)) as TextBlock;
```

добавьте новый оператор:

```
trg.Background = null;
```

Для компонента canvas1 создайте обработчик события DragLeave:

```
<Canvas x:Name="canvas1" … DragLeave="canvas1_DragLeave" >
```

```
private void canvas1_DragLeave(object sender, DragEventArgs e)
{
    e.Handled = true;
    var trg = e.Source as TextBlock;
    if (trg == null)
        return;
    trg.Background = null;
}
```

Результат. В режиме перетаскивания цвет текста метки-источника изменяется на красный, а текущая метка-приемник изображается на желтом фоне.

Комментарии

1. Для дополнительного выделения *источника* перетаскивания достаточно настроить соответствующим образом его свойства до вызова метода DoDragDrop, а после завершения этого метода (т. е. после выхода из режима перетаскивания) восстановить измененные свойства. Для выделения текущего *приемника* следует изменять его свойства в обработчике события DragEnter, а восстанавливать – в обработчике события DragLeave, которое возникает в тот момент, когда курсор мыши покидает текущий приемник. Заметим, что в случае *недопустимого* приемника событие DragLeave возникает также в ситуации, когда перетаскивание *завершается* над подобным приемником. Если же перетаскивание завершается над допустимым приемником, то событие DragLeave не возникает и для восстановления свойств приемника надо использовать обработчик события Drop (что также делается в нашей программе).

2. Обратите внимание на способ восстановления исходных цветов фона и текста для меток. Для восстановления *цвета фона* свойству Background присваивается значение null (такое же значение имеет это свойство и в начальный момент работы программы; для того чтобы в этом убедиться, достаточно обратиться к окну Properties, выбрав на ма-

кете окна одну из меток: свойство Background будет помечено как «No Brush»). Здесь проявляется важная особенность свойств зависимости: *если свойство не определено явным образом для некоторого компонента, то его значение берется из одноименного свойства родителя этого компонента* (в нашем случае фактическое значение свойства Background для всех меток предоставляет компонент Canvas).

С цветом текста дело обстоит по-другому: обратившись к окну Properties, можно убедиться в том, что свойство Foreground для меток имеет по умолчанию значение, соответствующее кисти черного цвета. Поэтому для восстановления исходного цвета текста достаточно присвоить свойству Foreground значение Brushes.Black.

В WPF имеется много способов настройки свойств зависимости (в том числе основанных на *шаблонах*, *стилях* и различных видах *триггеров*), и получение значения свойства от родителя – лишь одна из возможностей.

7.6. Настройка вида курсора в режиме перетаскивания

Для окна MainWindow создайте обработчик события GiveFeedback:

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

```
private void Window_GiveFeedback(object sender,
  GiveFeedbackEventArgs e)
{
    if (e.Effects == DragDropEffects.Move)
    {
        e.UseDefaultCursors = false;
        Mouse.SetCursor(Cursors.Hand);
    }
    else
        e.UseDefaultCursors = true;
    e.Handled = true;
}
```

Результат. В режиме перетаскивания курсор мыши имеет вид «указывающей руки» (при выходе курсора за пределы окна он по-прежнему принимает вид запрещающего знака).

Комментарий

Событие GiveFeedback специально предназначено для возможности изменения вида курсора в режиме перетаскивания; это событие воздействует на вид курсора для всех компонентов-приемников. Определить текущий эффект перетаскивания можно с помощью свойства e.Effects. В зависимости от этого эффекта можно установить вид курсора, однако перед

этим необходимо отключить отображение стандартных курсоров перетаскивания, положив свойство e.UseDefaultCursors равным false. Требуемый курсор надо указать в методе SetCursor класса Mouse.

7.7. Информация о текущем состоянии программы. Кнопки с комбинированным содержимым

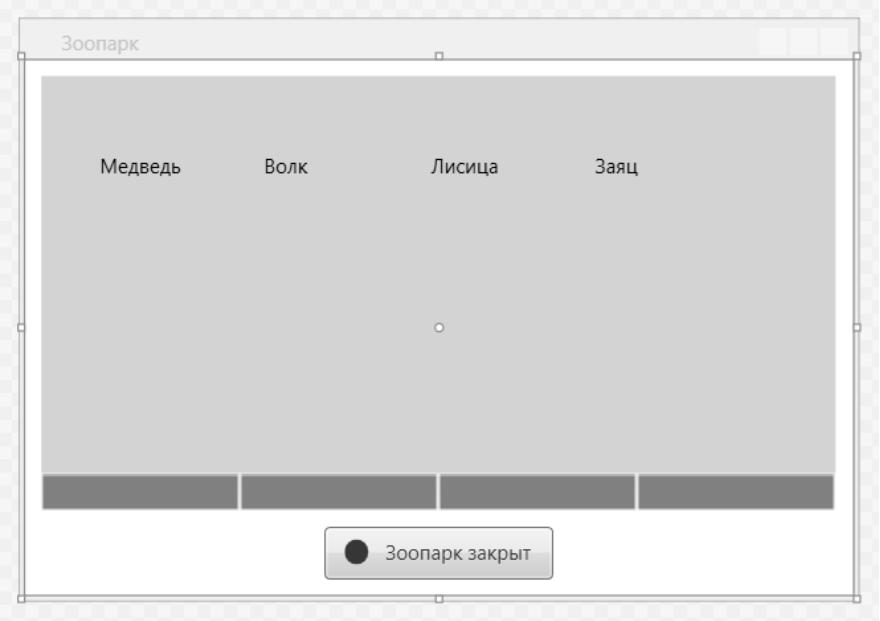


Рис. 27. Макет окна приложения ZOO (третий вариант)

```
<Window x:Class="ZOO.MainWindow"
    … >
  <StackPanel >
    <Canvas … >
      …
    </Canvas>
    <UniformGrid … >
      …
    </UniformGrid>
    <Button x:Name="button1" HorizontalAlignment="Center"
        Margin="0,0,0,10" >
      <StackPanel Margin="5,0" Orientation="Horizontal">
        <Ellipse x:Name="mark1" Margin="5" Fill="Red"
            Width="15" Height="15" />
        <TextBlock x:Name="caption1" Margin="5" Foreground="Red"
            Text="Зоопарк закрыт" />
      </StackPanel>
    </Button>
  </StackPanel>
```

```
</Window>
```

В метод label1_MouseDown добавьте операторы:

```
string s = "";
for (int i = 0; i < 4; i++)
{
    if (canvas1.Children[i].IsVisible)
        return;
    s += (grid1.Children[i] as TextBox).Text;
}
if (s == "")
    return;
mark1.Fill = Brushes.Green;
caption1.Foreground = Brushes.Green;
caption1.Text = "Зоопарк открыт";
```

Результат. В начале работы программы кнопка содержит изображение красного круга и текст «Зоопарк закрыт» (тоже красного цвета). Если в результате перетаскивания меток все они исчезли и при этом хотя бы одно поле ввода оказалось заполненным, то на кнопке выводится текст «Зоопарк открыт», а цвет оформления кнопки меняется с красного на зеленый. При попытке перетаскивания меток на кнопку или свободную часть окна слева или справа от кнопки курсор принимает вид запрещающего знака.

Комментарии

1. Любой компонент, имеющий свойство Content, может включать сколь угодно сложный набор компонентов в качестве своего содержимого. В нашем случае кнопка button1 содержит группирующий компонент StackPanel, который позволяет разместить на ней и изображение (цветной круг), и текст.

2. Для перебора дочерних компонентов как канвы canvas1, так и таблицы grid1 мы использовали свойство-коллекцию Children.

3. Запрещающий знак курсора перетаскивания объясняется тем, что для кнопки, как и для ее родителя – компонента StackPanel верхнего уровня, – свойство AllowDrop не установлено, и поэтому принимает значение по умолчанию, равное false.

7.8. Восстановление исходного состояния

В описание класса MainWindow добавьте поля:

```
double[] startPosX = new double[4];
double startPosY;
```

В конструктор класса MainWindow добавьте операторы:

```
startPosY = Canvas.GetTop(canvas1.Children[0]);
for (int i = 0; i < 4; i++)
```

```
    startPosX[i] = Canvas.GetLeft(canvas1.Children[i]);
button1.Focus();
```

Для компонента button1 определите обработчик события Click:

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

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    for (int i = 0; i < 4; i++)
    {
        var t = canvas1.Children[i];
        t.Visibility = Visibility.Visible;
        Canvas.SetTop(t, startPosY);
        Canvas.SetLeft(t, startPosX[i]);
        var tb = grid1.Children[i] as TextBox;
        tb.Text = "";
        tb.Tag = "0";
    }
    mark1.Fill = Brushes.Red;
    caption1.Foreground = Brushes.Red;
    caption1.Text = "Зоопарк закрыт";
}
```

Результат. Исходное положение меток-«зверей» сохраняется в полях startPosX и startPosY. В дальнейшем при нажатии на кнопку button1 исходное положение «зверей» восстанавливается, а «клетки» освобождаются. Кроме того, при запуске программы кнопка button1 становится активной (принимает фокус), что позволяет для ее нажатия просто нажать клавишу пробела или клавишу Enter.

Комментарий

Обратите внимание на то, что в методе button1_Click при обращении к элементу коллекции canvas1.Children не выполняется приведение к его фактическому типу (равному TextBlock). В этом нет необходимости, поскольку все настраиваемые для этого элемента свойства уже имеются у класса UIElement – общего предка всех визуальных компонентов, а коллекция Children возвращает элементы типа UIElement. В то же время свойства Text и Tag компонента TextBox *отсутствуют* у класса UIElement, поэтому для настройки этих свойств приходится выполнять явное приведение элементов коллекции grid1.Children к типу TextBox.

8. Курсоры и иконки: CURSORS

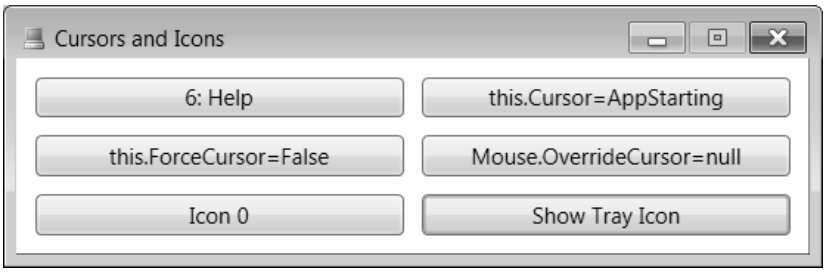


Рис. 28. Окно приложения CURSORS

8.1. Использование стандартных курсоров

Рис. 29. Макет окна приложения CURSORS (первый вариант)

```
<Window x:Class="CURSORS.MainWindow"
    …
    Title="Cursors and Icons"
    WindowStartupLocation="CenterScreen"
    SizeToContent="WidthAndHeight" ResizeMode="CanMinimize" >
  <UniformGrid Margin="5" Columns="2" >
    <Button x:Name="button1" Content="0: null" MinWidth="200"
        Margin="5" PreviewMouseDown="button1 PreviewMouseDown" />
  </UniformGrid>
</Window>
```

В описание класса MainWindow добавьте вспомогательное поле:

```
List<Cursor> cur = new List<Cursor>(29);
```

В конструктор класса MainWindow добавьте следующий фрагмент:

```
cur.Add(null);
foreach (System.Reflection.PropertyInfo pi in
  typeof(Cursors).GetProperties())
    cur.Add((Cursor)pi.GetValue(null, null));
button1.Tag = 0;
```

И определите обработчик события PreviewMouseDown для компонента button1, уже указанный в xaml-файле:

```
private void button1_PreviewMouseDown(object sender,
  MouseButtonEventArgs e)
```

```
{
    int k = (int)button1.Tag, c = cur.Count;
    switch (e.ChangedButton)
    {
        case MouseButton.Left:
            k = (k + 1) % c;
            break;
        case MouseButton.Right:
            k = (k - 1 + c) % c;
            break;
    }
    button1.Content = k + ": " +
        (cur[k] == null ? "null" : cur[k].ToString());
    button1.Cursor = cur[k];
    button1.Tag = k;
    e.Handled = true;
}
```

Результат. Щелчок левой или правой кнопкой мыши на кнопке button1 приводит к смене вида курсора для данной кнопки (на кнопке при этом выводится порядковый номер курсора и его название). Перебор всех 28 стандартных курсоров выполняется циклически; порядок перебора курсоров соответствует порядку их размещения в выпадающем списке для свойства Cursor в окне Properties. При нажатии не левой, а правой кнопки мыши курсоры перебираются в обратном порядке. Предусмотрено также особое значение null для курсора кнопки, при котором используется курсор ее родительского компонента (это значение имеет номер 0).

Комментарии

1. Мы использовали событие PreviewMouseDown, так как событие Click компонента Button реагирует только на *левую* кнопку мыши, а событие MouseDown переопределено для компонента Button таким образом, что с ним нельзя связать пользовательский обработчик. Обратите внимание на оператор e.Handled = true в конце обработчика button1_PreviewMouseDown. Благодаря этому оператору событие, связанное со щелчком левой кнопки мыши, «не доходит» до встроенного обработчика события MouseDown кнопки Button, и поэтому в окне не имитируется «нажатие» на кнопку. Таким образом, щелчок как левой, так и правой кнопкой мыши приводит к одинаковым результатам – смене вида курсора (без дополнительных эффектов для щелчка левой кнопкой).

2. Все стандартные курсоры можно получить с помощью класса Cursors, в котором с каждым стандартным курсором связано особое свойство, доступное только для чтения. В классе Cursors не предусмотрено

средств для циклического перебора стандартных курсоров. Поэтому в конструкторе окна с помощью *механизма отражения* (reflection), примененного к классу Cursors, формируется *список* cur, содержащий все курсоры, определенные в классе Cursors. Механизм отражения в данной книге подробно не рассматривается (см., например, [5, гл. 19]); отметим лишь, что он основан на использовании информации, хранящейся в объекте особого типа Type, который можно получить для *любой* переменной a, вызвав для нее метод a.GetType(), и для *любого* типа t, выполнив для него операцию typeof(t). Мы воспользовались типом Type для класса Cursors, чтобы получить информацию обо всех *свойствах* данного класса. Для этого мы вызвали метод GetProperties класса Type, возвращающий последовательность элементов типа PropertyInfo из пространства имен System.Reflection, с помощью которой можно определить как *имена*, так и *значения* всех свойств. *Значения* стандартных курсоров мы сохранили в списке cur типа List<Cursor>, а для определения их *имен* использовали вариант метода ToString, определенный в классе Cursor.

3. Для хранения индекса курсора, связанного с кнопкой button1, используется свойство Tag данной кнопки (это свойство мы уже использовали в проекте ZOO). Поскольку свойство Tag имеет тип object, ему можно присвоить значение любого типа, однако при *считывании* его значения необходимо выполнять явное преобразование к фактическому типу содержащегося в нем объекта (в нашем случае к типу int). Последний оператор в конструкторе окна обеспечивает инициализацию свойства Tag числом 0, связанным со значением null в списке cur, так как по умолчанию свойство Cursor для компонентов окна не задается (вид курсора определяется по курсору окна).

4. Циклический перебор индексов в диапазоне от 0 до c (где c = str.Count – 1, a str.Count равно количеству элементов, содержащихся в списке str) выполняется с применением операции % (взятие остатка от деления нацело). Дополнительное слагаемое c в выражении (k – 1 + c) % c добавлено для того, чтобы выражение в скобках никогда не принимало *отрицательных* значений.

5. Для размещения кнопок в окне мы использовали групповой компонент UniformGrid (мы уже применяли этот компонент в проекте ZOO). Компонент UniformGrid удобен при размещении одинаковых дочерних компонентов, так как для него не надо особым образом настраивать свойства строк и столбцов (достаточно указать количество столбцов в свойстве Columns), а также указывать индексы строки и столбца в дочерних компонентах (очередной дочерний компонент просто добавляется в следующую ячейку таблицы).

6. Начиная с версии C# 6.0, реализованной в Visual Studio 2015, использованное в методе button1_PreviewMouseDown выражение

```
(cur[k] == null ? "null" : cur[k].ToString())
```

можно записать более кратко:

```
(cur[k]?.ToString() ?? "null")
```

В этом варианте используется новая возможность языка C#: так называемая *null-условная операция* «?.». Выражение o?.m возвращает член m объекта o, если объект o не равен null, и null в противном случае (напомним, что при использовании выражения o.m в ситуации, когда объект o равен null, возбуждается исключение NullReferenceException). Операцию ?. удобно применять совместно с ранее введенной в C# операцией ??, которая возвращает первый операнд, если он не равен null, и второй операнд в противном случае.

8.2. Установка курсора для окна и приложения в целом

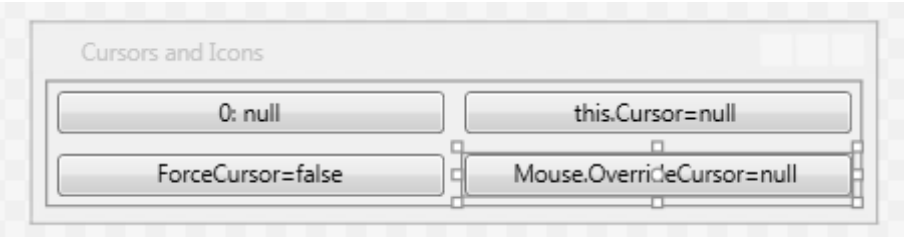


Рис. 30. Макет окна приложения CURSORS (второй вариант)

```
<Window x:Class="CURSORS.MainWindow"
    … >
  <UniformGrid Margin="5" Columns="2">
    <Button x:Name="button1" Content="0: null" MinWidth="200"
        Margin="5" PreviewMouseDown="button1_PreviewMouseDown"/>
    <Button x:Name="button2" Content="this.Cursor=null"
        MinWidth="200" Margin="5" Click="button2_Click"/>
    <Button x:Name="button3" Content="this.ForceCursor=false"
        MinWidth="200" Margin="5" Click="button3_Click"/>
    <Button x:Name="button4" Content="Mouse.OverrideCursor=null"
        MinWidth="200" Margin="5" Click="button4_Click"/>
  </UniformGrid>
</Window>
```

Определите обработчики события Click для компонентов button2, button3, button4, уже указанные в xaml-файле:

```
private void button2_Click(object sender, RoutedEventArgs e)
{
    Cursor = button1.Cursor;
    button2.Content = "this.Cursor=" +
        (Cursor == null ? "null" : Cursor.ToString());
}
private void button3_Click(object sender, RoutedEventArgs e)
```

```
{
    ForceCursor = !ForceCursor;
    button3.Content = "this.ForceCursor=" + ForceCursor;
}
private void button4_Click(object sender, RoutedEventArgs e)
{
    Mouse.OverrideCursor = Mouse.OverrideCursor == null ?
      button1.Cursor : null;
    button4.Content = "Mouse.OverrideCursor=" +
      (Mouse.OverrideCursor == null ? "null" :
      Mouse.OverrideCursor.ToString());
}
```

Результат. Нажатие на кнопку button2 распространяет действие нового курсора на все окно, включая те находящиеся в окне кнопки, для которых свойство Cursor равно null. Нажатие на кнопку button3 устанавливает для *всех* компонентов окна курсор окна, независимо от того, задан или нет для компонента какой-либо курсор (для этого используется свойство окна ForceCursor, которое полагается равным true). Нажатие на кнопку button4 изменяет свойство Mouse.OverrideCursor, которое *устанавливает курсор для всего приложения*, независимо от настроек любых других курсоров (действие свойства Mouse.OverrideCursor отменяется, если положить его равным null).

Комментарий

Свойство Mouse.OverrideCursor чаще всего используется для того, чтобы при выполнении какого-либо длительного действия установить для приложения *курсор ожидания* (имеющий вид песочных часов или каким-либо другим образом указывающий на то, что пользователю требуется подождать, пока программа не завершит требуемые действия). Для этих целей достаточно положить свойство Mouse.OverrideCursor равным Cursors.Wait, а после завершения действий – восстановить его исходное значение null. Если загрузка и начальные вычисления программы также занимают много времени, то на время этой загрузки можно положить свойство Mouse.OverrideCursor равным Cursors.AppStarting – *курсору ожидания окончания загрузки программы* (см. проект TRIGFUNC).

8.3. Использование в программе дополнительных курсоров

Кроме стандартных курсоров в программе можно использовать дополнительные курсоры. Файлы с курсорами имеют расширение .cur; их можно найти, например, в подкаталоге Cursors каталога Windows. Заметим, что в приложениях WPF (в отличие от приложений Windows Forms) можно использовать цветные курсоры, а также анимированные курсоры

(файлы с расширением .ani). Иконка любого файла, содержащего курсор, соответствует изображению этого курсора, что позволяет быстро ознакомиться с содержимым имеющихся файлов курсоров, используя Проводник.

Выберите два cur-файла и скопируйте их в каталог проекта CURSORS под именами C1.cur и C2.cur.

Загрузите в среду Visual Studio проект CURSORS, если это еще не сделано, и выполните команду меню Project | Add Existing Item… (Shift+Alt+A). В появившемся диалоговом окне выберите файл C1.cur (имя файла можно ввести непосредственно в строке ввода; можно также указать в выпадающем списке «Files of type» вариант «All Files» и выбрать файл C1.cur из списка всех файлов, содержащихся в каталоге). Нажмите клавишу Enter или кнопку Add. В результате файл C1.cur будет добавлен к проекту CURSORS и появится в окне Solution Explorer. Аналогичными действиями добавьте к проекту файл C2.cur.

Если выделить один из добавленных файлов в окне Solution Explorer, то в окне Properties появится список его свойств. Проверьте, что свойство Build Action для этих файлов равно значению Resource.

Файлы, добавленные в приложение описанными выше действиями, являются *встроенными ресурсами приложения*. Эти файлы встраиваются непосредственно в исполняемый файл приложения, причем к ним можно обращаться по именам как в xaml-файле, так и в программном коде.

В данном приложении мы будем использовать ресурсы в программном коде (в проектах TEXTEDIT версии 4 и COLORS ресурсы будут указываться в xaml-файле).

Добавьте в конструктор класса MainWindow следующий фрагмент:

```
var rs = Application.GetResourceStream(new
  Uri("pack://application:,,,/C" + i + ".cur"));
cur.Add(new Cursor(rs.Stream));
```

Результат. При создании окна новые курсоры загружаются в программу и добавляются к списку доступных курсоров. Заметим, что для курсоров, импортированных из файлов, метод ToString возвращает строку None.

Комментарий

Для загрузки курсора из ресурсов приложения используется вариант конструктора класса Cursor с параметром-*потоком*, который, в свою очередь, создается на основе встроенного ресурса методом Application.GetResourceStream. Вместо указания *полного* пути к встроенному ресурсу в конструкторе класса Uri можно использовать *относительный* путь с дополнительным параметром:

```
new Uri("C" + i + ".cur", UriKind.Relative)
```

8.4. Работа с иконками

Небольшие изображения, называемые *иконками*, *значками* или *пиктограммами*, хранятся в файлах с расширением .ico. В версии Visual Studio 2005, 2008, 2010 включались графические библиотеки (в виде архивов с именами – VS2005ImageLibrary.zip, VS2008ImageLibrary.zip, VS2010ImageLibrary.zip), содержащие большой набор ico-файлов, однако в последних версиях подобные библиотеки отсутствуют. Впрочем, в Интернете имеется множество сайтов, с которых можно скачать коллекции иконок, да и найти несколько ico-файлов на компьютере не составляет труда.

Добавление к проекту ico-файлов выполняется аналогично добавлению cur-файлов. Для определенности будем считать, что к проекту добавлены файлы Computer.ico и Folder.ico. Как и в случае cur-файлов, после добавления ico-файла в проект следует проверить, что его свойство Build Action равно значению Resource.

```
<Window x:Class="CURSORS.MainWindow"
    … >
  <UniformGrid Margin="5" Columns="2">
    …
    <Button x:Name="button5" Content="Icon 0" MinWidth="200"
        Margin="5" Click="button5_Click"/>
  </UniformGrid>
</Window>
```

В описание класса MainWindow добавьте поле

```
BitmapImage[] ico = new BitmapImage[2];
```

В конструктор класса MainWindow добавьте следующий фрагмент:

```
ico[0] = new BitmapImage(new
  Uri("pack://application:,,,/Computer.ico"));
ico[1] = new BitmapImage(new
  Uri("pack://application:,,,/Folder.ico"));
Icon = ico[0];
button5.Tag = 0;
```

И определите обработчик Click кнопки button5, уже указанный в xaml-файле:

```
private void button5_Click(object sender, RoutedEventArgs e)
{
    int k = ((int)button5.Tag + 1) % 2;
    button5.Content = "Icon " + k;
    button5.Tag = k;
    Icon = ico[k];
}
```

Результат. Кнопка button5 изменяет иконку приложения как в заголовке окна, так и на его кнопке, находящейся на панели задач.

Комментарии

1. Для работы с иконками в библиотеке WPF не предусмотрено специального класса; вместо этого иконки можно загружать в «универсальный» класс BitmapImage, предназначенный для хранения растровых изображений. Загрузка растровых изображений, хранящихся в виде ресурсов, выполняется проще, чем загрузка курсоров: достаточно задать путь к файлу ресурса в конструкторе объекта Uri, который, в свою очередь, указать в качестве параметра конструктора класса BitmapImage (таким образом, создавать специальный поток не требуется).

2. По умолчанию свойство Icon окна равно null; в этом случае в заголовке окна и на кнопке приложения на панели задач отображается стандартная иконка.

3. Иконка может быть связана не только с окном, но и с приложением в целом, однако иконка приложения используется только при отображении соответствующего exe-файла в Проводнике или подобных ему файловых браузерах, поддерживающих отображение иконок. Для задания иконки приложения надо открыть в редакторе вкладку со свойствами проекта, используя команду меню Project | <*имя проекта*> Properties…, и указать нужную иконку в строке Icon, находящейся в разделе Application.

8.5. Размещение иконки в области уведомлений. Использование объектов из библиотеки Windows Forms

На последнем этапе разработки проекта CURSORS мы добавим к нему возможность отображения его иконки в *области уведомлений* (notification area, traybar) панели задач.

В библиотеке Windows Forms для этих целей был предусмотрен невизуальный компонент NotifyIcon. В библиотеке WPF аналогичное средство отсутствует. Тем не менее мы можем реализовать данную возможность, использовав в нашем проекте класс NotifyIcon из библиотеки Windows Forms.

Прежде всего, надо подключить необходимые компоненты библиотеки Windows Forms. Для этого щелкните правой кнопкой мыши на пункте References в окне Solution Explorer, выберите из появившегося меню команду Add Reference…, отметьте в появившемся списке стандартных компонентов библиотеки .NET пункты System.Windows.Forms и System.Drawing (около этих пунктов должны появиться «галочки») и закройте окно, нажав кнопку OK. После этого выбранные компоненты появятся в разделе References.

Указывать новые пространства имен в директивах using не следует, так как это приведет к многочисленным конфликтам, связанным с совпадением имен классов из библиотек Windows Forms и WPF. Поэтому для классов из библиотеки Windows Forms придется указывать их полные имена.

В приложение необходимо также добавить файл с иконкой, которую мы хотим связать с объектом NotifyIcon (уже имеющиеся иконки-ресурсы использовать не удастся, так как для ресурсов, применяемых в библиотеке Windows Forms, надо указать другое значение свойства Build Action). Предположим, что новый файл с иконкой имеет имя ComputerWF.ico. Его необходимо загрузить как встроенный ресурс и *изменить для него значение свойства Build Action на Embedded Resource*.

Теперь можно приступать к модификации xaml- и cs-файлов.

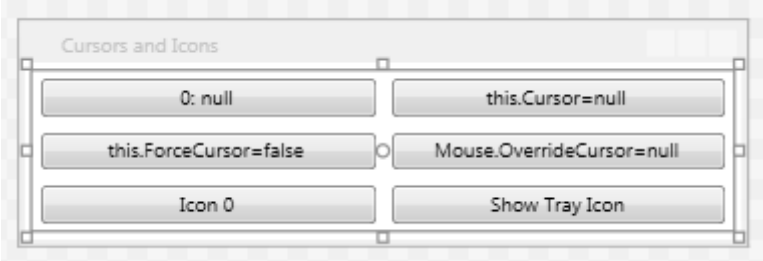


Рис. 31. Макет окна приложения CURSORS (третий вариант)

```
<Window x:Class="CURSORS.MainWindow"
    … >
  <UniformGrid Margin="5" Columns="2">
    …
    <Button x:Name="button6" Content="Show Tray Icon"
        MinWidth="200" Margin="5" Click="button6_Click"/>
  </UniformGrid>
</Window>
```

В описание класса MainWindow добавьте поле

```
System.Windows.Forms.NotifyIcon notifyIcon1 =
  new System.Windows.Forms.NotifyIcon();
```

В конструктор класса MainWindow добавьте следующий фрагмент, в котором настраиваются свойства объекта notifyIcon1:

```
notifyIcon1.Icon = new System.Drawing.Icon(GetType(),
  "ComputerWF.ico");
notifyIcon1.Visible = false;
notifyIcon1.Text = "Icon in Traybar";
notifyIcon1.Click += notifyIcon1_Click;
```

И определите два обработчика: для события Click кнопки button6, который уже указан в xaml-файле (и для которого, следовательно, уже имеется заготовка) и для события Click объекта notifyIcon1 (этот объект мы соз-

дали программно, поэтому текст его обработчика придется вводить полностью):

```
private void button6_Click(object sender, RoutedEventArgs e)
{
    bool b = (string)button6.Content == "Show Tray Icon";
    notifyIcon1.Visible = b;
    ShowInTaskbar = !b;
    button6.Content = b ? "Hide Tray Icon" : "Show Tray Icon";
}
private void notifyIcon1_Click(object sender, EventArgs e)
{
    button6_Click(null, null);
}
```

Результат. Нажатие на кнопку button6 скрывает кнопку приложения на панели задач и отображает связанную с ним иконку в области уведомлений в правой части панели задач. При наведении курсора на эту иконку возникает всплывающая подсказка «Icon in Traybar». Повторное нажатие кнопки button6 (или щелчок на иконке, расположенной в области уведомлений) восстанавливает исходное представление кнопки приложения на панели задач.

Комментарий

Обычно в области уведомлений размещаются иконки приложений, работающих в фоновом режиме и использующих окна только для показа и изменения своих настроек. Кроме всплывающих подсказок, возникающих при наведении курсора на иконку, компонент NotifyIcon позволяет выводить на экран рядом с иконкой «сообщения в пузыре» (используя метод ShowBalloonTip и связанные с ним свойства BalloonTipIcon, BalloonTipTitle, BalloonTipText), а также связывать с иконкой *контекстное меню*, вызываемое с помощью правой кнопки мыши (свойство ContextMenuStrip); заметим, что контекстное меню в данном случае тоже необходимо создавать на основе соответствующего класса из библиотеки Windows Forms.

9. Меню и работа с текстовыми файлами: TEXTEDIT, версия 1

Рис. 32. Верхняя часть окна приложения TEXTEDIT версии 1

9.1. Создание меню

Рис. 33. Макет окна приложения TEXTEDIT версии 1 (первый вариант)

```
<Window x:Class="TEXTEDIT.MainWindow"
    …
    Title="Text Edit" Width="500" Height="400"
    WindowStartupLocation="CenterScreen" >
  <DockPanel >
    <Menu DockPanel.Dock="Top" >
      <MenuItem x:Name="file1" Header="_File" >
        <MenuItem x:Name="exit1" Header="E_xit"
            InputGestureText="Esc" Click="exit1 Click" />
      </MenuItem>
    </Menu>
    <TextBox x:Name="textBox1" Text="" AcceptsReturn="True"
        VerticalScrollBarVisibility="Auto"
        HorizontalScrollBarVisibility="Auto" />
  </DockPanel>
</Window>
```

В конструктор класса MainWindow добавьте оператор

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

И определите обработчик события Click для команды меню exit1, уже указанный в xaml-файле:

```
private void exit1_Click(object sender, RoutedEventArgs e)
{
    Close();
}
```

Результат. Окно программы содержит главное меню и область редактирования, которая в начале выполнения программы получает фокус. При изменении размеров окна автоматически изменяются размеры области редактирования (компонента textBox1). Если введенный текст выходит за границы клиентской области окна (по горизонтали или вертикали), то в окне отображается соответствующая полоса прокрутки.

Команды главного меню можно вызывать с помощью мыши, клавиш быстрого доступа, указанных рядом с названием пункта меню второго уровня, или клавиатурных комбинаций Alt+<*подчеркнутая буква в названии команды меню* ***первого уровня***> (если развернуто меню второго уровня, то для выбора команды необходимо нажать клавишу, соответствующую подчеркнутому символу в названии команды *без клавиши Alt*). Для отображения символов подчеркивания следует нажать клавишу Alt (заметим, что при этом действии автоматически активизируется строка меню).

Команда Exit закрывает окно и тем самым завершает программу.

Комментарии

1. В качестве группового компонента мы использовали компонент DockPanel, так как он предоставляет удобные средства для *стыковки* вспомогательных компонентов (таких как меню, статусная панель и панель инструментов) к границам окна, размещая в оставшейся области окна основной компонент (в данном случае многострочное поле редактирования textBox1).

2. Чтобы установить для поля ввода textBox1 многострочный режим, достаточно задать его свойство AcceptsReturn равным true. Для многострочных полей целесообразно также устанавливать режим Auto для отображения полос прокрутки.

3. В дизайнере WPF, как и в дизайнере Windows Forms, имеются средства настройки меню, хотя и не столь наглядные (для добавления нового пункта, разделителя или меню следующего уровня надо воспользоваться контекстным меню, щелкнув правой кнопкой мыши на компоненте Menu в окне дизайнера). Однако в подобной визуальной настройке нет особой необходимости, так как структуру меню легко задать непосредственно в xaml-файле с помощью компонента Menu и вложенных в него компонентов MenuItem и Separator.

Недочет 1. При нажатии клавиши Tab фокус ввода перемещается с поля редактирования на строку меню. Это, во-первых, не позволяет использовать в тексте символы табуляции и, во-вторых, не соответствует стандартным способам перехода в меню.

Исправление. Добавьте в xaml-файле к элементу textBox1 атрибут AcceptsTab="True".

Результат. Теперь нажатие клавиши Tab в поле ввода обеспечивает ввод символа табуляции.

Комментарий

Для перехода к меню по-прежнему остается доступной комбинация Ctrl+Tab. Можно было бы ожидать, что для блокировки перевода фокуса на меню достаточно положить свойство Focusable компонента Menu равным false, однако этот способ не срабатывает. Впрочем, при желании можно заблокировать *все пункты меню первого уровня*, положив для каждого из них Focusable равным false (однако при этом будет также потеряна стандартная возможность перехода в меню по нажатию клавиши Alt, а вместе с ней и возможность использования клавиатурных Alt-комбинаций).

Недочет 2. Несмотря на указание клавиши быстрого доступа Esc рядом с пунктом меню Exit, нажатие на эту клавишу *не приводит к завершению программы*. Это объясняется тем, что свойство InputGestureText лишь выводит указанный текст в правой части пункта меню, не обрабатывая нажатия соответствующих клавиш. Для подобной обработки можно использовать событие KeyDown, однако при связывании с пунктами меню *команд WPF* (что мы собираемся сделать в следующем пункте) это не требуется. Таким образом, отмеченный недочет будет автоматически исправлен при последующей модификации программы.

9.2. Команды WPF и связывание с ними пунктов меню

```
<Window x:Class="TEXTEDIT.MainWindow"
    … >
  <Window.CommandBindings>
    <CommandBinding x:Name="exit0"
        Command="ApplicationCommands.Close"
        Executed="exit0_Executed" />
  </Window.CommandBindings>
  <DockPanel >
    <Menu DockPanel.Dock="Top" >
      <MenuItem x:Name="file1" Header="_File" >
        <MenuItem x:Name="exit1" Header="E_xit"
            InputGestureText="Esc" Click="exit1_Click"
            Command="ApplicationCommands.Close" />
```

```
      </MenuItem>
    </Menu>
    …
  </DockPanel>
</Window>
```

Удалите из класса MainWindow метод exit1_Click и определите обработчик exit0_Executed, указанный в xaml-файле:

```
private void exit0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    Close();
}
```

Результат. Теперь при разворачивании пункта меню File отображается команда «Закрыть», закрывающая окно и завершающая программу.

Комментарий

В новом варианте программы с пунктом меню связывается не обработчик события Click, а *команда WPF*. Команды WPF, называемые также *управляющими командами* (control commands), являются важным элементом данной библиотеки, и их использование считается более предпочтительным, чем использование обработчиков событий. Это связано, прежде всего, с тем, что, определив действие для некоторой команды, мы можем связать его со многими компонентами, просто указав эту команду в качестве свойства Command этих компонентов. Таким образом, действия пользователя будут обрабатываться *централизованно*: независимо от того, какой управляющий компонент он применил, выполняться будет одна и та же последовательность действий, определенная в команде, связанной с этим компонентом. Кроме того, такой подход позволяет более четко распределить обязанности между программистом, разрабатывающим логику программы, и дизайнером, определяющим ее макет: программист реализует команды, обеспечивающие требуемую функциональность приложения, а дизайнер просто связывает эти команды с нужными интерфейсными компонентами в xaml-файле.

В библиотеку WPF включено большое количество стандартных команд. Основные стандартные команды реализованы в виде свойств класса ApplicationCommands, одно из которых (Close) мы и использовали в программе. Можно также создавать свои собственные команды, в дальнейшем мы подробно рассмотрим эту возможность (см. проект TEXTEDIT версии 2, п. 10.3).

С командами можно связать ряд дополнительных характеристик, в том числе *название* и одну или несколько *клавиш быстрого доступа*. Таким образом, для команд автоматически становятся доступны связанные с ними клавиши быстрого доступа.

Для связывания команды с определенным действием используется один из вариантов *механизма привязки*. Команды, ассоциированные со своими действиями, хранятся в коллекции CommandBindings, которая обычно создается для окна приложения и определяется в xaml-файле. Действие, которое должна выполнять данная команда, связывается с событием Executed (использованным в нашей программе). С командой можно также связать событие CanExecute, обработчик которого позволяет в любой момент определить, доступна ли данная команда для выполнения.

Заметим, что указывать атрибут x:Name при связывании команды с обработчиками событий необязательно, но удобно, так как он будет использован в именах автоматически генерируемых обработчиков событий.

Недочет. В названии команды «Закрыть» отсутствуют подчеркнутые символы-ускорители. Кроме того, для различных пунктов меню используются названия на разных языках, что нежелательно.

Исправление. Добавьте в описание команды exit1 в xaml-файле два свойства:

```
<MenuItem x:Name="exit1" Header="E_xit" InputGestureText="Alt+F4"
    Command="ApplicationCommands.Close" />
```

Результат. Теперь с командой закрытия связано английское название, имеющее символ-ускоритель, а рядом с названием указывается клавиша быстрого доступа – Alt+F4.

Комментарий

Для пунктов меню можно явным образом настраивать названия даже при связывании с ними команд. Со стандартной командой ApplicationCommands.Close не связана клавиша быстрого доступа, однако в ней и нет необходимости, так как для закрытия любого окна доступна комбинация Alt+F4. Для большей наглядности мы указали эту комбинацию в свойстве InputGestureText пункта меню.

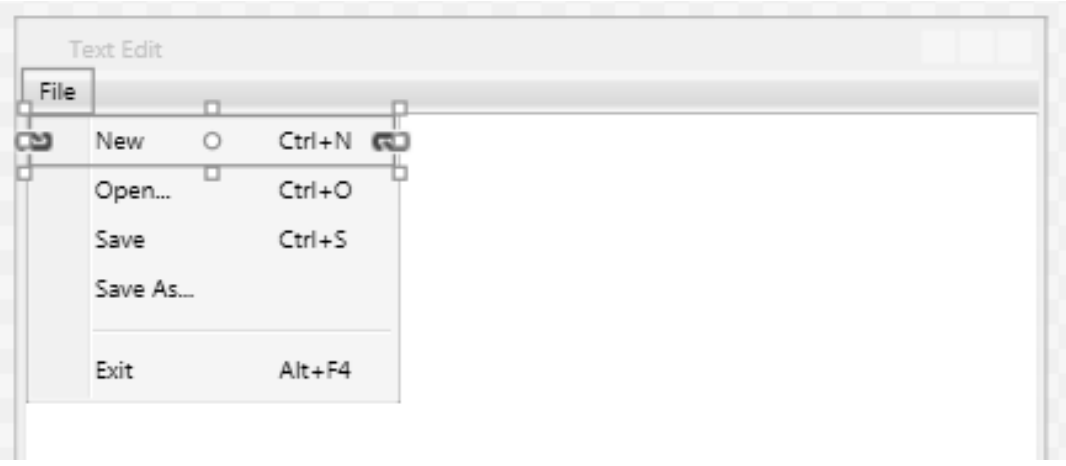


Рис. 34. Макет окна приложения TEXTEDIT версии 1 (второй вариант)

В заключение данного пункта добавим в меню File стандартные пункты для работы с файлами, связав их с соответствующими командами из класса ApplicationCommands (рис. 34).

```
<Window x:Class="TEXTEDIT.MainWindow"
    … >
  <Window.CommandBindings>
    <CommandBinding x:Name="new0"
        Command="ApplicationCommands.New"
        Executed="new0_Executed" />
    <CommandBinding x:Name="open0"
        Command="ApplicationCommands.Open"
        Executed="open0_Executed" />
    <CommandBinding x:Name="save0"
        Command="ApplicationCommands.Save"
        Executed="save0_Executed" />
    <CommandBinding x:Name="saveAs0"
        Command="ApplicationCommands.SaveAs"
        Executed="saveAs0_Executed" />
    <CommandBinding x:Name="exit0"
        Command="ApplicationCommands.Close"
        Executed="exit0_Executed" />
  </Window.CommandBindings>
  <DockPanel >
    <Menu DockPanel.Dock="Top" >
      <MenuItem x:Name="file1" Header="_File" >
        <MenuItem x:Name="new1" Header="_New"
            Command="ApplicationCommands.New" />
        <MenuItem x:Name="open1" Header="_Open..."
            Command="ApplicationCommands.Open" />
        <MenuItem x:Name="save1" Header="_Save"
            Command="ApplicationCommands.Save" />
        <MenuItem x:Name="saveAs1" Header="Save _As..."
            Command="ApplicationCommands.SaveAs" />
        <Separator />
        <MenuItem x:Name="exit1" Header="E_xit"
            InputGestureText="Alt+F4"
            Command="ApplicationCommands.Close" />
      </MenuItem>
    </Menu>
    …
  </DockPanel>
```

```
</Window>
```

Обработчики добавленных команд пока являются пустыми; они будут определены на следующих этапах проектирования приложения.

Обратите внимание на использование компонента Separator для изображения в группе пунктов меню горизонтальной черты, а также на добавление многоточия к названиям тех команд, выполнение которых связано с отображением диалоговых окон. Рядом со стандартными командами, имеющими клавиши быстрого доступа, отображаются названия этих клавиш. К сожалению, для стандартной команды SaveAs клавиатурной комбинации не предусмотрено.

Комментарий

> Если перед выполнением команды меню на экране возникает диалоговое окно, то в конце названия такой команды принято указывать многоточие. Наличие многоточия в названии команды, в частности, означает, что данную команду можно отменить уже после ее вызова, если закрыть связанное с ней диалоговое окно с помощью кнопки «Отмена» (Cancel) или клавиши Esc.

9.3. Сохранение текста в файле

К списку директив using в начале файла MainWindow.xaml.cs добавьте следующие директивы:

```
using System.IO;
using Microsoft.Win32;
```

В описание класса MainWindow добавьте новое поле

```
SaveFileDialog saveFileDialog1 = new SaveFileDialog();
```

и вспомогательный метод:

```
void SaveToFile(string path)
{
    File.WriteAllText(path, textBox1.Text, Encoding.Default);
}
```

В конструктор класса MainWindow добавьте оператор:

```
saveFileDialog1.Filter = "Text files (*.txt)|*.txt";
```

И определите обработчики save0_Executed и saveAs0_Executed, уже имеющиеся в классе MainWindow:

```
private void save0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    string path = saveFileDialog1.FileName;
    if (path == "")
        saveAs0_Executed(null, null);
    else
        SaveToFile(path);
```

```
}
private void saveAs0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    if (saveFileDialog1.ShowDialog() == true)
    {
        string path = saveFileDialog1.FileName;
        SaveToFile(path);
        Title = "Text Editor - " + path;
    }
}
```

Результат. При выполнении команды Save As возникает диалоговое окно «Сохранить как». Если указать в диалоговом окне имя файла и нажать кнопку «Сохранить» (или клавишу Enter), то введенный текст будет сохранен в этом файле, а полное имя файла появится в заголовке окна программы. По умолчанию имя файла снабжается расширением .txt. При выполнении команды Save имя файла не запрашивается, за исключением случая, когда текст еще ни разу не сохранялся.

При выходе из диалогового окна по нажатию кнопки «Отмена» (или клавиши Esc) сохранения текста в файле не происходит. При попытке сохранить текст под именем уже существующего файла выводится запрос на подтверждение данного действия. При указании несуществующего каталога выводится предупреждающее сообщение, причем диалоговое окно не закрывается, и ошибку можно немедленно исправить.

Комментарии

1. В библиотеке Windows Forms имеются особые *невизуальные компоненты* – SaveFileDialog и OpenFileDialog – для отображения стандартных диалогов сохранения и открытия файлов, однако в библиотеке WPF подобные компоненты отсутствуют (в данной библиотеке вообще не предусмотрена возможность работы с невизуальными компонентами). Однако в приложениях WPF можно использовать одноименные стандартные классы, определенные в пространстве имен Microsoft.Win32 и предоставляющие ту же функциональность, что и компоненты из библиотеки Windows Forms.

К числу основных свойств данных классов относятся следующие:

- FileName – имя файла, которое выводится в диалоговом окне при его отображении и в котором возвращается имя файла, выбранное пользователем (возвращается всегда полное имя файла);
- FileNames – *массив* выбранных имен файлов, который может содержать несколько элементов, если пользователь выбрал в диалоговом окне несколько файлов; при этом первый элемент массива содержит

то же имя, что и свойство FileName (заметим, что возможность выбора нескольких файлов имеется только у окна OpenFileDialog и только в том случае, если его свойство Multiselect равно true);

- InitialDirectory – каталог, который будет выведен при открытии диалогового окна в случае, если свойство FileName является пустым или содержит только имя файла без пути к нему (в настройке свойства InitialDirectory, как правило, нет необходимости, поскольку диалоговые окна реализованы таким образом, что автоматически сохраняют информацию о последнем каталоге, из которого были выбраны файлы, и при последующем открытии отображают именно этот каталог);
- Filter – строка с *фильтрами*, позволяющими отобрать группу файлов, отображаемую в диалоговом окне. Для каждого фильтра вначале указывается его описание, а затем сам фильтр в виде списка масок файлов через точку с запятой; описание отделяется от фильтра вертикальной чертой, этим же символом разделяются и различные фильтры;
- FilterIndex – индекс текущего фильтра (фильтры нумеруются от 1, по умолчанию устанавливается первый фильтр); при закрытии окна содержит информацию о выбранном пользователем фильтре;
- DefaultExt – расширение, которое автоматически добавляется к выбранному имени файла, если это имя не имеет расширения (следует, однако, иметь в виду, что если у текущего фильтра есть явно указанное расширение, то к файлам без расширения добавляется расширение из фильтра);
- Title – заголовок диалогового окна (если является пустым, то используется заголовок по умолчанию).

Некоторые свойства являются логическими (в скобках для этих свойств указываются значения по умолчанию, причем первое значение соответствует окну SaveFileDialog, а второе – окну OpenFileDialog):

- CheckFileExists (false, true) – при значении true разрешается выбирать только существующие файлы;
- CheckPathExists (true, true) – при значении true разрешается выбирать только файлы из существующих каталогов (файл при этом может не существовать);
- CreatePrompt (false, –) – данное свойство имеется только у класса SaveFileDialog; при значении true в случае отсутствия файла окно выводит запрос о его создании и возвращает выбранное имя только при положительном ответе на запрос (однако никаких действий по созданию файла не предпринимает);
- OvewritePrompt (true, –) – данное свойство имеется только у класса SaveFileDialog; при значении true в случае наличия файла окно выводит запрос о его перезаписи и возвращает выбранное имя только при

положительном ответе на запрос (однако никаких действий по перезаписи файла не предпринимает);

- Multiselect (–, false) – данное свойство имеется только у класса OpenFileDialog; при значении true в окне можно выбирать несколько файлов.

При сохранении файлов необходимо контролировать ситуации, связанные с попыткой перезаписи существующего файла и с попыткой указания несуществующего каталога. Этот контроль обеспечивается автоматически благодаря соответствующим значениям свойств OverwritePrompt и CheckPathExists, установленным по умолчанию.

2. Метод ShowDialog возвращает результат типа bool?, который может принимать три значения: true, false и null. Поэтому в условии оператора if приходится сравнивать возвращаемый результат с константой true.

3. Для записи данных в текстовый файл используется метод WriteAllText класса File из пространства имен System.IO. При сохранении текста в нем учитываются все маркеры конца строк. Последний, необязательный параметр метода определяет *формат кодирования* данных при записи. Использованный в нашей программе формат Encoding.Default обеспечивает сохранение текста в однобайтной ANSI-кодировке, принятой в Windows по умолчанию (для русской версии Windows это кодировка ANSI 1251 «Cyrillic (Windows)»). Если не указать формат кодирования, то файл будет содержать данные в кодировке UTF–8.

4. Несмотря на контроль особых ситуаций, предоставляемый диалоговым окном, при сохранении файла (а также при его открытии, которое будет реализовано в следующем пункте) может возникнуть много различных ошибок. Для их корректной обработки следует использовать операторы try–catch (примеры применения данных операторов при обработке файловых данных содержатся в описании проекта IMGVIEW, п. 17.1).

Недочет. При последующем сохранении командой Save As ранее сохраненного файла в поле ввода диалогового окна отображается *полный путь к этому файлу*, что усложняет его редактирование. Заметим, что в стандартных программах диалоговые окна *никогда* не отображают полные имена файлов в своих полях ввода.

Исправление. Добавьте в начало метода saveAs0_Executed следующий оператор:

```
saveFileDialog1.FileName =
  System.IO.Path.GetFileName(saveFileDialog1.FileName);
```

Результат. Теперь в диалоговом окне отображается только имя и расширение ранее сохраненного файла.

Комментарий

Необходимость в явном указании пространства имен System.IO для класса Path обусловлена тем, что такое же имя имеет класс

System.Windows.Shapes.Path из библиотеки WPF (другой способ исправления ошибок, связанных с конфликтами имен, описывается в проекте IMGVIEW, п. 17.2).

Ошибка. После сделанного исправления программа некорректно работает, если было вызвано диалоговое окно сохранения и пользователь закрыл его, *не выбрав файла*. В этой ситуации в свойстве saveFileDialog1.FileName сохраняется краткое имя файла (без пути к нему), и поэтому при последующем выполнении команды Save (в которой диалоговое окно не отображается) файл сохраняется не в своем исходном каталоге, а в текущем каталоге программы.

Исправление. Измените метод saveAs0_Executed следующим образом:

```
private void saveAs0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    var oldPath = saveFileDialog1.FileName;
    saveFileDialog1.FileName =
      System.IO.Path.GetFileName(saveFileDialog1.FileName);
      System.IO.Path.GetFileName(oldPath);
    if (saveFileDialog1.ShowDialog() == true)
    {
        string path = saveFileDialog1.FileName;
        SaveToFile(path);
        Title = "Text Edit - " + path;
    }
    else
        saveFileDialog1.FileName = oldPath;
}
```

Результат. Теперь отмеченная выше ошибка не возникает, поскольку при любом варианте работы метода saveAs0_Executed при выходе из него свойство saveFileDialog1.FileName содержит полное имя текущего файла.

9.4. Очистка области редактирования и открытие нового файла

В описание класса MainWindow добавьте новое поле:

```
OpenFileDialog openFileDialog1 = new OpenFileDialog();
```

В конструкторе класса MainWindow *дополните* последний оператор, указав в его начале новый фрагмент:

```
openFileDialog1.Filter =
  saveFileDialog1.Filter = "Text files (*.txt)|*.txt";
```

И определите обработчики new0_Executed и open0_Executed, уже имеющиеся в классе MainWindow:

```
private void new0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    textBox1.Clear();
    Title = "Text Edit";
    saveFileDialog1.FileName = "";
}
private void open0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    openFileDialog1.FileName = "";
    if (openFileDialog1.ShowDialog() == true)
    {
        string path = openFileDialog1.FileName;
        textBox1.Text = File.ReadAllText(path, Encoding.Default);
        Title = "Text Edit - " + path;
        saveFileDialog1.FileName = path;
    }
}
```

Результат. При выполнении команды New содержимое области редактирования очищается. При выполнении команды Open появляется диалоговое окно «Открыть», позволяющее выбрать файл для загрузки в окно редактирования. При попытке открыть несуществующий файл выдается предупреждающее сообщение, однако диалоговое окно не закрывается, и ошибку можно немедленно исправить.

Комментарии

1. В методе new0_Executed свойство saveFileDialog1.FileName полагается равным пустой строке. Это служит признаком того, что новый документ *еще не сохранен в файле* (для таких документов любое действие по их сохранению приведет к запросу имени файла). Напротив, в методе open0_Executed имя загруженного файла сохраняется в свойстве saveFileDialog1.FileName, что позволяет в дальнейшем не запрашивать имя файла при выполнении команды Save.

В начале метода open0_ Executed свойство openFileDialog1.FileName полагается равным пустой строке; это предотвращает отображение имени *уже загруженного файла* в поле ввода диалогового окна «Открыть».

2. Для чтения данных из текстового файла использован метод ReadAllText класса File, позволяющий прочесть все содержимое указанного файла в строку. Полученная строка включает *маркеры концов строк*, что позволяет правильно разбить прочитанный текст на отдельные строки при его отображении в компоненте textBox1.

При чтении данных, как и при их записи (см. предыдущий пункт), в нашей программе используется формат Encoding.Default.

3. Для корректной обработки особой ситуации, связанной с попыткой открытия несуществующего файла, необходимо, чтобы свойство CheckFileExists компонента openFileDialog1 было равно true (именно таким является его значение по умолчанию).

9.5. Контроль за сохранением изменений, внесенных в текст

```
<Window x:Class="TEXTEDIT.MainWindow"
    … Closing="Window_Closing" >
  …
  <DockPanel >
    …
    <TextBox x:Name="textBox1" Text="" AcceptsReturn="True"
        AcceptsTab="True" VerticalScrollBarVisibility="Auto"
        HorizontalScrollBarVisibility="Auto"
        TextChanged="textBox1_TextChanged" />
  </DockPanel>
</Window>
```

В описание класса MainWindow добавьте новое свойство

```
bool Modified { get; set; }
```

и новый вспомогательный метод:

```
bool TextSaved()
{
    if (Modified)
        switch (MessageBox.Show("Сохранить изменения в файле?",
          "Подтверждение", MessageBoxButton.YesNoCancel,
          MessageBoxImage.Question))
        {
            case MessageBoxResult.Yes:
                save0_Executed(null, null);
                return !Modified;
            case MessageBoxResult.Cancel:
                return false;
        }
    return true;
}
```

В конец метода SaveToFile добавьте оператор:

```
Modified = false;
```

Измените методы new0_Executed и open0_Executed следующим образом:

```
private void new0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    if (TextSaved())
    {
        textBox1.Clear();
        Title = "Text Edit";
        saveFileDialog1.FileName = "";
        Modified = false;
    }
}
private void open0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    if (TextSaved())
    {
        openFileDialog1.FileName = "";
        if (openFileDialog1.ShowDialog() == true)
        {
            string path = openFileDialog1.FileName;
            textBox1.Text = File.ReadAllText(path,
              Encoding.Default);
            Title = "Text Edit - " + path;
            saveFileDialog1.FileName = path;
            Modified = false;
        }
    }
}
```

И определите обработчики textBox1_TextChanged и Window_Closing, указанные в xaml-файле:

```
private void textBox1_TextChanged(object sender,
  TextChangedEventArgs e)
{
    Modified = true;
}
private void Window_Closing(object sender,
  System.ComponentModel.CancelEventArgs e)
{
    e.Cancel = !TextSaved();
```

```
}
```

Результат. Если в текущий текст были внесены изменения, то попытка выполнить команды New, Open или Exit приводит к появлению запроса о том, следует ли сохранять на диске эти изменения. При нажатии кнопки «Да» (Yes) текущий текст сохраняется под прежним именем, а если он ни разу не был сохранен, то его имя запрашивается в диалоговом окне «Сохранить как». При нажатии кнопки «Нет» (No) измененное содержимое области редактирования не сохраняется. При нажатии кнопки «Отмена» (Cancel) выбранная команда (New, Open или Exit) отменяется, и пользователь может продолжать редактирование текущего текста.

Комментарии

1. Для отслеживания того, изменялось ли содержимое компонента TextBox, в библиотеке Windows Forms у этого компонента было предусмотрено свойство Modified. Компонент TextBox из библиотеки WPF не имеет этого свойства, однако аналогичное свойство можно включить непосредственно в класс окна приложения. Данное свойство принимает значение true, если в текст были внесены изменения *пользователем*. При сохранении текста в файле, а также при загрузке файла или создании нового документа свойство Modified сбрасывается в значение false.

Поскольку ни при записи, ни при чтении свойства Modified не требуется предусматривать особых действий, это свойство сделано *автоматическим*. Методы-аксессоры для автоматических свойств определяются с применением простейшего синтаксиса { get; set; }; при этом сам компилятор создает вспомогательное закрытое поле и методы-аксессоры, которые обеспечивают чтение и изменение значения этого закрытого поля при чтении и записи соответствующего автоматического свойства.

2. Функция TextSaved возвращает true, если текущий текст был сохранен на диске или пользователь явно отказался от сохранения, выбрав вариант «Нет». Если был выбран вариант «Отмена», то функция возвращает false (это означает, что пользователь желает вернуться к редактированию текущего текста). Обратите внимание на оператор

```
return !Modified;
```

который обеспечивает корректную обработку следующей ситуации: пользователь ранее не сохранял текст в файле, при появлении запроса на сохранение текста он выбрал вариант «Да», но вышел из появившегося диалогового окна «Сохранение файла», нажав кнопку «Отмена» (т. е. *не сохранил* текущий текст). В этом случае функция TextSaved вернет значение false, поскольку отказ пользователя от действий по сохранению файла в данной ситуации, вероятнее всего, означает, что он хочет вернуться к его редактированию.

9.6. Проверка доступности команд WPF

```
<Window.CommandBindings>
  …
  <CommandBinding x:Name="save0"
      Command="ApplicationCommands.Save"
      Executed="save0_Executed"
      CanExecute="save0_CanExecute" />
  …
</Window.CommandBindings>
```

```
private void save0_CanExecute(object sender,
  CanExecuteRoutedEventArgs e)
{
    e.CanExecute = Modified;
}
```

Результат. Если только что был создан новый пустой документ или если текст, загруженный в редактор, не изменялся пользователем или уже был сохранен, то команда Save недоступна для выполнения (в чем можно убедиться, развернув меню File: эта команда отображается светло-серым цветом и ее нельзя выбрать).

Комментарий

Обратите внимание на то, что в программе проверяется доступность не пункта меню, а *команды WPF*. Если недоступная команда связана с несколькими интерфейсными компонентами, то все такие компоненты автоматически становятся недоступными (мы продемонстрируем это далее, когда в версии 4 нашей программы добавим в нее панель инструментов).

10. Дополнительные возможности меню, настройка шрифта, выравнивания и цвета: TEXTEDIT, версия 2

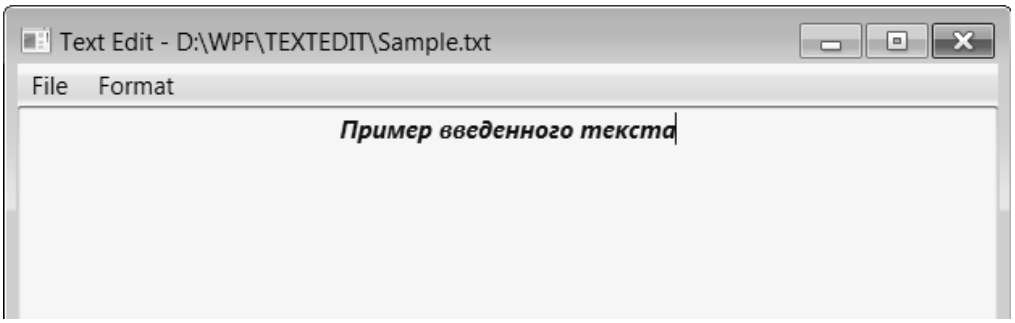


Рис. 35. Верхняя часть окна приложения TEXTEDIT версии 2

10.1. Установка начертания символов. Команды меню – флажки

```
<Window x:Class="TEXTEDIT.MainWindow"
    … PreviewKeyDown="Window_PreviewKeyDown" >
  …
  <DockPanel >
    <Menu DockPanel.Dock="Top" >
      <MenuItem x:Name="file1" Header="_File" >
        …
      </MenuItem>
      <MenuItem x:Name="format1" Header="_Format" >
        <MenuItem x:Name="bold1" Header="_Bold"
            InputGestureText="Ctrl+B" Click="bold1_Click" />
        <MenuItem x:Name="italic1" Header="_Italic"
            InputGestureText="Ctrl+I" Click="italic1_Click" />
        <MenuItem x:Name="underline1" Header="_Underline"
            InputGestureText="Ctrl+U" Click="underline1_Click" />
      </MenuItem>
    </Menu>
    …
  </DockPanel>
</Window>
```

Определите в классе MainWindow новые обработчики, указанные в xaml-файле:

```
private void bold1_Click(object sender, RoutedEventArgs e)
{
    bold1.IsChecked = !bold1.IsChecked;
    textBox1.FontWeight = bold1.IsChecked ? FontWeights.Bold :
      FontWeights.Normal;
}
private void italic1_Click(object sender, RoutedEventArgs e)
{
    italic1.IsChecked = !italic1.IsChecked;
    textBox1.FontStyle = italic1.IsChecked ? FontStyles.Italic :
      FontStyles.Normal;
}
private void underline1_Click(object sender, RoutedEventArgs e)
{
    underline1.IsChecked = !underline1.IsChecked;
    textBox1.TextDecorations = underline1.IsChecked ?
      TextDecorations.Underline : null;
}

private void Window_PreviewKeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyboardDevice.Modifiers == ModifierKeys.Control)
        switch (e.Key)
        {
            case Key.B:
                bold1_Click(null, null);
                e.Handled = true;
                break;
            case Key.I:
                italic1_Click(null, null);
                e.Handled = true;
                break;
            case Key.U:
                underline1_Click(null, null);
                e.Handled = true;
                break;
        }
}
```

Результат. При выполнении добавленных в меню команд устанавливается соответствующий *стиль* (начертание) символов в редакторе: «Bold» – **полужирный**, «Italic» – *курсив*, «Underline» – подчеркнутый. При

вызове меню «Format» около команд с установленными стилями изображаются «галочки» (как в случае флажков-переключателей). При повторном выполнении команды соответствующий стиль отменяется и «галочка» исчезает. Поскольку в нашем редакторе форматные настройки не сохраняются вместе с текстом, выполнение команд форматирования не влияет на значение свойства Modified.

Комментарии

1. Для описанных в этом пункте действий отсутствуют стандартные команды WPF. Подобные команды можно определить самостоятельно, и в дальнейшем мы рассмотрим такую возможность, а пока мы реализовали данные пункты меню в традиционном стиле, с применением обработчиков событий.

2. Нам пришлось определить обработчик события PreviewKeyDown для окна, позволяющий перехватывать нажатия клавиш и обрабатывать те комбинации, которые связаны с командами меню. Для того чтобы определить, входят ли в клавиатурную комбинацию клавиши-модификаторы, используется свойство e.KeyboardDevice.Modifiers. Следует отметить, что приведенное в программе условие e.KeyboardDevice.Modifiers == ModifierKeys.Control будет истинным только в том случае, если из модифицирующих клавиш нажата *только* Ctrl. Это удобно, так как в дальнейшем нам может понадобиться особым образом обрабатывать более сложные комбинации, например Ctrl+Shift+B.

3. У пунктов меню имеется свойство IsCheckable, позволяющее при выборе пункта меню *автоматически* переключать его состояние из установленного в снятое и обратно. Для подобного поведения необходимо положить это свойство равным true. По умолчанию свойство IsCheckable равно false; в этом случае при вызове команды ее состояние остается прежним и для его изменения необходимо выполнить явные действия. Мы сохранили для свойства IsCheckable значение по умолчанию. Заметим, что если бы мы установили для этого свойства значение true, то выполнять дополнительные действия по переключению состояния пункта меню нам пришлось бы при выполнении команд *с помощью клавиш быстрого доступа*.

4. Все три использованные настройки шрифта в библиотеке WPF связаны с различными свойствами визуальных компонентов – FontWeight для полужирного выделения, FontStyle для курсивного выделения и TextDecorations для подчеркивания. Это отличается от подхода, использованного в библиотеке Windows Forms, в которой все перечисленные настройки шрифта определяются в свойстве Style класса Font.

10.2. Установка выравнивания текста. Команды меню – радиокнопки

```
<Window x:Class="TEXTEDIT.MainWindow"
    … >
  …
  <DockPanel >
    <Menu DockPanel.Dock="Top" >
      <MenuItem x:Name="file1" Header="_File" >
        …
      </MenuItem>
      <MenuItem x:Name="format1" Header="_Format" >
        <MenuItem x:Name="bold1" Header="_Bold"
            InputGestureText="Ctrl+B" Click="bold1_Click" />
        <MenuItem x:Name="italic1" Header="_Italic"
            InputGestureText="Ctrl+I" Click="italic1_Click" />
        <MenuItem x:Name="underline1" Header="_Underline"
            InputGestureText="Ctrl+U" Click="underline1_Click" />
        <Separator/>
        <MenuItem x:Name="leftJustify1" Header="_Left justify"
            InputGestureText="Ctrl+L"
            Click="leftJustify1_Click" />
        <MenuItem x:Name="center1" Header="_Center"
            InputGestureText="Ctrl+E"
            Click="leftJustify1_Click" />
        <MenuItem x:Name="rightJustify1" Header="_Right justify"
            InputGestureText="Ctrl+R"
            Click="leftJustify1_Click" />
      </MenuItem>
    </Menu>
    …
  </DockPanel>
</Window>
```

В класс MainWindow добавьте новое поле:

```
MenuItem alignItem;
```

В конструктор класса MainWindow добавьте новые операторы:

```
alignItem = leftJustify1;
leftJustify1.IsChecked = true;
leftJustify1.Tag = HorizontalAlignment.Left;
center1.Tag = HorizontalAlignment.Center;
rightJustify1.Tag = HorizontalAlignment.Right;
```

Определите в классе MainWindow новый обработчик leftJustify1_Click (этот обработчик указан во всех трех добавленных пунктах меню):

```
private void leftJustify1_Click(object sender, RoutedEventArgs e)
{
    MenuItem mi = sender as MenuItem;
    if (mi.IsChecked)
        return;
    alignItem.IsChecked = false;
    alignItem = mi;
    mi.IsChecked = true;
    textBox1.HorizontalContentAlignment =
      (HorizontalAlignment)mi.Tag;
}
```

В оператор switch обработчика Window_PreviewKeyDown добавьте новые варианты клавиатурных комбинаций:

```
case Key.L:
    leftJustify1_Click(leftJustify1, null);
    e.Handled = true;
    break;
case Key.E:
    leftJustify1_Click(center1, null);
    e.Handled = true;
    break;
case Key.R:
    leftJustify1_Click(rightJustify1, null);
    e.Handled = true;
    break;
```

Результат. При выполнении добавленных в меню команд устанавливается соответствующее *выравнивание* текста в редакторе: «Left justify» – по левому краю, «Center» – по центру, «Right justify» – по правому краю. При вызове меню Format около команды с текущим выравниванием изображается «галочка».

Комментарии

1. К сожалению, в отличие от библиотеки Windows Forms, вид метки рядом с пунктом меню изменять нельзя: это всегда «галочка» (в Windows Forms можно было так настроить пункт меню, чтобы рядом с ним рисовался маркер •, что лучше соответствовало бы смыслу команд, ведущих себя как радиокнопки).

2. Установка метки около нового пункта меню не приводит к автоматическому сбросу старой метки в этой группе пунктов. Чтобы упростить действия по сбросу предыдущей метки, мы добавили в класс MainWindow

поле alignItem, в котором хранится текущий помеченный пункт меню, связанный с выравниванием. Если выбран другой вариант выравнивания, то с ранее помеченного пункта метка снимается, на новый пункт метка устанавливается, и этот пункт сохраняется в поле alignItem.

3. Для команд меню, связанных с выравниванием, в отличие от команд настройки свойств шрифта, нам удалось реализовать общий обработчик. Для этого мы воспользовались свойством Tag, записав в него для каждого пункта меню то значение выравнивания (из перечисления HorizontalAlignment), которое соответствует этому пункту. Так как в xaml-файле свойству Tag можно присвоить только значения *строкового типа*, указанные присваивания выполняются в cs-файле (в конструкторе окна). В методе leftJustify1_Click значение свойства Tag выбранной команды присваивается свойству HorizontalContentAlignment компонента textBox1; при этом выполняется явное преобразование свойства Tag к типу перечисления HorizontalAlignment.

10.3. Установка цвета символов и фона. Определение новых команд WPF и использование диалогового окна из библиотеки Windows Forms

Для действий по выбору цвета шрифта и цвета фона отсутствуют стандартные команды WPF, однако их несложно реализовать самостоятельно. Так как действия по настройке пунктов меню «традиционным» способом (основанным на обработчиках событий) мы уже описали ранее в этой версии программы TEXTEDIT, для новых пунктов меню выберем второй подход, связав их с командами WPF, которые предварительно определим.

Для определения новых команд надо создать новый класс. Оформим его в виде отдельного файла.

Выполните команду Project | Add Class…, в появившемся диалоговом окне введите имя класса FormatCommands и нажмите кнопку Add или клавишу Enter. Будет создана заготовка файла FormatCommands.cs, в которую надо ввести новые операторы. Приведем окончательный вид файла FormatCommands.cs, в котором полужирным шрифтом выделены добавленные операторы:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Input;

namespace TEXTEDIT
```

```
{
class FormatCommands
{
    private static RoutedUICommand fontcol;
    private static RoutedUICommand backcol;
    static FormatCommands()
    {
        InputGestureCollection inputs = new
          InputGestureCollection();
        inputs.Add(new KeyGesture(Key.F, ModifierKeys.Control |
          ModifierKeys.Shift, "Ctrl+Shift+F"));
        fontcol = new RoutedUICommand("Font Color...",
          "FontColor", typeof(FormatCommands), inputs);
        inputs = new InputGestureCollection();
        inputs.Add(new KeyGesture(Key.B, ModifierKeys.Control |
          ModifierKeys.Shift, "Ctrl+Shift+B"));
        backcol = new RoutedUICommand("Back Color...",
          "BackColor", typeof(FormatCommands), inputs);
    }
    public static RoutedUICommand FontColor
    {
        get { return fontcol; }
    }
    public static RoutedUICommand BackColor
    {
        get { return backcol; }
    }
}
}
```

При определении команды указывается ее подпись, имя, тип и набор клавиш быстрого доступа (а при определении этого набора также задается и текст, связанный с клавишами).

После этого необходимо *зарегистрировать* новые команды в окне нашей программы и связать их с обработчиками. Это делается в xaml-файле в элементе Window.CommandBindings:

```
<Window.CommandBindings>
  …
  <CommandBinding x:Name="fontColor0"
      Command="local:FormatCommands.FontColor"
      Executed="fontColor0_Executed" />
  <CommandBinding x:Name="backColor0"
```

```
      Command="local:FormatCommands.BackColor"
      Executed="backColor0_Executed" />
</Window.CommandBindings>
```

Обратите внимание на префикс local, указанный перед именами команд. Этот префикс был ранее определен в xaml-файле следующим образом:

```
xmlns:local="clr-namespace:TEXTEDIT"
```

Он позволяет указать *пространство имен*, в котором следует искать указанные классы. Если в xaml-файле требуется использовать другие нестандартные пространства имен, то для них тоже можно определить префиксы.

Заметим, что при описанном выше дополнении xaml-файла текст атрибутов Command может быть выделен как ошибочный (синим подчеркиванием), однако это не будет препятствовать успешной компиляции проекта. После такой компиляции подчеркивание должно исчезнуть.

К списку пунктов меню надо добавить следующие элементы:

```
<MenuItem x:Name="format1" Header="_Format" >
  <MenuItem x:Name="bold1" Header="_Bold" … />
  …
  <MenuItem x:Name="rightJustify1" … />
  <Separator/>
  <MenuItem x:Name="color1" Header="_Color" >
    <MenuItem x:Name="fontColor1"
        Command="local:FormatCommands.FontColor" />
    <MenuItem x:Name="backColor1"
        Command="local:FormatCommands.BackColor" />
  </MenuItem>
</MenuItem>
```

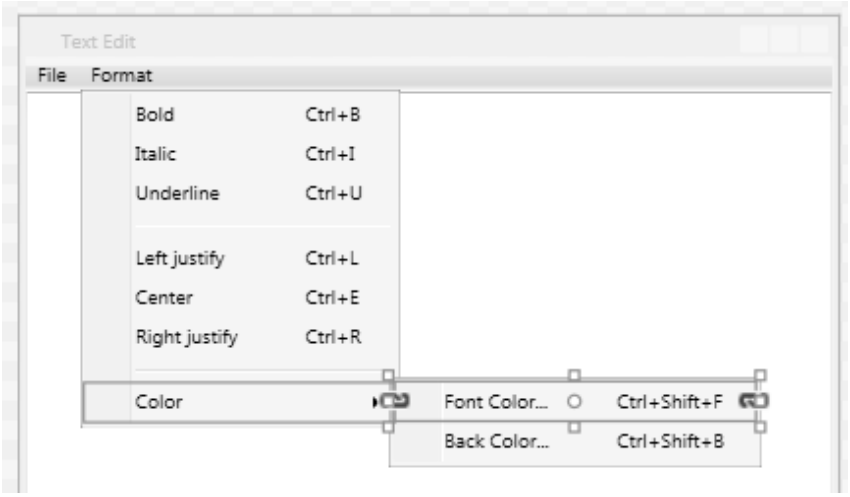


Рис. 36. Макет окна приложения TEXTEDIT версии 2 (окончательный вариант)

Таким образом, мы создали меню третьего уровня (см. рис. 36). Поскольку все настройки сделаны в самих командах WPF, при определении пунктов меню было достаточно только указать эти команды.

Осталось обратиться к файлу MainWindow.xaml.cs и определить обработчики для новых команд. Однако здесь нас ожидают дополнительные проблемы.

Дело в том, что в библиотеке WPF отсутствует не только компонент, обеспечивающий вызов диалогового окна для выбора цвета, но и обычный класс для выполнения такого действия. Поэтому, если мы не хотим вручную создавать подобное диалоговое окно, придется воспользоваться диалоговым окном выбора цвета из библиотеки Windows Forms.

Подключите к проекту необходимые компоненты библиотеки Windows Forms (System.Windows.Forms и System.Drawing), действуя так, как описано в п. 8.5 проекта CURSORS. Напомним, что указывать соответствующие пространства имен в директивах using не следует, чтобы избежать конфликтов совпадающих имен для классов из библиотек Windows Forms и WPF.

Теперь в нашей программе можно использовать класс ColorDialog из библиотеки Windows Forms. Но перед этим надо решить еще одну проблему, связанную с тем, что этот класс работает с классом Color из библиотеки Windows Forms, а компоненты WPF – с классом Color из библиотеки WPF, и эти классы несовместимы. Поэтому в классе MainWindow необходимо реализовать два вспомогательных метода-*конвертера*, преобразующих один класс Color в другой. Эти методы выглядят следующим образом:

```
Color WinFormsColorToWPFColor(System.Drawing.Color c)
{
    return Color.FromArgb(c.A, c.R, c.G, c.B);
}

System.Drawing.Color WPFColorToWinFormsColor(Color c)
{
    return System.Drawing.Color.FromArgb(c.A, c.R, c.G, c.B);
}
```

Осталось добавить в класс MainWindow новое поле

```
System.Windows.Forms.ColorDialog colorDialog1 =
  new System.Windows.Forms.ColorDialog();
```

и определить обработчики событий для новых команд (заготовки этих обработчиков уже содержатся в описании класса MainWindow):

```
private void fontColor0_Executed(object sender,
  ExecutedRoutedEventArgs e)
```

```
{
    colorDialog1.Color =
      WPFColorToWinFormsColor((textBox1.Foreground as
      SolidColorBrush).Color);
    if (colorDialog1.ShowDialog() ==
      System.Windows.Forms.DialogResult.OK)
        textBox1.Foreground = new
          SolidColorBrush(WinFormsColorToWPFColor(colorDialog1
          .Color));
}
private void backColor0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    colorDialog1.Color =
      WPFColorToWinFormsColor((textBox1.Background as
      SolidColorBrush).Color);
    if (colorDialog1.ShowDialog() ==
      System.Windows.Forms.DialogResult.OK)
        textBox1.Background = new
          SolidColorBrush(WinFormsColorToWPFColor(colorDialog1
          .Color));
}
```

Результат. При выполнении команды из группы Colors вызывается диалоговое окно «Цвет», позволяющее установить цвет символов (команда Font color) или цвет фона (команда Background color). При отображении диалогового окна в нем сплошной рамкой обводится текущий цвет. При выборе нового цвета и нажатии кнопки OK происходит изменение цвета в редакторе.

Комментарий

Описанный прием импортирования компонентов библиотеки Windows Forms можно использовать для получения в WPF-приложении диалогового окна *настройки шрифта*, которое имеется в Windows Forms и отсутствует в WPF. Правда, обеспечить конвертирование классов, связанных со шрифтами в этих библиотеках, будет гораздо сложнее.

11. Команды редактирования и контекстное меню: TEXTEDIT, версия 3

11.1. Команды редактирования

Рис. 37. Макет окна приложения TEXTEDIT версии 3

```
<Window x:Class="TEXTEDIT.MainWindow"
    … >
  <Window.CommandBindings>
    …
    <CommandBinding x:Name="undo0"
        Command="ApplicationCommands.Undo"
        Executed="undo0_Executed"
        CanExecute="undo0_CanExecute" />
    <CommandBinding x:Name="cut0"
        Command="ApplicationCommands.Cut"
        Executed="cut0_Executed" CanExecute="cut0_CanExecute" />
    <CommandBinding x:Name="copy0"
        Command="ApplicationCommands.Copy"
        Executed="copy0_Executed" CanExecute="cut0_CanExecute" />
    <CommandBinding x:Name="paste0"
        Command="ApplicationCommands.Paste"
        Executed="paste0_Executed"
        CanExecute="paste0_CanExecute" />
    <CommandBinding x:Name="selectAll0"
        Command="ApplicationCommands.SelectAll"
        Executed="selectAll0_Executed"
```

```
        CanExecute="selectAll0 CanExecute" />
  </Window.CommandBindings>
  <DockPanel >
    <Menu DockPanel.Dock="Top" >
      <MenuItem x:Name="file1" Header="_File" >
        …
      </MenuItem>
      <MenuItem x:Name="edit1" Header="_Edit" >
        <MenuItem x:Name="undo1" Header="_Undo"
            Command="ApplicationCommands.Undo" />
        <Separator/>
        <MenuItem x:Name="cut1" Header="Cu_t"
            Command="ApplicationCommands.Cut"/>
        <MenuItem x:Name="copy1" Header="_Copy"
            Command="ApplicationCommands.Copy"/>
        <MenuItem x:Name="paste1" Header="_Paste"
            Command="ApplicationCommands.Paste" />
        <Separator/>
        <MenuItem x:Name="selectAll1" Header="_Select All"
            Command="ApplicationCommands.SelectAll"/>
      </MenuItem>
      <MenuItem x:Name="format1" Header="_Format" >
        …
      </MenuItem>
    </Menu>
    …
  </DockPanel>
</Window>
```

При задании обработчиков событий для команд WPF в xaml-файле обратите внимание на то, что для команд cut0 и copy0 используется *один и тот же обработчик* для события CanExecuted: cut0_CanExecute.

Определите в классе MainWindow новые обработчики, указанные в xaml-файле:

```
private void undo0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    textBox1.Undo();
}
private void cut0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
```

```
    textBox1.Cut();
}
private void copy0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    textBox1.Copy();
}
private void paste0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    textBox1.Paste();
}
private void selectAll0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    textBox1.SelectAll();
}
private void undo0_CanExecute(object sender,
  CanExecuteRoutedEventArgs e)
{
    e.CanExecute = textBox1.CanUndo;
}
private void cut0_CanExecute(object sender,
  CanExecuteRoutedEventArgs e)
{
    e.CanExecute = textBox1.SelectedText.Length > 0;
}
private void paste0_CanExecute(object sender,
  CanExecuteRoutedEventArgs e)
{
    e.CanExecute = Clipboard.ContainsText();
}
private void selectAll0_CanExecute(object sender,
  CanExecuteRoutedEventArgs e)
{
    e.CanExecute = textBox1.Text.Length > 0;
}
```

Результат. В меню определены стандартные *команды редактирования*: отмена последнего действия (Undo), вырезание (Cut) и копирование (Copy) выделенного фрагмента в буфер, вставка фрагмента из буфера

(Paste), выделение всего текста (Select All). При вызове подменю группы Edit недоступные команды выделяются серым цветом.

Комментарии

1. Для всех стандартных команд редактирования предусмотрены команды WPF, реализованные в классе ApplicationCommands и снабженные клавиатурными комбинациями. Нам осталось связать эти команды с действиями по редактированию, которые реализованы в компоненте TextBox в виде методов.

2. Особенно просто реализовать действия по отмене предыдущего ввода, поскольку в компоненте TextBox имеются и метод Undo, и метод CanUndo. Отметим также свойство UndoLimit, определяющее *глубину* отмены действий. По умолчанию оно равно 100.

3. При проверке доступности команд Cut и Copy анализируется выделенный фрагмент текста, доступный в компоненте TextBox с помощью свойства SelectedText (ранее свойства компонента TextBox, связанные с выделением фрагментов текста, обсуждались в проекте TEXTBOXES).

4. При проверке доступности команды Paste анализируется содержимое буфера обмена. Для проверки наличия в буфере обмена текстового фрагмента мы воспользовались статическим методом ContainsText класса Clipboard из пространства имен System.Windows. Наряду со *специальными методами*, позволяющими проверить наличие в буфере обмена основных видов данных и получить эти данные (ContainsText и GetText, ContainsAudio и GetAutioStream, ContainsImage и GetImage, ContainsFileDropList и GetFileDropList), класс Clipboard содержит несколько *универсальных методов*, в частности, метод GetDataObject, возвращающий объект интерфейсного типа IDataObject, который позволяет проверить наличие данных *любого* требуемого типа в буфере обмена с помощью логической функции GetDataPresent, а также получить данные с помощью функции GetData, возвращающей значение типа object. Обе функции имеют несколько перегруженных вариантов, в том числе вариант с параметром типа Type, определяющим требуемый тип данных. Заметим, что буфер обмена может хранить *любые объекты* .NET (например, кнопки), однако получить подобный объект может только то .NET-приложение, которое поместило его в буфер (в то же время строки и рисунки, находящиеся в буфере обмена, доступны всем приложениям).

Следует учитывать, что буфер обмена может одновременно хранить данные разных форматов, поэтому из того факта, что метод ContainsText вернул true, не следует, что метод ContainsImage обязательно вернет false.

Для помещения элемента данных в буфер обмена предусмотрен ряд методов класса Clipboard, в частности, статический метод SetDataObject. В качестве первого параметра этого метода надо указать объект, содержащий требуемый элемент данных (например строку). В буфер всегда

помещается копия элемента данных; при этом из буфера удаляются «старые» данные *того же типа*. Метод SetDataObject может иметь второй параметр типа bool. Если этот параметр равен true, то элемент данных сохранится в буфере обмена и после завершения работы того приложения, которое поместило его в буфер. Понятно, что так следует поступать только для данных, «понимаемых» всеми приложениями, например строк или рисунков. Если второй параметр равен false или отсутствует, то при завершении работы приложения элемент данных будет удален из буфера обмена.

Имеется также возможность размещать в буфере обмена *несколько представлений* элемента данных (например, программа, помещающая в буфер текст в формате RTF, может одновременно поместить в него «простой» текст, не содержащий форматирования). Можно получить варианты форматов, в которых данные содержатся в буфере, и загрузить именно тот вариант, который требуется. Для этих целей функции GetDataPresent и GetData снабжены перегруженным вариантом, принимающим в качестве параметра текстовую строку с описанием требуемого формата (например «Rich Text Format»). Все стандартные форматы представлены в классе DataFormats в виде строковых свойств, доступных только для чтения.

Недочет. Эксперименты, проведенные с программой, показывают, что обработчик selectAll0_CanExecute *никогда не вызывается*. Соответственно, команда Select All всегда доступна (даже в случае, когда компонент TextBox не содержит никакого текста). Видимо, разработчики WPF решили, что невозможны ситуации, когда команда выделения всех данных должна быть недоступной. Можно или примириться с этим обстоятельством, или определить собственную команду WPF для выделения всех данных, которая будет обрабатывать событие CanExecute. В случае выбора второго варианта действий его реализацию мы предоставляем читателю.

11.2. Создание контекстного меню

Для подключения к полю ввода контекстного меню проще всего выбрать в окне Properties для компонента TextBox свойство ContextMenu и нажать находящуюся рядом с ним кнопку New. В xaml-файле будет создана следующая заготовка для этого меню:

```
<TextBox x:Name="textBox1" … >
  <TextBox.ContextMenu>
    <ContextMenu />
  </TextBox.ContextMenu>
</TextBox>
```

После этого надо добавить пункты меню в качестве дочерних элементов элемента ContextMenu (как мы это делали ранее для основного меню).

Напомним, что для превращения комбинированного тега <ContextMenu /> в парный достаточно стереть символы «/>» в конце этого тега и повторно ввести символ «>».

```
<TextBox x:Name="textBox1" >
  <TextBox.ContextMenu>
    <ContextMenu>
      <MenuItem Header="Cu_t" Command="ApplicationCommands.Cut"/>
      <MenuItem Header="_Copy"
          Command="ApplicationCommands.Copy"/>
      <MenuItem Header="_Paste"
          Command="ApplicationCommands.Paste"/>
      <Separator/>
      <MenuItem Header="_Select All"
          Command="ApplicationCommands.SelectAll"/>
      <Separator/>
      <MenuItem Command="local:FormatCommands.FontColor"/>
      <MenuItem Command="local:FormatCommands.BackColor"/>
    </ContextMenu>
  </TextBox.ContextMenu>
</TextBox>
```

Результат. При нажатии правой кнопки мыши в области редактирования или комбинации Shift+F10 вместо стандартного контекстного меню компонента TextBox (содержащего команды для работы с буфером «Вырезать», «Копировать», «Вставить») отображается контекстное меню, определенное в свойстве ContextMenu данного компонента. Недоступные команды контекстного меню выделяются серым цветом.

Таким образом, мы реализовали возможность использования собственного контекстного меню, не написав ни одной строки программного кода.

12. Панель инструментов: TEXTEDIT, версия 4

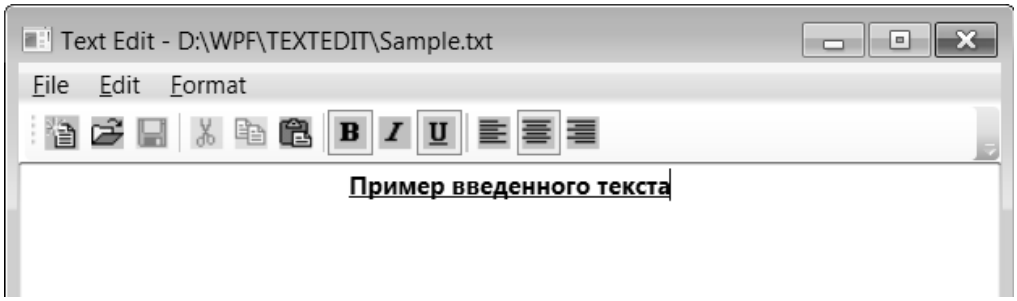


Рис. 38. Верхняя часть окна приложения TEXTEDIT версии 4

12.1. Создание панели инструментов. Добавление изображений к пунктам меню

Перед добавлением в наш проект панели инструментов необходимо включить в него графические файлы с изображениями, которые будут выводиться на кнопках панели инструментов. Эти файлы следует оформить как *встроенные ресурсы приложения* (действия по добавлению в приложение ресурсов были описаны в проекте CURSORS).

Графические файлы с изображениями, соответствующими стандартным командам для работы с файлами, обработки фрагментов текста и форматирования, можно найти в Интернете. При выборе формата файлов следует отдавать предпочтение формату png с прозрачным фоном, так как средства настройки прозрачного цвета для файлов bmp (подобные свойствам TransparentKey или ImageTransparentColor из библиотеки Windows Forms) в библиотеке WPF отсутствуют. Впрочем, можно использовать и непрозрачные изображения, если их фон является достаточно нейтральным (например, имеет серый цвет).

Будем считать, что у нас уже есть bmp-файлы с именами (в алфавитном порядке) Bold.bmp, Center.bmp, Copy.bmp, Cut.bmp, Italic.bmp, Left.bmp, New.bmp, Open.bmp, Paste.bmp, Right.bmp, Save.bmp, Underline.bmp, соответствующие всем действиям, с которыми мы хотим связать кнопки панели инструментов. Выполните для этих файлов действия по их подключению к приложению в виде ресурсов (см. проект CURSORS, п. 8.3). Заметим, что в диалоговом окне выбора файлов можно сразу указать *все* подключаемые файлы. Убедитесь, что свойство Build Action для добавленных файлов имеет значение Resource.

Комментарий

Вместо описанных в п. 8.3 действий, позволяющих добавить один или несколько файлов в приложение в качестве встроенных ресурсов, доста-

точно соответствующим образом отредактировать файл проекта TEXTEDIT.csproj еще до его загрузки в среду Visual Studio. Для этого после раздела

```
<ItemGroup>
  <None Include="App.config" />
</ItemGroup>
```

надо добавить раздел с указанием добавляемых файлов. Например, в нашем случае добавляемый раздел должен иметь вид:

```
<ItemGroup>
  <Resource Include="Center.bmp" />
  <Resource Include="Left.bmp" />
  <Resource Include="Right.bmp" />
  <Resource Include="Save.bmp" />
  <Resource Include="Open.bmp" />
  <Resource Include="New.bmp" />
  <Resource Include="Paste.bmp" />
  <Resource Include="Cut.bmp" />
  <Resource Include="Copy.bmp" />
  <Resource Include="Bold.bmp" />
  <Resource Include="Italic.bmp" />
  <Resource Include="Underline.bmp" />
</ItemGroup>
```

Кроме того, необходимо скопировать добавляемые файлы в каталог проекта (в котором содержится csproj-файл). Данный способ также позволяет быстро откорректировать имена имеющихся ресурсов, удалить их или добавить новые.

После определения встроенных графических ресурсов мы можем приступить к созданию макета панели инструментов. Вначале включим в нее только кнопки, связанные с командами группы File и Edit (рис. 39; после определения первой кнопки удобно копировать ее содержимое и вносить в копии необходимые корректировки).

Рис. 39. Макет окна приложения TEXTEDIT версии 4 (панель инструментов)

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

```
  <DockPanel >
    <Menu DockPanel.Dock="Top" >
      …
    </Menu>
    <ToolBar x:Name="toolBar1" DockPanel.Dock="Top" >
      <Button x:Name="new2" Command="ApplicationCommands.New"
          ToolTip="New" >
        <Image Source="New.bmp" Stretch="None" />
      </Button>
      <Button x:Name="open2" Command="ApplicationCommands.Open"
          ToolTip="Open" >
        <Image Source="Open.bmp" Stretch="None" />
      </Button>
      <Button x:Name="save2" Command="ApplicationCommands.Save"
          ToolTip="Save" >
        <Image Source="Save.bmp" Stretch="None" />
      </Button>
      <Separator/>
      <Button x:Name="cut2" Command="ApplicationCommands.Cut"
          ToolTip="Cut" >
        <Image Source="Cut.bmp" Stretch="None" />
      </Button>
      <Button x:Name="copy2" Command="ApplicationCommands.Copy"
          ToolTip="Copy" >
        <Image Source="Copy.bmp" Stretch="None" />
      </Button>
      <Button x:Name="paste2" Command="ApplicationCommands.Paste"
          ToolTip="Paste" >
        <Image Source="Paste.bmp" Stretch="None" />
      </Button>
    </ToolBar>
    <TextBox x:Name="textBox1" >
      …
    </TextBox>
  </DockPanel>
</Window>
```

Комментарий

Описание панели инструментов ToolBar следует размещать в xaml-файле сразу после описания панели меню и снабжать таким же атрибутом DockPanel.Dock="Top", что обеспечит стыковку панели инструментов к панели меню в верхней части окна.

Содержимым каждой кнопки является компонент Image, источником для которого является один из рисунков, добавленных ранее в проект. Свойство Stretch необходимо обязательно указать, так как по умолчанию его значение равно Uniform, что приведет к пропорциональному масштабированию изображения до максимальных размеров, доступных в пределах окна (более подробно свойство Stretch обсуждается в проекте IMGVIEW, п. 17.4). Для всех кнопок мы указали связанную с ними команду WPF, а также свойство ToolTip, определяющее текст *всплывающей подсказки*.

Изображения можно связать не только с кнопками, но и с пунктами меню. Для хранения изображения в компоненте MenuItem предназначено свойство Icon. Приведем соответствующие добавления для первых трех пунктов меню (рис. 40).

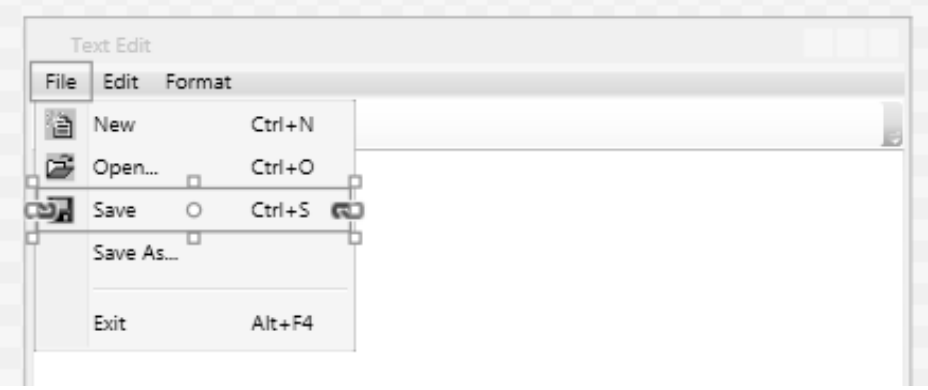


Рис. 40. Макет окна приложения TEXTEDIT версии 4 (меню с изображениями)

```
<MenuItem x:Name="new1" Header="_New"
    Command="ApplicationCommands.New" >
  <MenuItem.Icon>
    <Image Source="new.bmp" />
  </MenuItem.Icon>
</MenuItem>
<MenuItem x:Name="open1" Header="_Open..."
    Command="ApplicationCommands.Open" >
  <MenuItem.Icon>
    <Image Source="open.bmp" />
  </MenuItem.Icon>
</MenuItem>
<MenuItem x:Name="save1" Header="_Save"
    Command="ApplicationCommands.Save" >
  <MenuItem.Icon>
    <Image Source="save.bmp" />
  </MenuItem.Icon>
</MenuItem>
```

Ошибка. При попытке запуска полученного варианта программы возбуждается исключение NullReferenceException, причем в качестве фрагмента, вызвавшего исключение, указывается обработчик cut0_CanExecute.

Здесь мы сталкиваемся с проблемой, достаточно часто возникающей в приложениях WPF (ранее мы обсуждали ее в п. 3.5 при разработке проекта CALC). Дело в том, что некоторые события возникают непосредственно после создания компонента, а значит, связанные с этими событиями обработчики запускаются первый раз еще до того момента, как будут созданы все остальные компоненты приложения. В данном случае сразу после создания кнопки cut2 возникает событие CanExecute (чтобы определить, в каком состоянии следует отобразить кнопку). В момент создания кнопки компонент TextBox еще не создан (компоненты создаются в том порядке, в котором они описаны в xaml-файле), но обработчик cut0_CanExecute пытается обратиться к свойству SelectedText этого компонента, что и приводит к возбуждению исключения. Заметим, что в предыдущих версиях программы TEXTEDIT подобная проблема не возникала, так как связанные с редактированием пункты меню находятся в меню второго уровня, и до его активации доступность соответствующих команд WPF не проверяется.

Исправление. Есть два способа исправления подобных ошибок. Во-первых, можно *удалить* атрибуты Command="ApplicationCommands.Cut" и Command="ApplicationCommands.Copy" из определений кнопок cut2 и copy2 в xaml-файле и связать их с соответствующими командами WPF только в конструкторе класса MainWindow:

```
cut2.Command = ApplicationCommands.Cut;
copy2.Command = ApplicationCommands.Copy;
```

Второй способ исправления – добавить дополнительную проверку в обработчик cut0_CanExecute:

```
private void cut0_CanExecute(object sender,
  CanExecuteRoutedEventArgs e)
{
    e.CanExecute = textBox1 != null &&
      textBox1.SelectedText.Length > 0;
}
```

После любого из описанных вариантов исправления программа будет успешно запущена.

Результат. Для выполнения часто используемых команд теперь достаточно щелкнуть мышью на соответствующей кнопке панели инструментов. Чтобы определить, с какой командой связана кнопка, надо переместить на нее курсор: через 1–2 секунды около кнопки появится всплывающая подсказка с названием соответствующей команды. Около команд меню, связанных с кнопками, изображаются те же картинки, что и на кноп-

ках. Если какая-либо команда меню неактивна, то связанная с ней кнопка недоступна (в частности ее нельзя нажать, а при наведении на нее курсора мыши она не заключается в рамку и около нее не появляется всплывающая подсказка).

Комментарий

С помощью комбинации Ctrl+Tab можно переместить фокус на кнопки панели инструментов, что не характерно для такой категории интерфейсных элементов. Если такое поведение нежелательно, то достаточно для всех кнопок положить свойство Focusable равным false.

Недочет. Невозможно отличить по внешнему виду недоступную кнопку от доступной.

Исправление. Определите для каждой из четырех кнопок, которые могут быть недоступными (save2, cut2, copy2, paste2), один общий обработчик события IsEnableChanged (после задания этого обработчика для кнопки save2 следует выбрать *этот же* обработчик для трех остальных кнопок):

```
<Button x:Name="save2" Command="ApplicationCommands.Save"
    ToolTip="Save" IsEnabledChanged="save2_IsEnabledChanged" >
  <Image Source="Save.bmp" Stretch="None" />
</Button>
<Separator/>
<Button x:Name="cut2" Command="ApplicationCommands.Cut"
    ToolTip="Cut" IsEnabledChanged="save2_IsEnabledChanged" >
  <Image Source="Cut.bmp" Stretch="None" />
</Button>
<Button x:Name="copy2" Command="ApplicationCommands.Copy"
    ToolTip="Copy" IsEnabledChanged="save2_IsEnabledChanged" >
  <Image Source="Copy.bmp" Stretch="None" />
</Button>
<Button x:Name="paste2" Command="ApplicationCommands.Paste"
    ToolTip="Paste" IsEnabledChanged="save2_IsEnabledChanged" >
  <Image Source="Paste.bmp" Stretch="None" />
</Button>
```

Реализация обработчика должна быть следующей:

```
private void save2_IsEnabledChanged(object sender,
  DependencyPropertyChangedEventArgs e)
{
    var b = sender as Button;
    b.Opacity = b.IsEnabled ? 1 : 0.5;
}
```

Результат. Теперь неактивные кнопки выглядят «блеклыми» (рис. 41).

Например, при запуске программы подобным образом будут выглядеть кнопки сохранения, вырезания и копирования (и, возможно, вставки, если буфер обмена в данный момент не содержит текстовых данных).

Рис. 41. Окно приложения TEXTEDIT версии 4 с неактивными кнопками

Комментарий

Чтобы исправить недочет, мы установили для неактивных кнопок режим полупрозрачности.

12.2. Использование независимых кнопок-переключателей

Оставшиеся кнопки, которые мы планируем разместить на панели инструментов, должны принимать два состояния: «нажатое» и «отпущенное», т. е. работать как переключатели. В библиотеке WPF имеется соответствующий класс кнопок: ToggleButton (именно от этого класса порождаются компоненты для флажков CheckBox и радиокнопок RadioButton). Однако компонент ToggleButton отсутствует в палитре компонентов, поэтому разместить его в окне можно только с помощью непосредственного редактирования xaml-файла. В данном пункте мы добавим на панель те кнопки, которые изменяют свое состояние независимо от других (рис. 42).

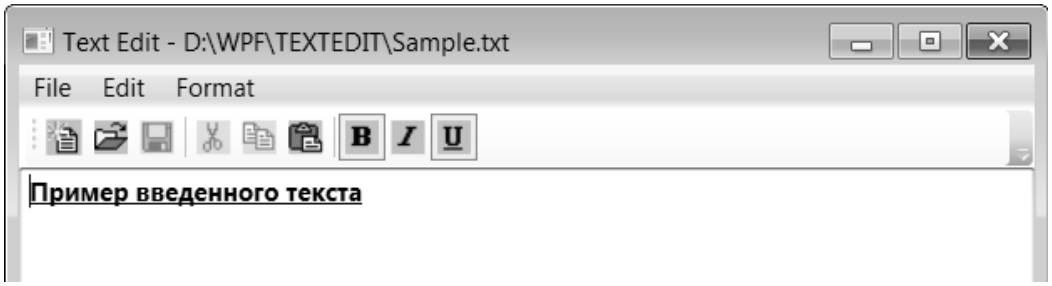


Рис. 42. Окно приложения TEXTEDIT версии 4 с кнопками-переключателями

```
<ToolBar x:Name="toolBar1" DockPanel.Dock="Top" >
  …
  <Separator/>
  <ToggleButton x:Name="bold2" ToolTip="Bold"
      Click="bold1_Click" >
    <Image Source="Bold.bmp" Stretch="None" />
  </ToggleButton>
  <ToggleButton x:Name="italic2" ToolTip="Italic"
```

```
      Click="italic1_Click" >
    <Image Source="Italic.bmp" Stretch="None" />
  </ToggleButton>
  <ToggleButton x:Name="underline2" ToolTip="Underline"
      Click="underline1_Click" >
    <Image Source="Underline.bmp" Stretch="None" />
  </ToggleButton>
</ToolBar >
```

Мы связали с кнопками-переключателями те обработчики, которые ранее были определены для соответствующих пунктов меню (напомним, что пункты меню, отвечающие за форматирование, мы не связывали с командами WPF). Текст этих обработчиков необходимо откорректировать:

```
private void bold1_Click(object sender, RoutedEventArgs e)
{
    bold2.IsChecked = bold1.IsChecked = !bold1.IsChecked;
    textBox1.FontWeight = bold1.IsChecked ? FontWeights.Bold :
      FontWeights.Normal;
}
private void italic1_Click(object sender, RoutedEventArgs e)
{
    italic2.IsChecked = italic1.IsChecked = !italic1.IsChecked;
    textBox1.FontStyle = italic1.IsChecked ? FontStyles.Italic :
      FontStyles.Normal;
}
private void underline1_Click(object sender, RoutedEventArgs e)
{
    underline2.IsChecked = underline1.IsChecked =
      !underline1.IsChecked;
    textBox1.TextDecorations = underline1.IsChecked ?
      TextDecorations.Underline : null;
}
```

Результат. Для настройки шрифта достаточно нажать на соответствующую кнопку панели инструментов, переведя ее тем самым в «нажатое состояние». Далее мы будем говорить о «нажатом» и «отжатом» состоянии кнопки, хотя при использовании стандартного стиля изображения кнопок в Windows «нажатая» кнопка просто обводится рамкой.

Нажатое состояние кнопки означает, что указанный режим включен. Кнопки настройки шрифта действуют как *флажки*: каждую кнопку можно перевести в нажатое или отжатое состояние независимо от остальных.

12.3. Использование зависимых кнопок-переключателей. Привязка свойств

Добавьте в xaml-файл кнопки, отвечающие за режим выравнивания:

```
<ToolBar x:Name="toolBar1" DockPanel.Dock="Top" >
  …
  <Separator/>
  <ToggleButton x:Name="leftJustify2" ToolTip="Left Justify"
      Click="leftJustify1_Click" >
    <Image Source="Left.bmp" Stretch="None" />
  </ToggleButton>
  <ToggleButton x:Name="center2" ToolTip="Center"
      Click="leftJustify1_Click" >
    <Image Source="Center.bmp" Stretch="None" />
  </ToggleButton>
  <ToggleButton x:Name="rightJustify2" ToolTip="Right Justify"
      Click="leftJustify1_Click" >
    <Image Source="Right.bmp" Stretch="None" />
  </ToggleButton>
</ToolBar >
```

Реализовать корректное поведение для кнопок, связанных с командами выравнивания, сложнее, чем для кнопок настройки шрифта. Это обусловлено тем, что для команд выравнивания мы определили один общий обработчик – leftJustify1_Click, в котором производится обращение к параметру sender, причем предполагается, что этот параметр является объектом типа MenuItem. После связывания обработчика leftJustify1_Click с кнопками типа ToggleButton именно они будут выступать в роли объекта sender при их нажатии, и попытка преобразования этого объекта к типу MenuItem приведет в итоге к возбуждению исключения NullReferenceException. В этом можно убедиться, запустив полученную программу и нажав на одну из добавленных кнопок.

Для исправления выявленной ошибки сохраним в кнопках выравнивания информацию о связанных с ними командами меню, записав ссылки на команды меню в свойство Tag. Соответствующие операторы надо добавить в конструктор MainWindow:

```
leftJustify2.Tag = leftJustify1;
center2.Tag = center1;
rightJustify2.Tag = rightJustify1;
```

И теперь откорректируем метод leftJustify1_Click, добавив в него дополнительную проверку параметра sender с помощью операции ??:

```
private void leftJustify1_Click(object sender, RoutedEventArgs e)
{
```

```
    MenuItem mi = sender as MenuItem ??
        (sender as ToggleButton).Tag as MenuItem;
    if (mi.IsChecked)
        return;
    alignItem.IsChecked = false;
    alignItem = mi;
    mi.IsChecked = true;
    textBox1.HorizontalContentAlignment =
         (HorizontalAlignment)mi.Tag;
}
```

Для того чтобы компилятор распознал тип ToggleButton, надо добавить в набор директив using следующую директиву:

```
using System.Windows.Controls.Primitives;
```

Теперь кнопки выравнивания позволяют выполнять требуемые действия. Однако эти кнопки не образуют группу связанных переключателей: нажатие на одну из них не освобождает остальные, более того, повторное нажатие кнопки переводит ее в отжатое состояние, хотя при этом выполняется связанная с ней команда. Таким образом, кнопки выравнивания все еще ведут себя как независимые переключатели, подобно кнопкам настройки шрифта.

Поскольку мы уже обеспечили согласованное поведение пунктов меню, относящихся к выравниванию, возникает идея *связать* состояние кнопок с состоянием соответствующих пунктов меню. Для этого в WPF имеется удобный механизм – *привязка* (binding).

Дополним описания кнопок выравнивания в xaml-файле следующим образом:

```
<ToggleButton x:Name="leftJustify2" ToolTip="Left Justify"
    Click="leftJustify1_Click" IsChecked ="{Binding
    Path=IsChecked, ElementName=leftJustify1, Mode=OneWay}" >
  <Image Source="Left.bmp" Stretch="None" />
</ToggleButton>
<ToggleButton x:Name="center2" ToolTip="Center"
    Click="leftJustify1_Click" IsChecked ="{Binding
    Path=IsChecked, ElementName=center1, Mode=OneWay}" >
  <Image Source="Center.bmp" Stretch="None" />
</ToggleButton>
<ToggleButton x:Name="rightJustify2" ToolTip="Right Justify"
    Click="leftJustify1_Click" IsChecked ="{Binding
    Path=IsChecked, ElementName=rightJustify1, Mode=OneWay}" >
  <Image Source="Right.bmp" Stretch="None" />
</ToggleButton>
```

Мы осуществили привязку свойства IsChecked кнопки к одноименному свойству соответствующей команды меню. Обратите внимание на используемый синтаксис: все настройки привязки указываются в общих фигурных скобках, причем, в отличие от «обычных» XML-атрибутов, значения настроек не заключаются в кавычки, а сами настройки разделяются запятыми.

В нашем случае кнопка является *приемником* привязки, а пункт меню – ее *источником*. Имя объекта-источника указывается в настройке ElementName, а свойство источника, к которому привязывается свойство приемника, – в настройке Path. Настройка Mode устанавливает *режим* привязки; использованное значение OneWay является наиболее распространенным, в этом случае только источник влияет на приемник (из других режимов отметим режим TwoWay, при котором воздействие распространяется «в обе стороны»).

Комментарий

В фигурные скобки при определении атрибутов-свойств в xaml-файле заключаются так называемые *расширения разметки*. Встретив подобный текст, компилятор или анализатор XAML не будет считать его обычной строкой (и, при необходимости, конвертировать в другой тип данных, с учетом типа определяемого свойства). Вместо этого он по первому идентификатору, содержащемуся в фигурных скобках, определит имя класса, связанного с данным расширением (это имя получается из имени идентификатора добавлением суффикса Extension; например, в нашем случае будет использован класс BindingExtension), а прочие настройки в фигурных скобках будут использованы как параметры для конструктора класса расширения (параметры могут быть как позиционными, так и именованными). Созданный объект класса расширения будет присвоен указанному атрибуту-свойству.

Результат. Любое изменение свойства IsChecked пункта меню немедленно приводит к изменению одноименного свойства связанной с этим пунктом кнопки. Теперь кнопки выравнивания действительно работают как набор связанных переключателей: нажатие на одну из них обеспечивает изменение свойств IsChecked пунктов меню, а они, в свою очередь, изменяют вид кнопок.

Недочет. Эта согласованная работа дает сбой в единственном случае: когда пользователь попытается *повторно нажать на уже нажатую кнопку выравнивания*. В результате установленный режим выравнивания не изменится, не изменится также и установленный пункт меню, но сама кнопка *перейдет в отжатое состояние*. Это объясняется двумя обстоятельствами. Во-первых, кнопка (в отличие от пункта меню MenuItem) не имеет настраиваемого свойства IsCheckable; она *всегда* изменяет свое состояние при нажатии на нее. Таким образом, нажатие на уже нажатую кнопку обя-

зательно приводит к ее «отжатию» (заметим, что в случае независимых кнопок-переключателей это очень удобно). Во-вторых, при повторном нажатии на *уже нажатую* кнопку выравнивания обработчик leftJustify1_Click не дорабатывает до конца: обнаружив, что соответствующий пункт меню уже установлен во включенное состояние, он просто завершает работу. Но раз состояние источника привязки не изменилось, то и обновление приемника привязки не выполняется. Таким образом, указанные два обстоятельства приводят к неправильному отображению состояния кнопки.

Исправление. Самым простым способом исправления является удаление фрагмента обработчика leftJustify1_Click, приводящего к досрочному выходу из него:

```
private void leftJustify1_Click(object sender, RoutedEventArgs e)
{
    MenuItem mi = sender as MenuItem ??
        (sender as ToggleButton).Tag as MenuItem;
    if (mi.IsChecked)
        return;
    alignItem.IsChecked = false;
    alignItem = mi;
    mi.IsChecked = true;
    textBox1.HorizontalContentAlignment =
         (HorizontalAlignment)mi.Tag;
}
```

Результат. Теперь обработчик всегда дорабатывает до конца, в процессе его работы обязательно изменяется состояние пункта меню, и благодаря этому синхронно изменяется и состояние связанной с ним кнопки. Таким образом, даже бессмысленные действия пользователя (повторное нажатие на уже нажатую кнопку выравнивания) корректно обрабатываются.

13. Статусная панель и дополнительные возможности привязки: TXTEDIT, версия 5.

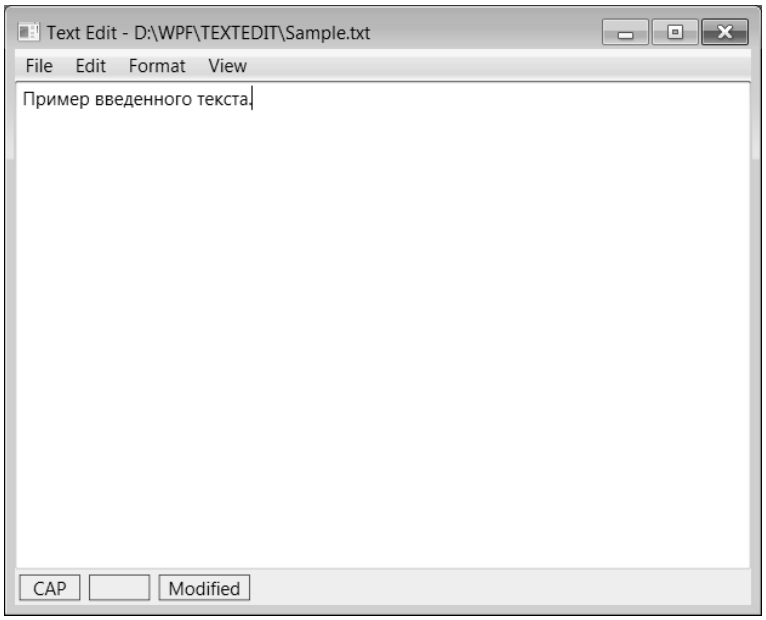


Рис. 43. Окно приложения TEXTEDIT версии 5

13.1. Использование статусной панели. Определение свойств зависимости. Привязка данных с использованием конвертеров типов

```
<Window x:Class="TEXTEDIT.MainWindow" … >
  …
  <DockPanel >
    …
    <ToolBar x:Name="toolBar1" DockPanel.Dock="Top" >
      …
    </ToolBar>
    <StatusBar x:Name="statusBar1" DockPanel.Dock="Bottom" >
      <Border BorderBrush="Black" BorderThickness="1" Width="40">
        <TextBlock x:Name="cap1" Text="CAP"
            HorizontalAlignment="Center"/>
      </Border>
      <Border BorderBrush="Black" BorderThickness="1" Width="40">
        <TextBlock x:Name="num1" Text="NUM"
            HorizontalAlignment="Center"/>
      </Border>
```

```
      <Border BorderBrush="Black" BorderThickness="1" Width="60">
        <TextBlock x:Name="modified1" Text=""
            HorizontalAlignment="Center"/>
      </Border>
    </StatusBar>
    …
  </DockPanel>
</Window>
```

В описание класса MainWindow добавьте вспомогательный метод:

```
void CheckKeyModifiers()
{
    cap1.Text = (Keyboard.GetKeyStates(Key.CapsLock) &
      KeyStates.Toggled) == KeyStates.Toggled ? "CAP" : "";
    num1.Text = (Keyboard.GetKeyStates(Key.NumLock) &
      KeyStates.Toggled) == KeyStates.Toggled ? "NUM" : "";
}
```

И добавьте вызов данного метода в конец конструктора класса MainWindow и в начало метода Window_PreviewKeyDown.

Результат. На статусной панели отображается текущее состояние клавиш CapsLock и NumLock.

В третьем элементе статусной панели должна выводиться информация о том, изменялся ли документ после его последнего сохранения.

Изменять значение третьего элемента статусной панели надо при каждом изменении значения свойства Modified. Поэтому было бы удобно, если бы само свойство могло информировать другие компоненты о своих изменениях. Подобное информирование легко реализовать, если свойство является *свойством зависимости* (см. проект EVENTS, п. 1.2).

Для того чтобы превратить свойство Modified в свойство зависимости, необходимо заменить в классе MainWindow прежнее описание свойства Modified на следующий набор членов класса:

```
bool Modified { get; set; }
static readonly DependencyProperty ModifiedProperty =
  DependencyProperty.Register("Modified", typeof(bool),
  typeof(MainWindow), new PropertyMetadata(false));
bool Modified
{
    get { return (bool)GetValue(ModifiedProperty); }
    set { SetValue(ModifiedProperty, value); }
}
```

Комментарии

1. Со свойством зависимости связывается статическое поле типа DependencyProperty. Кроме того, свойство зависимости должно *регистрироваться* в содержащем его классе с помощью метода Register; для этого можно использовать либо статический конструктор, либо (как в нашем случае) *инициализирующее выражение* для статического поля, поскольку подобное действие по инициализации статического поля автоматически связывается со статическим конструктором. Дополнительные настройки для регистрируемого свойства указываются в последнем параметре метода Register; в частности, в нем можно указать значение свойства по умолчанию (в нашем случае false).

Свойства зависимости удобно использовать в качестве источников привязки, так как их изменение отслеживается системой автоматически (заметим, что в качестве источника привязки можно использовать и обычное свойство, но для этого необходимо, чтобы в класс с этим свойством было добавлено *дополнительное событие*, позволяющее отслеживать изменение свойства – см., например, [9, гл. 13]). В качестве приемников привязки можно применять *только* свойства зависимости. Кроме того, как уже отмечалось ранее, свойства зависимости используются во многих других механизмах WPF, в частности, при назначении стилей, при передаче значений свойств от родителей к дочерним компонентам, при реализации присоединенных свойств и т. д.

2. Быстро создать заготовку для определения свойства зависимости можно с помощью специального *шаблона для автогенерации кода* (code snippet), набрав в редакторе текст этого шаблона *propdp* и дважды нажав клавишу Tab. В результате будет сгенерирован следующий фрагмент:

```
public int MyProperty
{
    get { return (int)GetValue(MyPropertyProperty); }
    set { SetValue(MyPropertyProperty, value); }
}
public static readonly DependencyProperty MyPropertyProperty =
  DependencyProperty.Register("MyProperty", typeof(int),
  typeof(ownerclass), new PropertyMetadata(0));
```

Сразу после создания свойства можно легко изменить его настройки: тип (значение по умолчанию int), название (MyProperty), имя класса-владельца (ownerclass) и значение по умолчанию (0). Для перемещения по указанным настройкам достаточно нажимать клавишу Tab; изменение любой настройки приводит к соответствующим корректировкам во всем определении свойства.

В редакторе кода Visual Studio предусмотрено много различных шаблонов автогенерации. Для их просмотра достаточно вызвать окно менед-

жера шаблонов (Code Snippets Manager), нажав комбинацию (Ctrl+K, B) и выбрав требуемый язык программирования в списке Language. Среди наиболее полезных можно отметить шаблоны *for* и *forr* для заголовков цикла (второй шаблон генерирует заголовок для цикла с убывающим параметром), а также шаблон *cw* для вывода на консоль Console.WriteLine().

Мы хотим использовать свойство Modified как *источник* для свойства Text элемента modified1 статусной строки. Но типы свойства-источника и свойства-приемника не совпадают. В этом случае надо использовать *конвертеры типов*. Для каждого конвертера надо создавать новый класс. Опишем этот класс в файле MainWindow.xaml.cs, добавив его перед описанием класса MainWindow:

```
public class ModifiedToStringConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
      object parameter, System.Globalization.CultureInfo culture)
    {
        return (bool)value ? "Modified" : "";
    }
    public object ConvertBack(object value, Type targetType,
      object parameter, System.Globalization.CultureInfo culture)
    {
        return (string)value != "";
    }
}
```

Обратное преобразование нам не требуется, но его все равно надо определить.

Еще одна проблема, связана с тем, что для указания конвертера типов в xaml-файле надо иметь *экземпляр* этого класса. В xaml-файле для создания объектов используются *ресурсы XAML* (ресурсы XAML не следует путать с *ресурсами приложения*, примером которых являются графические файлы, включенные в наш проект). Ресурсы в xaml-файле играют роль «переменных», поскольку имеют имена (*ключи*) и значения. Ресурс можно использовать во всех дочерних компонентах того компонента, в котором ресурс определен (в частности, чтобы сделать ресурс доступным для всех компонентов окна, его надо определить в самом окне, т. е. в XML-элементе Window).

Добавьте в xaml-файл определение ресурса (а также добавьте в элемент Window атрибут x:Name, поскольку нам потребуется обращаться к окну по имени):

```
<Window x:Class="TEXTEDIT.MainWindow"
    …
```

```
    xmlns:local="clr-namespace:TEXTEDIT"
    … x:Name="mainWindow" >
    <Window.Resources>
        <local:ModifiedToStringConverter x:Key="ModifConv" />
    </Window.Resources>
  …
</Window>
```

Префикс local означает, что класс ModifiedToStringConverter находится в пространстве имен приложения TEXTEDIT, так как в начале xaml-файла содержится соответствующее определение данного префикса (см. приведенный выше фрагмент xaml-файла). Префиксы, связанные с пространствами имен, ранее уже обсуждались в версии 2 проекта TEXTEDIT, п. 10.3.

Заметим, что после добавления в xaml-файла указанного определения ресурса оно может быть помечено как ошибочное, однако после перекомпиляции проекта пометка ошибки будет удалена.

Теперь у нас есть все необходимое для создания привязки свойства Text компонента TextBlock из статусной панели к свойству Modified окна:

```
<StatusBar x:Name="statusBar1" DockPanel.Dock="Bottom" >
  …
  <Border BorderBrush="Black" BorderThickness="1"  Width="60">
    <TextBlock x:Name="modified1" Text=""
        HorizontalAlignment="Center"
        Text="{Binding ElementName=mainWindow, Path=Modified,
        Converter={StaticResource ModifConv}}" />
  </Border>
</StatusBar>
```

Для обращения к статическому ресурсу используется специальный синтаксис: значение атрибута обрамляется фигурными скобками и включает текст StaticResource и ключ ресурса. Таким образом, здесь используется *расширение разметки* (см. комментарий в п. 12.3) с позиционным параметром-ключом ModifConv.

Результат. Если редактируемый документ изменялся после последнего сохранения, то в третьем элементе статусной панели выводится текст «Modified».

13.2. Скрытие панелей: два варианта реализации

Реализуем возможность скрытия панели инструментов и статусной панели, причем используем для этого два разных подхода: с применением обработчика события и с применением привязки к свойству.

Вначале добавим к меню новую группу команд:

```
<MenuItem x:Name="view1" Header="_View" >
```

```
  <MenuItem x:Name="viewToolBar1" Header="_Tool bar"
      IsCheckable="True" IsChecked="True" />
  <MenuItem x:Name="viewStatusBar1" Header="_Status bar"
      IsCheckable="True" IsChecked="True" />
</MenuItem>
```

Напомним, что значение true свойства IsCheckable обеспечивает автоматическую установку или снятие флажка около пункта меню при выборе этого пункта пользователем.

Выполнение команды viewToolBar1 реализуем с помощью обработчика события, указав его в xaml-файле и определив его действие в cs-файле:

```
<MenuItem x:Name="viewToolBar1" Header="_Tool bar"
    IsCheckable="True" IsChecked="True"
    Click="viewToolBar1_Click" />
```

```
private void viewToolBar1_Click(object sender, RoutedEventArgs e)
{
    toolBar1.Visibility = viewToolBar1.IsChecked ?
      Visibility.Visible : Visibility.Collapsed;
}
```

Для команды viewStatusBar1 вместо определения обработчика установим привязку свойства IsVisible статусной панели к свойству IsChecked данной команды. При этом нас ожидает приятный сюрприз: в WPF предусмотрен стандартный конвертер BooleanToVisibilityConverter, позволяющий преобразовывать тип bool в тип Visibility. Таким образом, определять еще один класс конвертера в нашем проекте не придется. Надо лишь создать в xaml-файле экземпляр этого конвертера, который затем использовать при установке привязки (привязка устанавливается для свойства Visibility компонента statusBar1):

```
<Window.Resources>
  <local:ModifiedToStringConverter x:Key="ModifConv" />
  <BooleanToVisibilityConverter x:Key="BoolConv" />
</Window.Resources>
…
<StatusBar x:Name="statusBar1" VerticalAlignment="Top"
    DockPanel.Dock="Bottom"
    Visibility="{Binding ElementName=viewStatusBar1,
    Path=IsChecked, Converter={StaticResource BoolConv}}" >
```

Результат. С помощью команд-переключателей меню «View» можно скрывать и восстанавливать статусную панель и панель инструментов. Напомним, что вариант скрытия Visibility.Collapsed освобождает область, ранее занятую скрытым компонентом, и поэтому она может быть занята

другим видимым компонентом (в то время как вариант Visibility.Hidden сохраняет за скрытым компонентом занимаемую им область).

13.3. Дополнение. Реализация команд-переключателей без использования обработчиков событий

В данном, завершающем пункте, связанном с разработкой проекта TEXTEDIT, мы не будем добавлять к проекту новые возможности. Вместо этого рассмотрим, каким образом можно реализовать имеющиеся команды-переключатели *без обработчиков событий*. Ранее уже отмечалось, что альтернативой обработчикам событий в библиотеке WPF является применение команд WPF и механизма привязки. Обе эти возможности мы уже использовали в нашем проекте. Попытаемся применить их и в данном случае.

Для краткости изложения ограничимся командой установки полужирного шрифта; прочие команды-переключатели можно откорректировать аналогичными действиями.

Прежде всего, определим новую команду WPF SetBold, добавив ее в имеющийся класс FormatCommands:

```
class FormatCommands
{
    …
    private static RoutedUICommand bold;
    static FormatCommands()
    {
        InputGestureCollection inputs = new
          InputGestureCollection();
        …
        inputs = new InputGestureCollection();
        inputs.Add(new KeyGesture(Key.B, ModifierKeys.Control,
         "Ctrl+B"));
        bold = new RoutedUICommand("_Bold", "SetBold",
          typeof(FormatCommands), inputs);
    }
    …
    public static RoutedUICommand SetBold
    {
        get { return bold; }
    }
}
```

После этого зарегистрируем новую команду в xaml-файле:

```
<Window.CommandBindings>
```

```
  …
  <CommandBinding x:Name="bold0"
      Command="local:FormatCommands.SetBold"
      Executed="bold0_Executed" />
</Window.CommandBindings>
```

Созданный обработчик bold0_Executed будет просто изменять свойство FontWeight компонента textBox1:

```
private void bold0_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    textBox1.FontWeight = textBox1.FontWeight ==
      FontWeights.Normal ? FontWeights.Bold : FontWeights.Normal;
}
```

Обратите внимание на то, что в нем нет никаких отсылок на те интерфейсные элементы нашего приложения, которые будут связаны с данной командой. Теперь надо позаботиться о том, чтобы состояние этих интерфейсных элементов соответствовало текущему режиму настройки полужирного шрифта. Однако данная настройка определяется свойством типа FontWeight, а состояние как пункта меню, так и кнопки задается свойством типа bool. Поэтому нам потребуется определить еще один конвертер типа, добавив его описание, как и описание предыдущего конвертера, перед описанием класса MainWindow):

```
public class FontWeightToBoolConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object
      parameter, System.Globalization.CultureInfo culture)
    {
        return (FontWeight)value == FontWeights.Bold;
    }
    public object ConvertBack(object value, Type targetType,
      object parameter, System.Globalization.CultureInfo culture)
    {
        return (bool)value ? FontWeights.Bold :
          FontWeights.Normal;
    }
}
```

Экземпляр этого конвертера надо создать в xaml-файле, добавив его в статические ресурсы:

```
<Window.Resources>
  …
  <local:FontWeightToBoolConverter x:Key="BoldConv" />
```

```
</Window.Resources>
```

Напомним, что если элементы xaml-файла, связанные с только что определенными классами, выделяются как ошибочные, то достаточно выполнить перекомпиляцию проекта, чтобы пометки ошибок исчезли.

Теперь все готово для подключения команды WPF к интерфейсным компонентам и привязки этих компонентов к настраиваемому свойству:

```
<MenuItem x:Name="bold1" Header="_Bold" InputGestureText="Ctrl+B"
    Click="bold1_Click" Command="local:FormatCommands.SetBold"
    IsChecked="{Binding ElementName=textBox1, Path=FontWeight,
    Converter={StaticResource ResourceKey=BoldConv}}" />
…
<ToggleButton x:Name="bold2" ToolTip="Bold" Click="bold1_Click"
    Command="local:FormatCommands.SetBold"
    IsChecked="{Binding ElementName=textBox1, Path=FontWeight,
    Converter={StaticResource ResourceKey=BoldConv}}" >
  <Image Source="Bold.bmp" Stretch="None" />
</ToggleButton>
```

После внесения этих изменений в xaml-файл можно удалить из описания класса MainWindow метод bold1_Click и фрагмент обработчика Window_PreviewKeyDown, связанный с клавиатурной комбинацией Ctrl+B:

```
case Key.B:
    bold1_Click(null, null);
    e.Handled = true;
    break;
```

Ошибка. Стиль шрифта будет правильно изменяться при использовании команды меню или клавиши быстрого доступа. Но щелчок на кнопке не будет приводить ни к какому результату. Если выполнить трассировку метода bold0_Executed, связанного с командой bold0, то можно обнаружить, что при нажатии на кнопку он выполняется *один раз*; таким образом, стиль шрифта должен измениться. Но после этого состояние шрифта меняется еще раз и кнопка переходит в «отжатый» режим.

Причина ошибки состоит в том, что установленная нами для кнопки привязка по умолчанию *действует в обе стороны*: не только кнопка переходит в «нажатый» режим при изменении шрифта на полужирный, но и шрифт становится полужирным при нажатии кнопки. Получается следующая картина: при щелчке на кнопке она переходит в «нажатый» режим, что автоматически приводит к изменению шрифта на полужирный. Затем запускается команда bold0, которая переводит шрифт обратно в обычный, а привязка при этом переводит кнопку в отжатое состояние. В итоге ничего не происходит.

Возникает вопрос: почему все правильно работает для пункта меню? Это связано с тем, что для него мы не устанавливали свойство IsCheckable, и поэтому щелчок на пункте меню *не приводит к его выделению* и тем самым к корректировке шрифта. Если положить IsCheckable равным true, то пункт меню будет вести себя так же неправильно, как и кнопка.

Исправление. После того как причина ошибки выявлена, нетрудно реализовать два варианта ее исправления.

1. Можно добавить в настройки привязки для кнопки параметр Mode=OneWay:

```
<ToggleButton x:Name="bold2" ToolTip="Bold"
    Command="local:FormatCommands.SetBold"
    IsChecked="{Binding ElementName=textBox1, Path=FontWeight,
    Converter={StaticResource ResourceKey=BoldConv},
    Mode=OneWay}" >
```

В этом случае изменение шрифта поля ввода будет влиять на состояние кнопки, а изменение состояния кнопки не будет влиять на шрифт.

2. Еще проще вообще убрать атрибут Command из описания кнопки bold2 (и, разумеется, не указывать настройку Mode=OneWay, оставив для параметра Mode значение по умолчанию TwoWay):

```
<ToggleButton x:Name="bold2" ToolTip="Bold"
    Command="local:FormatCommands.SetBold"
    IsChecked="{Binding ElementName=textBox1, Path=FontWeight,
    Converter={StaticResource ResourceKey=BoldConv}}" >
```

Между прочим, удалить атрибут Command можно и для пункта меню bold1. Однако при этом потребуется добавить три атрибута (два из которых мы ранее удалили вместе с обработчиком события):

```
<MenuItem x:Name="bold1" Header="_Bold" InputGestureText="Ctrl+B"
    IsCheckable="True" Command="local:FormatCommands.SetBold"
    IsChecked="{Binding ElementName=textBox1, Path=FontWeight,
    Converter={StaticResource ResourceKey=BoldConv}}" />
```

Если внести это исправление в проект, то команда bold0 и ее обработчик будут использоваться только при реакции на нажатие клавиши быстрого доступа Ctrl+B.

Комментарий

Для команд выравнивания текста тоже можно отказаться от обработчиков событий, заменив их командами WPF и привязкой компонентов. При этом придется определить *три* команды WPF, поскольку с ними надо связать разные клавиши быстрого доступа. Казалось бы, потребуется также определить *три* новых конвертера значений, поскольку для настройки состояния разных интерфейсных компонентов значение выравнивания требуется интерпретировать по-разному. Однако здесь удастся ограничиться одним «универсальным» конвертером, если воспользовать-

ся возможностью передачи в конвертер *дополнительного параметра* parameter (указывая в нем, например, строки Left, Center или Right). При определении привязки в xaml-файле дополнительный параметр конвертера можно передавать с помощью именованного параметра Converter-Parameter расширения разметки Binding.

14. Цвета: COLORS

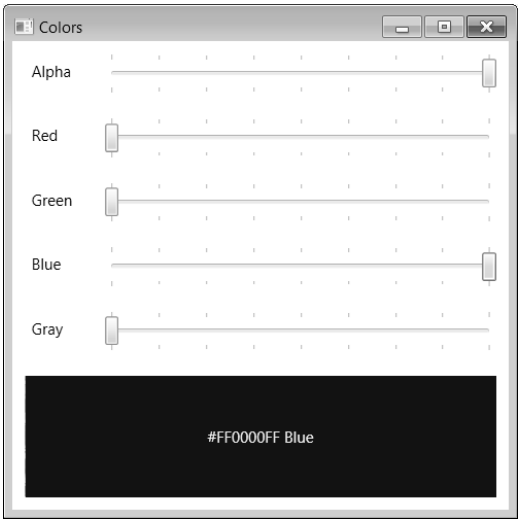


Рис. 44. Окно приложения COLORS

14.1. Начальная настройка макета окна

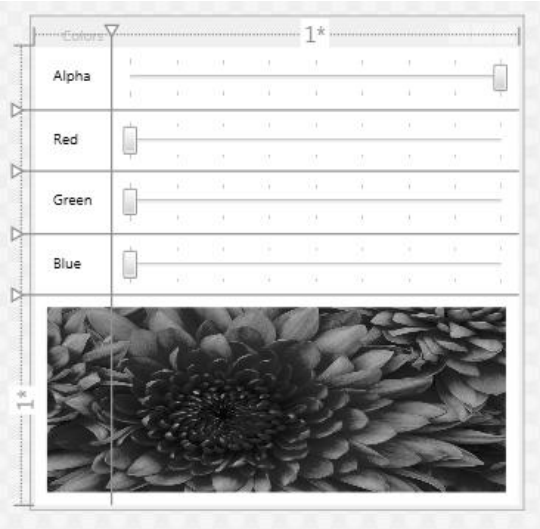


Рис. 45. Макет окна приложения COLORS

```
<Window x:Class="COLORS.MainWindow"
    …
    Title="Colors" Height="400" Width="400"
    MinHeight="400" MinWidth="400"
    WindowStartupLocation="CenterScreen" >
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
```

```
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Label x:Name="label1" Margin="10" Content="Alpha" />
    <Label x:Name="label2" Grid.Row="1" Margin="10"
        Content="Red" />
    <Label x:Name="label3" Grid.Row="2" Margin="10"
        Content="Green" />
    <Label x:Name="label4" Grid.Row="3" Margin="10"
        Content="Blue" />
    <Slider x:Name="slider1" Grid.Column="1" Margin="10"
        SmallChange="1" LargeChange="32" Maximum="255"
        TickFrequency="32" TickPlacement="Both" Value="255" />
    <Slider x:Name="slider2" Grid.Column="1" Grid.Row="1"
        Margin="10" SmallChange="1" LargeChange="32"
        Maximum="255" TickFrequency="32" TickPlacement="Both" />
    <Slider x:Name="slider3" Grid.Column="1" Grid.Row="2"
        Margin="10" SmallChange="1" LargeChange="32"
        Maximum="255" TickFrequency="32" TickPlacement="Both" />
    <Slider x:Name="slider4" Grid.Column="1" Grid.Row="3"
        Margin="10" SmallChange="1" LargeChange="32"
        Maximum="255" TickFrequency="32" TickPlacement="Both" />
    <DockPanel Grid.ColumnSpan="2" Grid.Row="5" Margin="10" >
      <DockPanel.Background>
        <ImageBrush ImageSource="Chrysanthemum.jpg"/>
      </DockPanel.Background>
    </DockPanel>
  </Grid>
</Window>
```

Кроме определения xaml-файла, необходимо добавить к проекту графический файл с достаточно рельефным изображением, сохранив его как *встроенный ресурс приложения* (действия по добавлению файла в проект в качестве ресурса приложения были описаны в п. 8.3 проекта CURSORS, а также в п. 12.1 проекта TEXTEDIT версии 4). Мы выбрали рисунок

Chrysanthemum.jpg, расположенный в системном каталоге изображений Windows (Users\Public\Pictures\Sample Pictures).

Результат. При запуске программы в окне отображаются четыре вертикально расположенных *ползунка* (компонента Slider), снабженных метками, а также прямоугольная область (компонент DockPanel), заполненная изображением (изображение масштабируется по размерам компонента без сохранения пропорций). При изменении размеров окна происходит изменение ширины ползунков и рисунка, а также изменение высоты рисунка. Окно не может быть сделано меньше его начального размера; таким образом, при любом изменении размеров окна на нем отображаются все компоненты.

Ползунки имеют одинаковый диапазон значений: от 0 до 255, причем в начале работы программы первый ползунок имеет значение 255, а остальные – значение 0. Чтобы изменить значения ползунков, можно использовать как мышь, так и клавиатуру (для возможности использования клавиатуры надо предварительно установить фокус на требуемом ползунке). Для управления ползунками с помощью клавиатуры предназначены клавиши со стрелками (изменяющие положение ползунка на 1), клавиши PgUp и PgDn (изменяющие положение на 32) и клавиши Home и End, устанавливающие ползунок в начало или конец диапазона значений.

Пока изменение ползунков не приводит ни к какому результату; это будет исправлено в следующем пункте.

Комментарии

1. Атрибуты Height="Auto", указанные для всех строк компонента Grid, кроме последней, обеспечивают подбор высоты этих строк по размерам содержащихся в них компонентов. Если бы эти атрибуты отсутствовали, то по умолчанию для всех строк была бы установлена одинаковая высота. По аналогичной причине для первого столбца был установлен атрибут Width="Auto".

2. При задании рисунка в качестве фона используется объект типа ImageBrush, свойства которого можно просмотреть и настроить с помощью окна Properties. Одним из важных свойств является свойство Stretch, определяющее способ вписывания фонового изображения в содержащий его компонент. По умолчанию для типа ImageBrush это свойство имеет значение Fill, при котором изображение масштабируется по размерам компонента без сохранения пропорций. Этот вариант наилучшим образом подходит для наших целей. Еще одним подходящим вариантом является UniformToFill, который также масштабирует изображение по размерам компонента, но при этом сохраняет пропорции и, кроме того, обеспечивает заполнение изображением всей области компонента (при этом часть изображения отсекается). Вариант Uniform нас не устраивает, так как он сохраняет пропорции и при этом целиком выводит изображение в компо-

ненте, в результате чего слева и справа (или сверху и снизу) от изображения остаются незаполненные промежутки.

3. Использованный в макете группирующий компонент DockPanel позволяет пристыковывать свои дочерние компоненты к различным границам, причем по умолчанию последний дочерний компонент будет занимать все оставшееся незаполненным пространство своего родителя (ранее мы использовали этот компонент в проекте TEXTEDIT). Мы воспользуемся этой особенностью компонента DockPanel в следующем пункте. Обратите внимание на свойство Grid.ColumnSpan="2" для DockPanel.

4. Поведение и внешний вид ползунков настраиваются с помощью соответствующих свойств, причем следует иметь в виду, что все числовые свойства имеют тип double.

5. Для фиксации минимальных размеров любого компонента (в том числе окна) достаточно определить требуемым образом свойства MinHeight и MinWidth.

14.2. Определение цвета с использованием ползунков как комбинации трех основных цветов и альфа-составляющей

```
<DockPanel Grid.ColumnSpan="2" Grid.Row="5" Margin="10" >
  …
  <Label x:Name="caption1" Background ="Black" />
</DockPanel>
```

Для компонента slider1 определите обработчик события ValueChanged:

```
<Slider x:Name="slider1" … ValueChanged="slider1 ValueChanged" />
```

```
private void slider1_ValueChanged(object sender,
  RoutedPropertyChangedEventArgs<double> e)
{
    caption1.Background =
      new SolidColorBrush(Color.FromArgb((byte)slider1.Value,
      (byte)slider2.Value, (byte)slider3.Value,
      (byte)slider4.Value));
}
```

После создания обработчика *удалите* связанный с ним атрибут ValueChanged="slider1_ValueChanged" в xaml-файле:

```
<Slider x:Name="slider1" … ValueChanged="slider1_ValueChanged" />
```

В конструктор класса MainWindow добавьте вызов метода AddHandler:

```
AddHandler(Slider.ValueChangedEvent,
  new RoutedPropertyChangedEventHandler<double>
  (slider1_ValueChanged));
```

Результат. Цвет фона метки caption1 определяется как комбинация четырех цветовых составляющих: *прозрачности* (Alpha) и интенсивности трех *базовых цветов*: красного (Red), зеленого (Green) и синего (Blue). Каждая цветовая составляющая может меняться в пределах от 0 до 255; значение 255 для составляющей Alpha соответствует *полной непрозрачности*. В нашей программе значения цветовых составляющих задаются положением соответствующего компонента Slider. Метка caption1 расположена на компоненте DockPanel, имеющем фоновый рисунок, этот рисунок «просвечивает» сквозь фон метки при уровне прозрачности, отличном от 255.

Комментарии

1. Поскольку метка caption1 является единственным дочерним компонентом группирующего компонента DockPanel, она по умолчанию захватывает всю его клиентскую область.

2. Для определения нового фонового цвета метки caption1 необходимо создать сплошную кисть типа SolidColorBrush, передав в ее конструктор требуемый цвет, который, в свою очередь, надо сформировать на основе четырех базовых составляющих с помощью статического метода FromArgb класса Color. Обратите внимание на необходимость приведения значений свойств Value к типу byte.

3. В данном проекте мы снова встречаемся с ситуацией, когда обработчик события указывается для группирующего компонента (в данном случае окна), у которого *отсутствует* соответствующее событие. Тем не менее, благодаря механизму маршрутизируемых событий (см. проект CALC, п. 3.1), это обеспечивает вызов данного обработчика для всех его дочерних компонентов, для которых предусмотрено это событие.

Мы связали обработчик slider1_ValueChanged с компонентом Window с помощью метода AddHandler (этот метод ранее упоминался в проекте CALC, п. 3.5, комментарий 2, и в проекте TEXTBOXES, п. 5.3, комментарий 3), поскольку аналогичное связывание в xaml-файле (в виде атрибута Slider.ValueChanged="slider1_ValueChanged") привело бы к ошибке времени выполнения. Действительно, при указании данного атрибута в xaml-файле обработчик slider1_ValueChanged был бы запущен первый раз уже при создании ползунка 1 (поскольку в нем явно задается начальное значение Value, равное 255), но в этот момент *еще не созданы* другие компоненты окна, используемые в обработчике slider1_ValueChanged (ползунки 2, 3, 4 и метка caption1), поэтому попытка обращения к их свойствам приведет к исключению NullReferenceException.

4. Вызов метода AddHandler имеет в данном случае одну особенность: во втором параметре необходимо использовать конструктор *обобщенного* класса, параметризуемого типом double. Эта особенность обусловлена тем обстоятельством, что для различных компонентов связанные с ними свойства Value могут иметь разные типы.

14.3. Инвертирование цветов и вывод цветовых констант

```
<Label x:Name="caption1" Background ="Black"
    Foreground="White" Content="Color"
    HorizontalContentAlignment="Center"
    VerticalContentAlignment="Center" />
```

Добавьте в метод slider1_ValueChanged следующие операторы:

```
var c = (caption1.Background as SolidColorBrush).Color;
caption1.Foreground = new
  SolidColorBrush(Color.FromRgb((byte)(0xFF ^ c.R),
  (byte)(0xFF ^ c.G), (byte)(0xFF ^ c.B)));
caption1.Content = c.ToString();
```

Результат. Числовое значение текущего цвета в формате ARGB (Alpha–Red–Green–Blue) отображается как текст метки caption1 в виде шестнадцатеричного числа, снабженного префиксом #; при этом каждой цветовой составляющей соответствует *два знака*, а буквы A–F (обозначающие шестнадцатеричные цифры от 10 до 15) изображаются в верхнем регистре. Например, значение цвета Maroon (непрозрачный темно-красный цвет интенсивности 128) имеет вид #FF800000, а значение полностью прозрачного черного цвета имеет вид #00000000. Цвет текста является непрозрачным и *инверсным* по отношению к цвету фона метки.

Комментарии

1. Для получения цвета кисти типа SolidColorBrush достаточно обратиться к ее свойству Color (доступному и для чтения, и для записи).

2. Свойства A, R, G, B класса Color (в библиотеке WPF, в отличие от библиотеки Windows Forms, они доступны и для чтения, и для записи) позволяют получить числовое значение соответствующей цветовой составляющей. Для инвертирования каждого из базовых цветов использована побитовая операция ^ («исключающее ИЛИ»). При использовании варианта метода FromRgb с *тремя* параметрами (R, G, B) предполагается, что прозрачность A равна 255.

Недочет. При запуске программы метка caption1 содержит текст «Color», а не числовое значение непрозрачного черного цвета.

Исправление. В конструктор класса MainWindow добавьте оператор:

```
slider1_ValueChanged(null, null);
```

Результат. Теперь обработчик slider1_ValueChanged вызывается в момент создания окна (после создания всех его компонентов), что обеспечивает правильную настройку внешнего вида метки caption1.

Комментарий

Отмеченный недочет можно было бы исправить, просто задав в xaml-файле значение свойства Content метки caption1 равным "#FF000000"

(числовое значение непрозрачного черного цвета). Однако использованный вариант исправления является более гибким, поскольку позволяет правильно отображать начальный вид метки *при любых изменениях* метода slider1_ValueChanged, которые могут быть сделаны впоследствии (см. далее п. 14.5).

14.4. Отображение оттенков серого цвета

```
<Window x:Class="COLORS.MainWindow"
    … >
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    …
    <Label x:Name="label4" Grid.Row="3" Margin="10"
        Content="Blue" />
    <Label x:Name="label5" Grid.Row="4" Margin="10"
        Content="Gray" />
    …
    <Slider x:Name="slider4" Grid.Column="1" Grid.Row="3"
        Margin="10" SmallChange="1" LargeChange="32"
        Maximum="255" TickFrequency="32" TickPlacement="Both" />
    <Slider x:Name="slider5" Grid.Column="1" Grid.Row="4"
        Margin="10" SmallChange="1" LargeChange="32"
        Maximum="255" TickFrequency="32" TickPlacement="Both"
        ValueChanged="slider5_ValueChanged" />
    …
  </Grid>
</Window>
```

```
private void slider5_ValueChanged(object sender,
  RoutedPropertyChangedEventArgs<double> e)
{
    slider2.Value = slider3.Value = slider4.Value =
      slider5.Value;
}
```

Результат. Перемещение добавленного пятого ползунка обеспечивает синхронное изменение всех трех базовых цветов, давая в итоге различные оттенки серого цвета (значение прозрачности при этом не изменяется).

Недочет. При выполнении обработчика slider5_ValueChanged метод slider1_ValueChanged вызывается *четыре раза*. В этом легко убедиться, добавив в начало метода slider1_ValueChanged оператор

```
Title += "!";
```

а в начало метода slider5_ValueChanged – оператор

```
Title = "";
```

В результате при любом изменении положения пятого ползунка в заголовке окна будут выводиться *четыре* восклицательных знака.

Данный недочет объясняется тем, что в обработчике slider5_ValueChanged изменяются значения свойства Value для трех ползунков, поэтому при каждом таком изменении будет вызван связанный с ним обработчик. Последний, четвертый вызов происходит для пятого ползунка, поскольку на него распространяется действие обработчика slider1_ValueChanged (так как ползунок slider5 тоже является дочерним компонентом окна Window). При изменении значения Value пятого ползунка вначале вызывается обработчик slider5_ValueChanged, связанный непосредственно с этим компонентом, а затем событие ValueChanged перенаправляется родителям этого компонента, в частности окну, к которому методом AddHandler был присоединен обработчик slider1_ValueChanged. Обнаружив, что для пятого ползунка произошло событие ValueChanged, окно вызывает для него этот обработчик.

Исправление. Откорректируйте метод slider5_ValueChanged следующим образом:

```
private void slider5_ValueChanged(object sender,
  RoutedPropertyChangedEventArgs<double> e)
{
    RemoveHandler(Slider.ValueChangedEvent,
      new RoutedPropertyChangedEventHandler<double>
      (slider1_ValueChanged));
    slider2.Value = slider3.Value = slider4.Value =
      slider5.Value;
    AddHandler(Slider.ValueChangedEvent,
      new RoutedPropertyChangedEventHandler<double>
      (slider1_ValueChanged));
}
```

Результат. Если теперь протестировать количество вызовов метода slider1_ValueChanged, добавив в начало обработчиков события ValueChanged указанные выше (при описании недочета) операторы, то можно убедиться, что при изменении пятого ползунка в заголовке окна выводится

единственный восклицательный знак, что означает единственный вызов обработчика slider1_ValueChanged.

Комментарий

Метод RemoveHandler выполняет действие, обратное действию метода AddHandler: он *отсоединяет* обработчик от указанного события, причем, в отличие от операции -=, позволяет это делать даже для тех событий, которые отсутствуют у данного компонента. Благодаря этому отсоединению ни одно из событий, связанных с изменением свойств Value для ползунков 2, 3 и 4, не приводит к вызову обработчика slider1_ValueChanged. После повторного присоединения обработчика к событию, выполненного в конце метода slider5_ValueChanged, метод slider1_ValueChanged вызывается для ползунка 5, так как к моменту вызова метода AddHandler обработка события ValueChanged для этого ползунка *еще не завершилась* (в частности, информация об этом событии еще не была передана родительским компонентам).

Напомним, что подавить вызов обработчиков событий, определенных в родительских компонентах, можно, выполнив оператор e.Handled = true. Если добавить этот оператор в обработчик slider5_ValueChanged, то при изменении положения ползунка 5 цвет метки caption1 *вообще не будет изменяться*.

14.5. Вывод цветовых имен

Добавьте в описание класса MainWindow новое поле, сразу указав для него конструктор:

```
Dictionary<Color, string> colorName =
  new Dictionary<Color, string>(141);
```

В конструктор класса MainWindow *перед* вызовом метода slider1_ValueChanged(null, null) добавьте следующий фрагмент:

```
foreach (System.Reflection.PropertyInfo pi in
  typeof(Colors).GetProperties())
    colorName[(Color)pi.GetValue(null, null)] = pi.Name;
```

Откорректируйте метод slider1_ValueChanged следующим образом:

```
private void slider1_ValueChanged(object sender,
  RoutedPropertyChangedEventArgs<double> e)
{
    caption1.Background = new
      SolidColorBrush(Color.FromArgb((byte)slider1.Value,
      (byte)slider2.Value, (byte)slider3.Value,
      (byte)slider4.Value));
    var c = (caption1.Background as SolidColorBrush).Color;
    caption1.Foreground = new
      SolidColorBrush(Color.FromRgb((byte)(0xFF ^ c.R),
```

```
      (byte)(0xFF ^ c.G), (byte)(0xFF ^ c.B)));
    caption1.Content = c.ToString();
    var s = c.ToString();
    switch (c.A)
    {
        case 0:
            s += " Transparent";
            break;
        case 255:
            if (colorName.ContainsKey(c))
                s += " " + colorName[c];
            break;
    }
    caption1.Content = s;
}
```

Результат. В том случае, когда с текущим цветом связана определенное *имя* (например, «Black» или «Maroon»), текст метки caption1 содержит не только числовое значение текущего цвета в шестнадцатеричном формате, но и его имя. Если прозрачность имеет значение, равное 0, то рядом с числовым значением цвета выводится текст «Transparent».

Комментарий

Все цвета, имеющие имена (*именованные цвета*), содержатся в классе Colors, где для каждого именованного цвета предусмотрено свойство с соответствующим именем, возвращающее данный цвет (всего класс содержит 141 свойство – по количеству именованных цветов). Все именованные цвета являются полностью непрозрачными.

В начале нашей программы мы формируем вспомогательный словарь colorName (типа Dictionary<Color, string>), ключами которого являются значения именованных цветов (типа Color), а значениями – имена этих цветов (типа string). Для формирования словаря используется механизм *отражения* (reflection), благодаря которому для любого класса .NET можно получить полную информацию о его членах. Мы используем механизм отражения для получения информации обо всех свойствах класса Colors (в проекте CURSORS, п. 8.1, мы аналогичным образом получили информацию обо всех свойствах класса Cursors).

В дальнейшем мы применяем сформированный словарь colorName в методе slider1_ValueChanged для проверки того, имеет ли текущий цвет имя, и получения этого имени.

14.6. Связывание компонентов с метками-подписями

При редактировании xaml-файла обратите внимание на добавление символов подчеркивания к именам меток.

```
<Window x:Class="COLORS.MainWindow" … >
  <Grid>
    …
    <Label x:Name="label1" Margin="10" Content="_Alpha"
        Target="{Binding ElementName=slider1}" />
    <Label x:Name="label2" Grid.Row="1" Margin="10"
        Content="_Red" Target="{Binding ElementName=slider2}" />
    <Label x:Name="label3" Grid.Row="2" Margin="10"
        Content="_Green" Target="{Binding ElementName=slider3}" />
    <Label x:Name="label4" Grid.Row="3" Margin="10"
        Content="_Blue" Target="{Binding ElementName=slider4}" />
    <Label x:Name="label5" Grid.Row="4" Margin="10"
        Content="Gra_y" Target="{Binding ElementName=slider5}" />
    …
    </Grid>
</Window>
```

Результат. Переключение между ползунками теперь можно осуществлять с помощью Alt-комбинаций символов, подчеркнутых в подписях к ползункам (Alt+A для ползунка, определяющего прозрачность, Alt+R для ползунка, определяющего интенсивность красного цвета, и т. д.). Если при запуске программы подчеркивание не отображается, следует нажать клавишу Alt.

Комментарии

1. Сами метки (компоненты Label) не могут принимать фокус. Однако с меткой можно связать компонент, который будет получать фокус, если нажата Alt-комбинация, связанная с данной меткой. Для указания компонента, с которым требуется связать метку, предусмотрено свойство Target, которое можно задать непосредственно в xaml-файле, используя специальный синтаксис расширения разметки для указания *привязки* свойств (по поводу привязки свойств и расширения разметки см. проект TEXTEXIT версии 4, п. 12.3). Следует отметить, что именно возможность передачи фокуса другим компонентам является одним из основных отличий меток Label от похожих на них по функциональности компонентов TextBlock.

2. Завершая описание данного проекта, заметим, что, согласно [7, с. 135], первый вариант подобной программы для среды Windows 1.0 был разработан Чарльзом Петцольдом в 1987 году.

15. Выпадающие и обычные списки: LISTBOXES

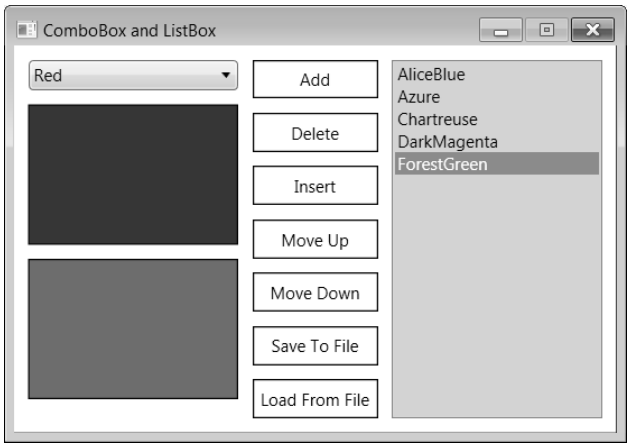


Рис. 46. Окно приложения LISTBOXES

15.1. Создание и использование выпадающих списков

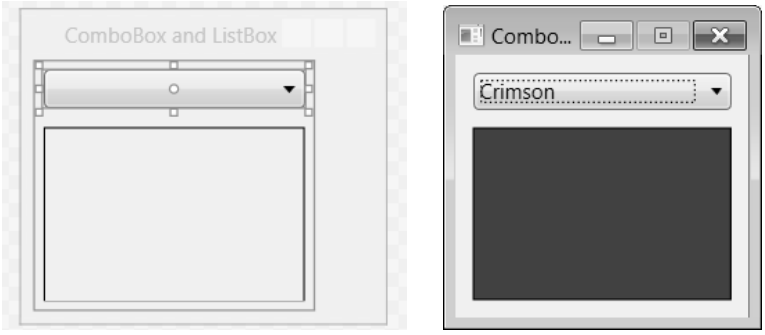


Рис. 47. Макет окна LISTBOXES (первый вариант) и его вид при запуске приложения

```
<Window x:Class="LISTBOXES.MainWindow"
    …
    Title="ComboBox and ListBox" SizeToContent="WidthAndHeight"
    WindowStartupLocation="CenterScreen"
    ResizeMode="CanMinimize">
  <StackPanel Margin="5" Orientation="Horizontal" >
    <StackPanel >
      <ComboBox x:Name="comboBox1" Margin="5" MinWidth="150"
          SelectionChanged="comboBox1_SelectionChanged" />
      <Rectangle x:Name="rect1" Margin="5" Height="100"
          Stroke="Black" />
    </StackPanel>
  </StackPanel>
</Window>
```

В классе MainWindow определите внутренний вспомогательный класс NamedBrush:

```
class NamedBrush
{
    static Dictionary<string, SolidColorBrush> colorNames =
      new Dictionary<string, SolidColorBrush>(141);
    string name;
    static NamedBrush()
    {
        foreach (System.Reflection.PropertyInfo pi in
          typeof(Colors).GetProperties())
            colorNames[pi.Name] = new
              SolidColorBrush((Color)pi.GetValue(null, null));
    }
    NamedBrush(string n)
    {
        name = n;
    }
    public SolidColorBrush Brush
    {
        get { return colorNames[name]; }
    }
    public string Name
    {
        get { return name; }
    }
    public override string ToString()
    {
        return name;
    }
    public static IEnumerable<NamedBrush> AllNamedBrushes()
    {
        return colorNames.Select(e => new NamedBrush(e.Key));
    }
    public static Brush GetBrush(string name)
    {
        return colorNames.ContainsKey(name ?? "") ?
          colorNames[name] : null;
     }
}
```

В конструктор класса MainWindow добавьте операторы:

```
comboBox1.ItemsSource = NamedBrush.AllNamedBrushes();
comboBox1.SelectedValuePath = "Brush";
comboBox1.SelectedIndex = 0;
comboBox1.Focus();
```

Определите обработчик события SelectionChanged для компонента comboBox1, уже указанный в xaml-файле:

```
private void comboBox1_SelectionChanged(object sender,
  SelectionChangedEventArgs e)
{
    rect1.Fill = (Brush)comboBox1.SelectedValue;
}
```

Результат. При запуске программы в выпадающем списке содержатся значения идентификаторов для всех именованных цветов. Идентификаторы указаны в алфавитном порядке. Выбор идентификатора из списка приводит к соответствующему изменению цвета прямоугольника rect1. Для перебора пунктов выпадающего списка можно использовать клавиши со стрелками, а также клавиши Home и End. Кроме того, нажатие буквенной клавиши приводит к выбору первого пункта с текстом, начинающимся с этой буквы. Для разворачивания списка можно использовать комбинацию Alt+↓ (однако данная комбинация срабатывает только при использовании той клавиши со стрелкой, которая *не находится* на цифровом блоке).

Следует заметить, что при выборе начального цвета AliceBlue (первого в списке comboBox1) прямоугольник rect1 остается практически белым, поскольку этот цвет является очень светлым.

Комментарии

1. Рассматриваемые в данном проекте компоненты ComboBox (выпадающий список) и ListBox (обычный список, который будет добавлен к проекту в следующем пункте) относятся к особой категории *списочных компонентов*, предназначенных для хранения однотипных объектов (элементов, items) в виде списков (простых или иерархических). В трех следующих проектах (CHECKBOXES, IMGVIEW и TRIGFUNC) тоже будут применяться различные виды списочных компонентов. Все они – потомки класса ItemsControl, который, в свою очередь, является потомком класса Control. Основным свойством, наследуемым всеми потомками класса ItemsControl, является свойство-коллекция Items типа ItemCollection с элементами типа object, в котором хранятся элементы списка. Тот факт, что элементы данной коллекции имеют тип object, означает, что список может включать элементы любого типа, хотя в простейших вариантах списков обычно используются строки.

2. Мы использовали вариант настройки компонента ComboBox, основанный на применении свойства ItemsSource. Этот вариант требует предварительных действий по созданию коллекции объектов подходящего типа, однако он более удобен для последующего использования по сравнению с традиционным вариантом, основанным на явном формировании коллекции Items как набора строк (традиционный вариант будет использован в дальнейшем при работе с компонентом ListBox). Оба варианта настройки доступны и для обычных, и для выпадающих списков.

Свойство ItemsSource для списков можно не определять. Если же его положить равным некоторой последовательности объектов (в нашем случае объектов типа NamedBrush), то в список будут автоматически включены *названия* этих объектов (полученные с помощью метода ToString). Кроме того, с каждым пунктом списка, помимо его названия, будет связано его *значение* (value), равное одному из свойств соответствующего элемента последовательности ItemsSource. Это дает возможность в дальнейшем для доступа к текущему элементу списка использовать не только свойства SelectedIndex (равное *индексу* текущего элемента) и SelectedItem (равное *названию* текущего элемента, которое берется из коллекции Items), но и SelectedValue (равное *значению* текущего элемента). Благодаря данной возможности код обработчика comboBox1_SelectionChanged удалось сделать очень кратким и наглядным. Для объекта, возвращаемого свойством SelectedValue, требуется явное приведение типа (в нашем случае достаточно выполнить приведение к типу Brush – абстрактному предку всех классов, определяющих *кисти* WPF).

Обратите внимание на то, что при задании свойства ItemsSource необходимо одновременно определить свойство SelectedValuePath, указав в нем имя того свойства элементов последовательности ItemsSource, которое будет использоваться при вычислении *значения* элементов списка. В нашем случае было указано свойство Brush.

Если требуется *изменить* набор пунктов списка, в котором определено свойство ItemsSource, то достаточно присвоить этому свойству новую (измененную) последовательность элементов. При этом также потребуется повторно задать значение свойства SelectedIndex, так как изменение свойства ItemsSource автоматически полагает свойство SelectedIndex равным –1, что означает *отсутствие* выбранного элемента в списке. Заметим также, что в случае отсутствия выбранного элемента свойства SelectedItem и SelectedValue возвращают значение null.

2. Для того чтобы можно было применять дополнительные возможности, связанные со свойством ItemsSource, обычно требуется определить вспомогательный класс. Созданный для этих целей класс NamedBrush демонстрирует много различных аспектов, связанных с объектной моделью языка C#. Он включает следующие члены:

- *статическое поле* colorNames – словарь, ключами которого являются стандартные названия цветов, а значениями – кисти SolidColorBrush соответствующего цвета;
- *поле* name типа string, содержащее название цвета (данное поле, в отличие от поля colorNames, является *экземплярным*, т. е. определяется не для класса в целом, а для каждого его экземпляра по отдельности);
- *статический конструктор* static NamedBrush(), в котором формируется словарь colorName (для его формирования используется механизм отражения, ранее применявшийся в проектах CURSORS и COLORS);
- *конструктор* NamedBrush(string n), создающий объект с названием цвета n (обратите внимание на то, что данный конструктор, как и все перечисленные выше члены класса, является *закрытым*, т. е. его нельзя вызвать вне класса NamedBrush);
- связанное с полем name открытое *свойство* Name, доступное только для чтения;
- открытое свойство Brush типа SolidColorBrush, доступное только для чтения и возвращающее кисть цвета Name;
- открытый *метод* ToString, возвращающий значение свойства Name (обратите внимание на его модификатор override);
- открытый *статический метод* AllNamedBrushes, создающий коллекцию объектов типа NamedBrush, связанных со всеми стандартными именованными цветами;
- вспомогательный открытый статический метод GetBrush(string name), позволяющий по имени name определить соответствующую кисть (если имя name не является именем стандартного цвета, то метод возвращает null).

Метод GetBrush класса NamedBrush в нашей программе пока не используется; он окажется полезным при работе с обычным списком, который будет добавлен в программу в следующем пункте. Поскольку метод ContainsKey не может принимать параметр, равный null (в то время как параметр name может иметь такое значение), мы используем в методе GetBrush операцию ??, возвращающую свой первый операнд, если он не равен null, и второй операнд в противном случае.

В статическом методе AllNamedBrushes используется запрос Select (один из стандартных запросов LINQ – см. [4, гл. 10], [5, гл. 9]), который применяется к статическому полю colorNames.

3. Внешняя (горизонтальная) панель StackPanel пока содержит единственный элемент – внутреннюю (вертикальную) панель. В дальнейшем в нее добавятся новые дочерние компоненты, которые будут отображаться во втором и третьем столбце окна.

4. Выпадающий список ComboBox является в нашей программе *неизменяемым*, т. е. не допускающим редактирование отображаемого в нем текста. Можно разрешить подобное редактирование, положив свойство IsEditable компонента ComboBox равным true. В этом случае для компонента ComboBox будут доступны возможности как выпадающего списка, так и обычного поля ввода. В частности, можно будет получить доступ к введенному тексту с помощью свойства Text (для неизменяемого списка свойство Text возвращает название выбранного элемента или пустую строку "", если выбранный элемент отсутствует). Интересно, что попытка ввода текста, соответствующего начальной части названия какого-либо элемента выпадающего списка, обеспечит немедленный выбор этого элемента, сопровождаемое «автозавершением» ввода этого названия (причем предложенный вариант автозавершения можно отменить, продолжив ввод текста). Для демонстрации указанных возможностей достаточно добавить в xaml-файл нашей программы атрибут IsEditable="True" для элемента ComboBox.

5. Для связывания фона прямоугольника rect1 со значением текущего элемента выпадающего списка можно использовать *привязку* – механизм, который в WPF считается более предпочтительным, чем применение обработчиков. В нашем случае вместо определения обработчика comboBox1_SelectionChanged достаточно дополнить элемент Rectangle в xaml-файле еще одним атрибутом со значением, заданным в формате *расширения разметки* (см. проекты TEXTEDIT версии 4, п. 12.3, и TEXTEDIT версии 5, п. 13.1, где привязка обсуждается более подробно):

```
<Rectangle x:Name="rect1" Margin="5" Height="100" Stroke="Black"
    Fill="{Binding ElementName=comboBox1, Path=SelectedValue}" />
```

15.2. Список: добавление и удаление элементов

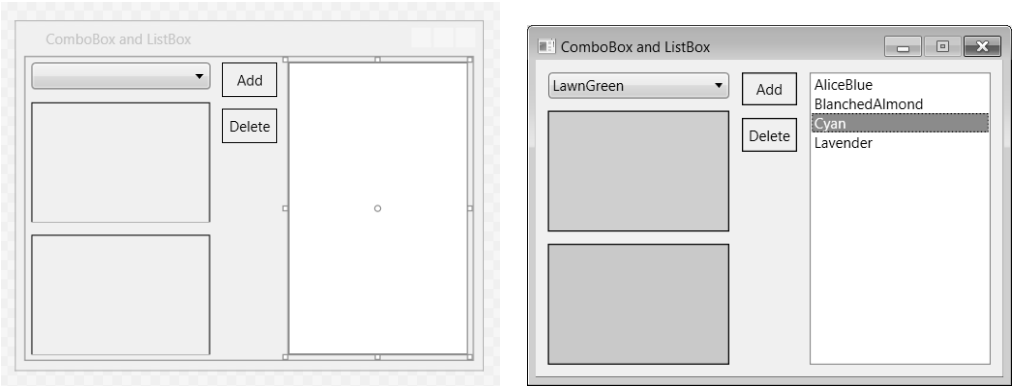


Рис. 48. Макет окна LISTBOXES (второй вариант) и его вид при запуске приложения

```
<Window x:Class="LISTBOXES.MainWindow"
    … >
  <StackPanel Margin="5" Orientation="Horizontal" >
```

```
    <StackPanel >
      <ComboBox x:Name="comboBox1" Margin="5" MinWidth="150"
          SelectionChanged="comboBox1_SelectionChanged" />
      <Rectangle x:Name="rect1" Margin="5" Height="100"
          Stroke="Black" />
      <Rectangle x:Name="rect2" Margin="5" Height="100"
          Stroke="Black" />
    </StackPanel>
    <StackPanel >
      <Label x:Name="label1" Content="Add"
          Padding="5" Margin="5" BorderBrush="Black"
          BorderThickness="1" HorizontalContentAlignment="Center"
          MouseDown="label1_MouseDown" />
      <Label x:Name="label2" Content="Delete"
          Padding="5" Margin="5" BorderBrush="Black"
          BorderThickness="1" HorizontalContentAlignment="Center"
          MouseDown="label2_MouseDown" />
    </StackPanel>
    <ListBox x:Name="listBox1" Margin="5" MinWidth="150"
          SelectionChanged="listBox1_SelectionChanged" />
  </StackPanel>
</Window>
```

Определите обработчики события SelectionChanged для списка listBox1 и событий MouseDown для меток label1 и label2 (уже указанные в xaml-файле):

```
private void listBox1_SelectionChanged(object sender,
  SelectionChangedEventArgs e)
{
    rect2.Fill =
      NamedBrush.GetBrush((string)listBox1.SelectedItem);
}
private void label1_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    listBox1.SelectedIndex = listBox1.Items.Add(comboBox1.Text);
}

private void label2_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    listBox1.Items.RemoveAt(listBox1.SelectedIndex);
```

```
}
```

Результат. При щелчке на метке «Add» в список listBox1 добавляется строка из выпадающего списка comboBox1. Фон прямоугольника rect2 соответствует выделенному элементу списка listBox1. Если список является пустым, то прямоугольник rect2 сохраняет белый фон. При щелчке на метке «Delete» из списка удаляется выделенный элемент.

Комментарии

1. *Выделенный* элемент списка изображается на цветном (обычно синем) фоне. Если список имеет фокус (т. е. является активным компонентом окна) *и при этом хотя бы раз для переключения между компонентами окна использовалась клавиша Tab* (или была нажата клавиша Alt), то выделенный элемент дополнительно обводится пунктирной рамкой. Элемент, обведенный такой рамкой, называется *текущим*.

В некоторых библиотеках, связанных с разработкой интерфейса, имеется возможность различать текущий и выделенный элемент списка; такая возможность оказывается очень полезной, если список может содержать несколько выделенных элементов. Однако, хотя в библиотеке WPF список ListBox имеет свойство SelectionMode, позволяющее устанавливать режим множественного выделения, отмеченной выше возможности определять текущий элемент списка в этой библиотеке не предусмотрено (следует отметить, что такая возможность отсутствовала и в библиотеке Windows Forms). Иными словами, если в списке выделено несколько элементов, то программными средствами WPF можно узнать, какие элементы выделены, но отсутствуют простые способы определить, какой элемент является текущим, т. е. обведен пунктирной рамкой (заметим, что в режиме множественного выделения текущий элемент не обязан быть выделенным). Данное обстоятельство затрудняет работу со списками, имеющими множественное выделение, поскольку не позволяет программно реагировать на изменение их важной характеристики – позиции текущего элемента.

При разработке программ в WPF (и в Windows Forms) предпочтительнее вместо списков со множественным выделением использовать *списки флажков* с единичным выделением (см. проект CHECKBOXES); при этом, наряду с возможностью работы с текущим (и одновременно выделенным) элементом, пользователь получает возможность *помечать* любое количество элементов списка, устанавливая в них флажки (таким образом, при использовании списка флажков программе доступна вся информация о состоянии элементов: как текущего, так и помеченных).

2. Для изучения особенностей поведения текущего (обведенного рамкой) элемента списка при выполнении различных операций необходимо, чтобы список *не терял фокуса*. Поэтому выполнение операций мы связали с метками Label: эти компоненты, в отличие от обычных кнопок

Button, не могут получать фокус (во всяком случае, таковы их свойства по умолчанию). Таким образом, в окне располагаются лишь два компонента, которые могут получать фокус: это comboBox1 и listBox1. Для переключения между ними достаточно использовать клавишу Tab.

Между прочим, указанная возможность позволила выявить недостаток в реализации компонента ListBox, связанный с взаимодействием выделенного и текущего элементов (напомним, что текущий элемент обводится пунктирной рамкой): при добавлении нового элемента выделение на него переносится, а пунктирная рамка нет (соответствующая ситуация приведена на рис. 49).

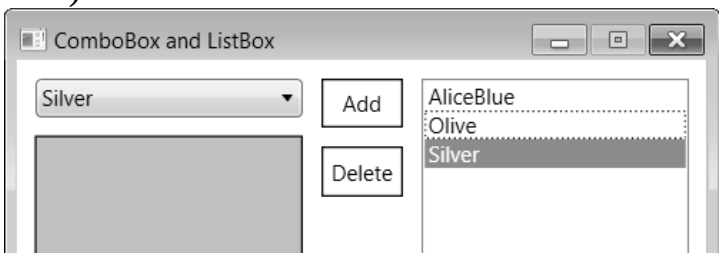


Рис. 49. Пример рассогласования текущего и выделенного элемента списка

Впрочем, подобная ситуация проявляется редко, поскольку для этого необходимо, во-первых, чтобы в момент добавления нового элемента список имел фокус, и, во-вторых, чтобы в нем отображалась пунктирная рамка, что происходит только если ранее для работы с программой (например, для изменения выделенного элемента или переключения между компонентами) использовалась клавиатура. Кроме того, эта ситуация никак не влияет на последующую работу со списком.

Недочет 1. Первая неприятность в нашей программе может возникнуть уже после добавления новых элементов в список listBox1. При использовании многих стандартных тем Windows 7 потеря фокуса списком listBox1 (например, в результате нажатия клавиши Tab или щелчка на выпадающем списке comboBox1) приводит к тому, что выделенный элемент отображается на таком светлом фоне, что на некоторых мониторах оказывается очень сложно отличить его от других, невыделенных элементов (рис. 50, изображение левого окна).

Рис. 50. Вид неактивного списка для различных тем Windows 7

Заметим, что при установке темы «Классическая» ситуация меняется: при потере фокуса списком выделенный элемент отображается на сером фоне более темного, и поэтому вполне различимого, оттенка (рис. 50, изображение правого окна).

Исправление. Добавьте в описание компонента listBox1 в xaml-файле новый атрибут:

```
<ListBox x:Name="listBox1" Margin="5" MinWidth="150"
    Background="LightGray"
    SelectionChanged="listBox1_SelectionChanged" />
```

Результат. Теперь элементы списка отображаются на сером фоне, а при потере фокуса выделенный элемент становится более светлым (рис. 51). Благодаря данному исправлению мы можем легко отличить выделенный элемент списка даже в случае, когда список не имеет фокуса.

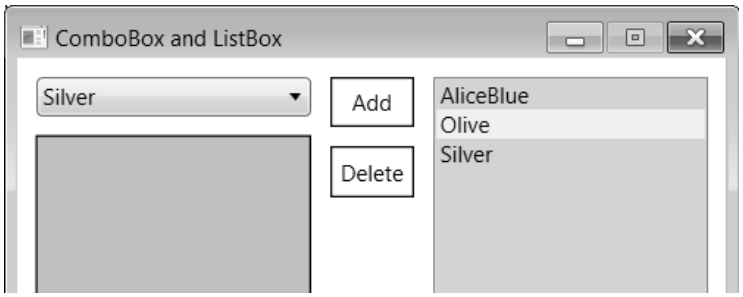


Рис. 51. Вид неактивного списка в случае использования серого фона

Комментарий

Впрочем, теперь возникнет проблема при использовании темы «Классическая». Выбранный нами серый цвет фона будет очень близок к используемому в этой теме цвету выделенного элемента неактивного списка. Эту проблему можно решить, если в качестве фона Background задать еще более темный серый цвет, например "#FFBBBBBB".

Заметим также, что при использовании библиотеки Windows Forms отмеченный недочет невозможен в принципе, так как в этой библиотеке выделенные элементы как активного, так и неактивного списка *отображаются одинаковым образом*.

Недочет 2. После выполнения операции удаления в списке *исчезает выделенный элемент* (а фон прямоугольника rect2 становится белым). В этой ситуации свойство SelectedIndex равно –1, поэтому повторный щелчок на метке «Delete» приводит к ошибке времени выполнения, так как значение –1 не является допустимым для метода RemoveAt. По этой же причине к ошибке приводит щелчок на метке «Delete» в случае *пустого* списка.

Исправление. Измените метод label2_MouseDown:

```
private void label2_MouseDown(object sender,
  MouseButtonEventArgs e)
```

```
{
    int i = listBox1.SelectedIndex;
    if (i == -1)
    {
        Console.Beep();
        return;
    }
    listBox1.Items.RemoveAt(i);
    if (i == listBox1.Items.Count)
        i--;
    listBox1.SelectedIndex = i;
}
```

Результат. При удалении элемента в *середине* списка выделение сохраняется на текущей позиции (которую теперь занимает следующий элемент). При удалении элемента в *конце* списка выделяется предыдущий элемент. Таким образом, *если список не пуст, он всегда имеет выделенный элемент*. При нажатии кнопки «Delete» в случае пустого списка выдается звуковой сигнал.

Комментарии

1. Наряду с методом Beep без параметров класс Console имеет вариант данного метода с двумя параметрами типа int, первый из которых определяет *частоту* звука (в диапазоне от 37 до 32767 герц), а второй – *продолжительность* звучания (в миллисекундах).

2. В процессе удаления элемента из списка (между вызовом метода RemoveAt и заданием нового значения для свойства SelectedIndex) свойство SelectedIndex остается равным –1. Поэтому для правильной работы программы важно, чтобы данная ситуация не приводила к исключению в методе listBox1_SelectionChanged. У нас в этом отношении все обстоит благополучно: если список не содержит выделенного элемента, то его свойство SelectedItem возвращает значение null, при обработке которого метод GetBrush тоже возвращает null, не возбуждая никаких исключений.

15.3. Дополнительные операции для элементов списка. Использование стилей в xaml-файле

```
<Window x:Class="LISTBOXES.MainWindow"
    … >
  <StackPanel Margin="5" Orientation="Horizontal" >
    …
    <StackPanel >
      <StackPanel.Resources>
        <Style x:Key="label">
```

```
            <Setter Property="Control.Padding" Value="5" />
            <Setter Property="Control.Margin" Value="5" />
            <Setter Property="Control.BorderBrush" Value="Black" />
            <Setter Property="Control.BorderThickness" Value="1" />
            <Setter Property="Control.HorizontalContentAlignment"
                Value="Center" />
          </Style>
        </StackPanel.Resources>
        <Label x:Name="label1" Content="Add"
            Padding="5" Margin="5" BorderBrush="Black"
            BorderThickness="1" HorizontalContentAlignment="Center"
            MouseDown="label1_MouseDown"
            Style="{StaticResource label}" />
        <Label x:Name="label2" Content="Delete"
            Padding="5" Margin="5" BorderBrush="Black"
            BorderThickness="1" HorizontalContentAlignment="Center"
            MouseDown="label2_MouseDown"
            Style="{StaticResource label}" />
        <Label x:Name="label3" Content="Insert"
            MouseDown="label3_MouseDown"
            Style="{StaticResource label}" />
        <Label x:Name="label4" Content="Move Up"
            MouseDown="label4_MouseDown"
            Style="{StaticResource label}" />
        <Label x:Name="label5" Content="Move Down"
            MouseDown="label5_MouseDown"
            Style="{StaticResource label}" />
        <Label x:Name="label6" Content="Save To File"
            MouseDown="label6_MouseDown"
            Style="{StaticResource label}" />
        <Label x:Name="label7" Content="Load From File"
            MouseDown="label7_MouseDown"
            Style="{StaticResource label}" />
      </StackPanel>
      <ListBox x:Name="listBox1" … />
    </StackPanel>
</Window>
```

В конец списка директив using в файле MainWindow.xaml.cs добавьте директиву

```
using System.IO;
```

Обработчики событий MouseDown для новых меток label3–label7 определите следующим образом:

```
private void label3_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    int i = listBox1.SelectedIndex;
    if (i == -1)
        label1_MouseDown(null, null);
    else
    {
        listBox1.Items.Insert(i, comboBox1.Text);
        listBox1.SelectedIndex = i;
    }
}
private void label4_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    int i = listBox1.SelectedIndex;
    if (i <= 0)
    {
        Console.Beep();
        return;
    }
    var x = listBox1.Items[i];
    listBox1.Items[i] = listBox1.Items[i - 1];
    listBox1.Items[i - 1] = x;
    listBox1.SelectedIndex = i - 1;
}
private void label5_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    int i = listBox1.SelectedIndex;
    if (i == -1 || i == listBox1.Items.Count - 1)
    {
        Console.Beep();
        return;
    }
    var x = listBox1.Items[i];
    listBox1.Items[i] = listBox1.Items[i + 1];
    listBox1.Items[i + 1] = x;
```

```
    listBox1.SelectedIndex = i + 1;
}
private void label6_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    if (listBox1.Items.Count == 0)
    {
        Console.Beep();
        return;
    }
    File.WriteAllLines("LISTBOXES.dat",
      listBox1.Items.Cast<string>());
}
private void label7_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    if (!File.Exists("LISTBOXES.dat"))
    {
        Console.Beep();
        return;
    }
    listBox1.Items.Clear();
    foreach (var e1 in File.ReadLines("LISTBOXES.dat"))
        listBox1.Items.Add(e1);
    listBox1.SelectedIndex = listBox1.Items.Count - 1;
}
```

Результат. Метка «Insert» обеспечивает вставку нового элемента перед выделенным и выделяет вставленный элемент (в случае пустого списка метка «Insert» действует аналогично метке «Add»). Метки «Move Up» и «Move Down» перемещают выделенный элемент вверх и вниз по списку, сохраняя его выделение (при попытке выполнить перемещение первого элемента вверх или последнего элемента вниз выдается звуковой сигнал). Метки «Save To File» и «Load From File» позволяют сохранить *непустой* список в файле LISTBOXES.dat и впоследствии загрузить список из этого файла (если файл отсутствует, то при попытке загрузить данные выдается звуковой сигнал).

Комментарии

1. Многие свойства добавленных меток имеют совпадающие значения (Padding, Margin, BorderBrush, BorderThickness и HorizontalContentAlignment). Вместо того чтобы задавать явным образом все эти свойства в каждой метке, мы определили в компоненте StackPanel – родителе всех

меток – *стиль* с ключом label, указав в нем имена и значения требуемых свойств и оформив его как *статический ресурс* данного компонента. Применение стилей позволяет уменьшить размер xaml-файла и упрощает его последующее редактирование.

Стили xaml-файла являются частным случаем его *ресурсов* (ранее ресурсы XAML использовались в версии 5 проекта TEXTEDIT). В библиотеке WPF имеется большое количество стандартных ресурсов. В частности, в виде стандартных *динамических* ресурсов представлены системные кисти, которые удобно задавать с помощью окна Properties. Например, для указания системной кисти ControlBrush в качестве фона окна (и всех его компонентов, в которых свойство Background явно не переопределяется) достаточно щелкнуть мышью на поле ввода рядом с именем свойства Background, в появившемся окне выбора цвета щелкнуть на правой кнопке в верхнем ряду (для этой кнопки выводится подсказка «Brush resources») и в появившемся списке ресурсов, связанных с системными кистями, выбрать вариант ControlBrushKey. В результате свойство Background в окне свойств будет помечено зеленой меткой, а в список атрибутов элемента Window в xaml-файле будет добавлен новый атрибут:

```
Background="{DynamicResource {x:Static
    SystemColors.ControlBrushKey}}"
```

Подобная настройка позволяет адаптировать вид окна к любой теме Windows, поскольку имеются темы (например «Классическая» из набора тем Window 7), для которых фон по умолчанию отличен от белого.

2. При реализации новых действий были использованы методы Insert (вставка нового элемента в указанную позицию) и Clear (очистка списка элементов) коллекции Items. Следует заметить, что класс ItemCollection имеет не слишком богатый набор методов. В частности, в нем отсутствуют методы, позволяющие добавлять или вставлять в требуемую позицию *набор* элементов. Поэтому для добавления в коллекцию Items *всех* данных, сохраненных в файле LISTBOXES.DAT, нам пришлось использовать цикл foreach по всем элементам прочитанной из файла коллекции строк.

3. Для проверки существования файла использован статический метод Exists класса File. Для записи коллекции строк в текстовый файл и для считывания содержимого текстового файла в виде коллекции строк удобно использовать еще два статических метода класса File – WriteAllLines и ReadLines. Эти методы обеспечивают автоматическое открытие файла с указанным именем, выполнение для него требуемых действий и последующее закрытие.

Метод WriteAllLines требует указания набора записываемых строк или в виде *массива*, или в виде *последовательности* (типа IEnumerable<string>); последовательность требуемого типа можно полу-

чить из исходной коллекции Items с помощью обобщенного запроса Cast<string>.

При считывании строк из файла никакие преобразования выполнять не надо: достаточно организовать цикл foreach, где будут обработаны все строки, прочитанные из файла методом ReadLines. Заметим, что данный метод возвращает не массив, а *последовательность* строк, которая не хранится целиком в оперативной памяти (в силу особенностей реализации типа IEnumerable<T> фактически в цикле foreach выполняется *построчное* считывание данных из файла и их немедленная обработка).

15.4. Выполнение операций над списком с помощью мыши

```
<Window x:Class="LISTBOXES.MainWindow"
    … >
  <StackPanel Margin="5" Orientation="Horizontal" >
    <StackPanel >
      <ComboBox x:Name="comboBox1" Margin="5" MinWidth="150"
          SelectionChanged="comboBox1_SelectionChanged"
          PreviewMouseDown="comboBox1_PreviewMouseDown" />
      …
    </StackPanel>
    <StackPanel >
      …
    </StackPanel>
    <ListBox x:Name="listBox1" Margin="5" MinWidth="150"
        Background="LightGray"
        SelectionChanged="listBox1_SelectionChanged"
        MouseDoubleClick="label2_MouseDown" AllowDrop="True"
        PreviewMouseDown="listBox1_PreviewMouseDown"
        Drop="listBox1_Drop" />
  </StackPanel>
</Window>
```

Обратите внимание на то, что событие MouseDoubleClick для компонента listBox1 связывается с *уже имеющимся* обработчиком label2_MouseDown (и по этой причине мы не подчеркиваем имя обработчика).

В описание класса MainWindow добавьте новое поле, которое в режиме перетаскивания будет содержать индекс перетаскиваемого элемента списка:

```
int iSrc;
```

Определите добавленные в xaml-файл обработчики событий comboBox1_PreviewMouseDown, listBox1_PreviewMouseDown, listBox1_Drop, MouseDown, а также вспомогательные методы IsMouseOverTarget и IndexFromPoint:

```
static bool IsMouseOverTarget(Visual target, Point point)
{
    var bounds = VisualTreeHelper.GetDescendantBounds(target);
    return bounds.Contains(point);
}
static int IndexFromPoint(ListBox lb,
  Func<IInputElement,Point> getPos)
{
    for (int i = 0; i < lb.Items.Count; i++)
    {
        var lbi = lb.ItemContainerGenerator.ContainerFromIndex(i)
          as ListBoxItem;
        if (lbi == null) continue;
        if (IsMouseOverTarget(lbi, getPos((IInputElement)lbi)))
            return i;
    }
    return -1;
}
private void comboBox1_PreviewMouseDown(object sender,
        MouseButtonEventArgs e)
{
    if (e.ChangedButton == MouseButton.Right)
        DragDrop.DoDragDrop(comboBox1, comboBox1.Text,
          DragDropEffects.Copy);
}
private void listBox1_PreviewMouseDown(object sender,
        MouseButtonEventArgs e)
{
    if (e.ChangedButton == MouseButton.Right)
    {
        iSrc = IndexFromPoint(listBox1, e.GetPosition);
        if (iSrc != -1)
            DragDrop.DoDragDrop(listBox1,
              (string)listBox1.Items[iSrc],
              DragDropEffects.Move);
    }
}
private void listBox1_Drop(object sender, DragEventArgs e)
{
    string s = e.Data.GetData(typeof(string)) as string;
```

```
    int iTrg = IndexFromPoint(listBox1, e.GetPosition);
    if (e.AllowedEffects == DragDropEffects.Move)
        listBox1.Items.RemoveAt(iSrc);
    if (iTrg == -1)
        listBox1.SelectedIndex = listBox1.Items.Add(s);
    else
    {
        listBox1.Items.Insert(iTrg, s);
        listBox1.SelectedIndex = iTrg;
    }
}
```

Результат. При двойном щелчке на элементе списка происходит удаление этого элемента. Для перемещения элемента списка на новую позицию теперь можно использовать механизм Drag & Drop: достаточно «зацепить» любой (не обязательно выделенный) элемент *правой* кнопкой мыши и перетащить его на новое место. Текст из выпадающего списка comboBox1 также можно поместить в список с помощью перетаскивания правой кнопкой мыши. Если элемент перетаскивается на существующий элемент списка, то он *вставляется* в указанную позицию, а если элемент перетаскивается на свободную область списка, то он *добавляется* к списку. В любом случае он становится выделенным элементом.

При перетаскивании текста из выпадающего списка изображение курсора содержит символ «+», что является признаком режима перетаскивания Copy («Копировать»); при перетаскивании элемента списка на новое место курсор *не содержит* символа «+», что является признаком режима перетаскивания Move («Переместить»).

Комментарии

1. Выбор правой кнопки мыши в качестве «инициатора» режима перетаскивания связан с тем, что и выпадающий список comboBox1, и обычный список listBox1 должны стандартным образом реагировать на левую кнопку мыши: выпадающий список при щелчке левой кнопкой разворачивается, а в обычном списке выделяется тот элемент, на котором произведен щелчок. Кроме того, с двойным щелчком на списке listBox1 мы также связали особое действие (удаление элемента). Если бы режим перетаскивания активизировался по нажатию левой кнопки, то стандартные действия, связанные с левой кнопкой, было бы невозможно выполнить.

2. Методы, события и свойства, связанные с перетаскиванием, были ранее подробно описаны в примере ZOO, п. 7.1. Единственной дополнительной проблемой при организации перетаскивания является необходимость определения *индексов* элемента-источника и элемента-приемника списка. В библиотеке Windows Forms для решения этой проблемы доста-

точно использовать метод IndexFromPoint, входящий в класс ListBox и позволяющий определить индекс элемента списка, содержащего точку с указанными координатами. По непонятным причинам аналогичный стандартный метод в библиотеке WPF отсутствует, поэтому пришлось явно реализовать его аналог (метод IndexFromPoint и связанный с ним вспомогательный метод IsMouseOverTarget). Использованные в этих методах возможности WPF в нашей книге не изучаются, поэтому данные методы можно рассматривать как «черный ящик» (подобно другим стандартным методам), предоставляющий программе требуемую функциональность.

Для инициализации режима Drug & Drop используется обработчик события PreviewMouseDown, поскольку событие MouseDown переопределено для компонентов ComboBox и ListBox, и с ним нельзя связать свой обработчик.

Обратите внимание на то, что в программе не определяются обработчики событий DragEnter и DragOver для компонента listBox1, поскольку в библиотеке WPF при их отсутствии любой компонент, который может служить приемником (т. е. для которого свойство AllowDrop равно true), автоматически принимает источник, перетаскиваемый в любом режиме.

3. Благодаря использованию различных режимов перетаскивания (Copy и Move), по виду курсора можно определить, какой компонент является *источником данных* (для режима DragDropEffects.Copy это comboBox1, для режима DragDropEffects.Move – listBox1). Путем проверки текущего режима перетаскивания (с помощью свойства e.AllowedEffects) в начале метода listBox1_Drop определяется, надо ли удалять элемент-«источник» из списка (впрочем, здесь можно было обойтись и без использования свойства e.AllowedEffects; для этого достаточно при перетаскивании элемента из *выпадающего списка* присвоить полю iSrc какое-либо особое значение, например –1, а в начале метода listBox1_Drop проверить значение этого поля).

Недочет. В начале перетаскивания элемента из выпадающего списка курсор имеет вид запрещающего знака. В примере ZOO мы уже отмечали, что это нежелательно, так как может ввести в заблуждение пользователя, который решит, что он сделал что-то не так.

Исправление. Добавьте к элементу ComboBox в xaml-файле атрибут AllowDrop со значением true и определите *общий* обработчик для событий DravEnter и DragOver:

```
<ComboBox x:Name="comboBox1" Margin="5" MinWidth="150"
    SelectionChanged="comboBox1_SelectionChanged"
    PreviewMouseDown="comboBox1_PreviewMouseDown"
    AllowDrop="True" DragEnter="comboBox1_DragEnter"
    DragOver="comboBox1_DragEnter" />
```

```
private void comboBox1_DragEnter(object sender, DragEventArgs e)
{
    e.Effects = DragDropEffects.Copy;
    e.Handled = true;
}
```

Результат. Теперь при перетаскивании элемента из выпадающего списка курсор над данным списком является разрешающим (хотя, разумеется, отпускание элемента над выпадающим списком не приведет ни к каким результатам). Если перетаскивание начато из обычного списка listBox1, то над компонентом comboBox1 курсор будет иметь запрещающий вид (это связано с тем, что в обработчике comboBox1_DragEnter для событий DragEnter и DragOver в качестве допустимого режима перетаскивания указан только режим Copy).

16. Флажки и группы флажков: CHECKBOXES

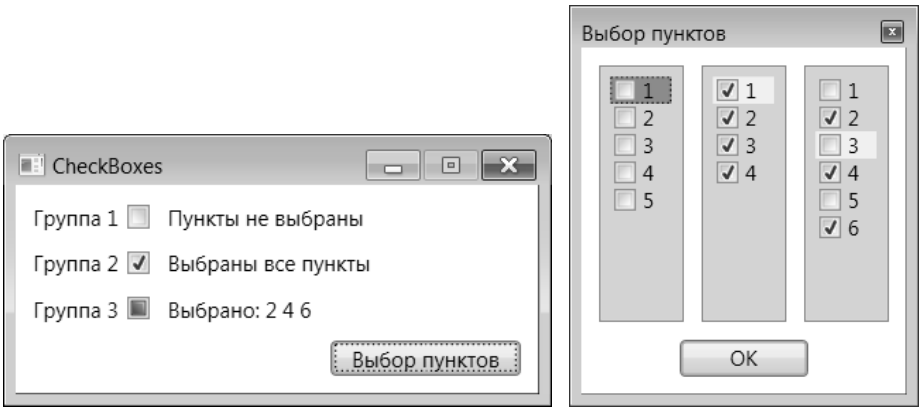


Рис. 52. Окна приложения CHECKBOXES

16.1. Установка флажков и контроль за их состоянием

В этом проекте, как ранее в проекте WINDOWS, будут использоваться два окна – главное MainWindow и диалоговое Window1. После создания заготовки проекта следует сразу добавить к нему новое окно Window1, выполнив действия, описанные в начале раздела, посвященного проекту WINDOWS.

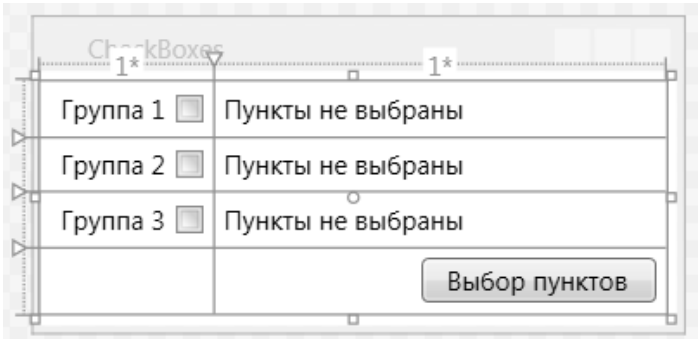


Рис. 53. Макет окна MainWindow приложения CHECKBOXES

```
<Window x:Class="CHECKBOXES.MainWindow"
    …
    Title="CheckBoxes" SizeToContent="WidthAndHeight"
    ResizeMode="CanMinimize" Loaded="Window_Loaded" >
  <Grid x:Name="grid1" Margin="0,5,5,5" >
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition/>
```

```
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <TextBlock x:Name="label1" Grid.Column="1" Margin="5"
        Text="Пункты не выбраны" MinWidth="200" />
    <TextBlock x:Name="label2" Grid.Column="1" Grid.Row="1"
        Margin="5" Text="Пункты не выбраны" MinWidth="200" />
    <TextBlock x:Name="label3" Grid.Column="1" Grid.Row="2"
        Margin="5" Text="Пункты не выбраны" MinWidth="200" />
    <CheckBox x:Name="checkBox1" Content="Группа 1" Margin="5"
        Padding="5,0" FlowDirection="RightToLeft" />
    <CheckBox x:Name="checkBox2" Content="Группа 2" Margin="5"
        Padding="5,0" Grid.Row="1" FlowDirection="RightToLeft" />
    <CheckBox x:Name="checkBox3" Content="Группа 3" Margin="5"
        Padding="5,0" Grid.Row="2" FlowDirection="RightToLeft" />
    <Button x:Name="button1" Grid.Row="3" Grid.ColumnSpan="2"
        Content="Выбор пунктов" Padding="10,0" Margin="5"
        HorizontalAlignment="Right" IsDefault="True"
        Click="button1 Click" />
  </Grid>
</Window>
```

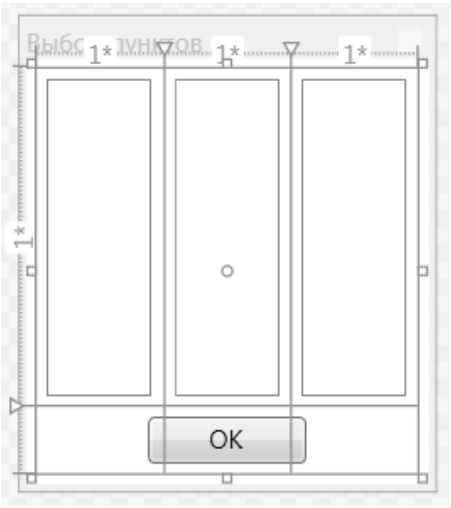


Рис. 54. Макет окна Window1 приложения CHECKBOXES

```
<Window x:Class="CHECKBOXES.Window1"
    …
    Title="Выбор пунктов" ResizeMode="NoResize"
    WindowStartupLocation="CenterScreen" WindowStyle="ToolWindow"
```

```
    SizeToContent="WidthAndHeight"
    IsVisibleChanged="Window IsVisibleChanged"
    Closing="Window Closing" >
  <Grid x:Name="grid1" Margin="5" >
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <ListBox x:Name="listBox1" Padding="5"
        Height="150" Width="50" Margin="5" />
    <ListBox x:Name="listBox2" Padding="5" Grid.Column="1"
        Height="150" Width="50" Margin="5" />
    <ListBox x:Name="listBox3" Padding="5" Grid.Column="2"
        Height="150" Width="50" Margin="5" />
    <Button x:Name="button1" Grid.ColumnSpan="3" Grid.Row="1"
        Content="OK" HorizontalAlignment="Center" Margin="5"
        Width="75" IsDefault="True" IsCancel="True" />
  </Grid>
</Window>
```

В описание класса MainWindow добавьте поле win1

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

а также вспомогательный метод MakeListBoxList:

```
IEnumerable<CheckBox> MakeCheckBoxList(int count)
{
    return Enumerable.Range(1, count).Select(
        e =>
        {
            var res = new CheckBox();
            res.Content = e.ToString();
            return res;
        });
}
```

В конструктор класса MainWindow добавьте операторы:

```
win1.listBox1.ItemsSource = MakeCheckBoxList(5);
win1.listBox2.ItemsSource = MakeCheckBoxList(4);
```

```
win1.listBox3.ItemsSource = MakeCheckBoxList(6);
for (int i = 0; i < 3; i++)
    (win1.grid1.Children[i] as ListBox).SelectedIndex = 0;
button1.Focus();
```

И определите уже указанные в xaml-файле обработчики событий:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    win1.Owner = this;
}
private void button1_Click(object sender, RoutedEventArgs e)
{
    win1.ShowDialog();
}
```

В классе Window1 определите указанные в xaml-файле обработчики событий:

```
private void Window_IsVisibleChanged(object sender,
  DependencyPropertyChangedEventArgs e)
{
    listBox1.Focus();
}
private void Window_Closing(object sender,
  System.ComponentModel.CancelEventArgs e)
{
    e.Cancel = true;
    Hide();
    for (int i = 0; i < 3; i++)
    {
        ListBox lb = grid1.Children[i] as ListBox;
        var a = lb.Items.Cast<CheckBox>()
          .Where(e1 => (bool)e1.IsChecked);
        var tb = (Owner as MainWindow).grid1.Children[i] as
          TextBlock;
        if (a.Count() == lb.Items.Count)
        {
            tb.Text = "Выбраны все пункты";
        }
        else if (a.Count() == 0)
        {
            tb.Text = "Пункты не выбраны";
        }
        else
```

```
        {
            tb.Text = a.Aggregate("Выбрано:",
              (acc, e1) => acc + " " + e1.Content);
        }
    }
}
```

Результат. Если вызвать диалоговое окно «Выбор пунктов» (нажав на кнопку главного окна), установить в нем какие-либо флажки и закрыть его (любым способом, в том числе с помощью клавиш Enter или Esc), то в главном окне отобразится информация о выбранных пунктах в каждой группе. При каждом отображении диалогового окна его активным компонентом (т. е. компонентом, имеющим фокус) является первый список, независимо от того, какой компонент был активным в момент предыдущего закрытия окна.

Для вызова диалогового окна в главном окне достаточно нажать Enter, так как кнопка вызова диалогового окна сделана кнопкой по умолчанию.

Флажки в главном окне пока не используются.

Примечания

1. Для флажков в главном окне мы изменили расположение подписей, указав для свойства FlowDirection значение RightToLeft. Следует отметить, что свойство Padding для флажков (а также для радиокнопок RadioButton) реализовано особым образом: его значение влияет только на расположение *подписи* к флажку; положение маркера с меткой не меняется. Кроме того, необходимо учитывать, что в случае, если свойство FlowDirection равно RightToLeft, первое число в списке Padding определяет *правое* внутреннее поле, в третье (если оно указано) – *левое*.

2. Чтобы диалоговое окно Windiw1 закрывалось по нажатию клавиш Esc и Enter, для кнопки button1 достаточно установить свойства IsDefault и IsCancel равными true; специального обработчика события Click для кнопки в этом случае не требуется. Как и в проекте WINDOWS, программа перехватывает событие закрытия диалогового окна (в обработчике Window_Closing) и заменяет действие по закрытию окна действием по скрытию окна на экране, вызывая метод Hide(). Это предотвращает уничтожение диалогового окна и позволяет использовать его повторно, с сохранением ранее сделанных настроек.

3. О необходимости обеспечить отображение диалогового окна в одном и том же начальном состоянии ранее говорилось в п. 2.6 проекта WINDOWS. Для этого здесь, как и в проекте WINDOWS, достаточно определить для окна Window1 обработчик события IsVisibleChanged.

4. В отличие от библиотеки Windows Forms, где для создания списка флажков предусмотрен особый компонент CheckListBox с большим набором дополнительных свойств и методов и стандартными реакциями

на действия пользователя, в библиотеке WPF списки флажков приходится создавать на базе обычного списка ListBox, помещая в него в качестве элементов компоненты CheckBox. В большинстве изданий, посвященных WPF, подобная возможность преподносится как особое преимущество библиотеки WPF и пример ее гибкости, однако на практике такая гибкость нередко оборачивается неестественным поведением в ответ на стандартные действия пользователя, что требует от разработчика дополнительных усилий по настройке поведения комбинированных компонентов (см. далее в этом пункте описания недочетов и способов их исправления).

Для создания набора флажков, помещаемого в список, используется свойство ItemsSource, позволяющее сразу включить в набор элементов списка некоторую последовательность объектов (ранее свойство ItemsSource и связанные с ним свойства подробно описывались в проекте LISTBOXES, п. 15.1). Поскольку списки в диалоговом окне содержат большое число флажков с единообразными подписями, все флажки создаются и включаются в списки программным способом в конструкторе главного окна (а не определяются с помощью явного указания в файле Window1.xaml). При этом применяется вспомогательный метод MakeCheckBoxList(count), возвращающий готовую последовательность флажков с требуемыми подписями. При генерации последовательности флажков используются запросы Range и Select технологии LINQ. Обратите внимание на тип возвращаемого значения: IEnumerable<CheckBox>; он означает, что возвращается *последовательность* с элементами-флажками.

В конструкторе главного окна для каждого списка с флажками в качестве текущего элемента устанавливается первый элемент (это действие выполняется в цикле).

При анализе флажков в методе Window_Closing также используются запросы LINQ: вначале с помощью запросов Cast<CheckBox> и Where формируется последовательность *установленных* флажков для каждого списка, после чего анализируется ее размер, и, в случае если была установлена *часть* флажков, с помощью запроса Aggregate формируется строка со списком установленных флажков, которая присваивается соответствующему компоненту TextBlock главного окна.

5. Для перебора списков с флажками в цикле (который организуется в конструкторе главного окна и в методе Window_Closing диалогового окна) используется свойство Children компонента grid1, для которого списки с флажками являются дочерними компонентами. Это же свойство, но уже для компонента grid1 главного окна, использовано в методе Window_Closing для перебора компонентов TextBlock; причем в этом случае требуется выполнять явное приведение окна-владельца Owner к его фактическому типу MainWindow. Заметим, что указанного приведе-

ния можно избежать, если вместо свойства Children использовать метод FindName окна:

```
var tb = Owner.FindName("label" + (i + 1)) as TextBlock;
```

Недочет 1. Для некоторых стандартных тем Windows 7 в неактивных списках сложно отличить текущий элемент от остальных. Подобная проблема уже обсуждалась в проекте LISTBOXES, п. 15.2, там же приводится простейший способ ее решения.

Исправление. Добавьте в описание всех трех компонентов ListBox в файле Window1.xaml новый атрибут:

```
Background="LightGray"
```

Результат. Теперь при потере фокуса списком его текущий элемент отображается на более светлом фоне по сравнению с остальными элементами (рис. 55). Текущий элемент списка, имеющего фокус, отображается на светло-синем фоне.

Рис. 55. Вид списков с флажками при использовании серого фона

Следует обратить внимание на то, что в списке, состоящем из флажков, *отсутствует инвертирование цвета заголовков* для текущего элемента: текст выделенного флажка, как и остальных флажков, имеет черный цвет. Для некоторых тем Windows (например, темы «Классическая») это плохо смотрится, так как в этих темах в качестве фона текущего элемента используется темно-синий цвет, на котором черный текст практически не виден. К сожалению, простых средств исправить этот недочет не существует.

Недочет 2. Хотя с помощью клавиатуры можно перемещаться по пунктам-флажкам в списках диалогового окна, выполнять установку или снятие флажков с помощью клавиатуры очень неудобно: необходимо *вначале* нажать клавишу Tab (при этом фокус перейдет с самого списка на его текущий элемент-флажок) и *затем* нажать пробел. Подобный способ действий, помимо всего остального, делает неудобным и переключение между списками: для данного переключения требуется *дважды* нажи-

мать клавишу Tab (поскольку первое нажатие переводит фокус на текущий флажок и только второе обеспечивает переход на следующий список). Впрочем, эти «согласованные» недочеты сравнительно легко исправляются.

Исправление. В классе MainWindow дополните вспомогательный метод MakeCheckBoxList следующим образом:

```
IEnumerable<CheckBox> MakeCheckBoxList(int count)
{
    return Enumerable.Range(1, count).Select(
        e =>
        {
            var res = new CheckBox();
            res.Content = e.ToString();
            res.IsTabStop = false;
            return res;
        });
}
```

Кроме того, в файле Window1.xaml определите обработчик события PreviewKeyDown для списка listBox1:

```
  <ListBox x:Name="listBox1" …
      PreviewKeyDown="listBox1_PreviewKeyDown" />
```

```
private void listBox1_PreviewKeyDown(object sender,
  KeyEventArgs e)
{
    if (e.Key == Key.Space)
    {
        var lb = e.Source as ListBox;
        if (lb == null)
            return;
        var cb = lb.SelectedItem as CheckBox;
        cb.IsChecked = !(bool)cb.IsChecked;
    }
}
```

После этого *переместите* атрибут PreviewKeyDown="listBox1_PreviewKeyDown" из элемента ListBox в его родительский элемент Grid (это действие обеспечит вызов данного обработчика для всех дочерних компонентов таблицы Grid, в частности, для всех списков флажков):

```
<Grid x:Name="grid1" Margin="5"
    PreviewKeyDown="listBox1_PreviewKeyDown" >
  …
  <ListBox x:Name="listBox1" …
```

```
    PreviewKeyDown="listBox1_PreviewKeyDown" />
```

Результат. Теперь для установки/снятия флажка в списке достаточно нажать клавишу пробела, а клавиша Tab (и Shift+Tab) сразу обеспечивает переход от одного списка к другому.

Комментарий

Для исправления недочета мы отключили возможность перевода фокуса на элемент-флажок списка, положив его свойство IsTabStop равным false, и, кроме того, определили обработчик события от клавиатуры, особым образом обрабатывающий событие, связанное с нажатием клавиши пробела. Обратите внимание на то, что для этого мы использовали событие PreviewKeyDown, поскольку событие KeyDown обрабатывается списком особым образом и не может быть связано с обработчиком.

Необходимость приведения свойства IsChecked к типу bool здесь и в других фрагментах программы объясняется тем, что данное свойство имеет тип bool?, и поэтому без явного приведения к bool к нему нельзя применять логические операции (без этого приведения нельзя также указывать это свойство в выражениях, которые должны возвращать логические значения). Причина, по которой используется тип bool?, связана с тем, что таким образом в WPF была реализована возможность работы с флажками, принимающими три состояния (которая будет описана в п. 16.3).

Недочет 3. С исправлением описанных ранее недочетов клавиатурные неприятности не кончаются. Несмотря на то что при открытии диалогового окна фокус получает первый список, сразу начать с ним работать не удается. Если вначале нажать клавишу Tab с целью быстрого перехода ко второму списку, то вместо этого рамка появится около текущего элемента первого списка. Еще более неестественной будет реакция на нажатие клавиш со стрелками. Например, при нажатии клавиши ↓ (с целью перехода на один элемент списка вниз) произойдет переход на кнопку «ОК». И такие проблемы будут возникать при каждом открытии диалогового окна. Правда, они возникают только в начальный момент. В дальнейшем клавиши выполняют ожидаемые действия.

Исправление. Измените обработчик Window_IsVisibleChanged:

```
private void Window_IsVisibleChanged(object sender,
  DependencyPropertyChangedEventArgs e)
{
    (listBox1.SelectedItem as CheckBox).Focus();
}
```

Теперь при *последующих* открытиях диалогового окна не возникает проблем, описанных в недочете 3. Однако при *первом* открытии окна проблемы остаются прежними.

Дополнительное исправление. Добавьте для класса Window1 обработчик события Loaded:

```
<Window x:Class="CHECKBOXES.Window1"
    … Loaded="Window_Loaded" >
```

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    (listBox1.SelectedItem as CheckBox).Focus();
}
```

Результат. Теперь проблемы не возникают и при первом открытии диалогового окна.

Комментарии

1. Необходимость в добавлении особого обработчика для события Loaded обусловлена тем, что вызываемый в обработчике Window_IsVisibleChanged метод Focus будет обеспечивать выполнение требуемых действий только в случае, если диалоговое окно отображается на экране (это связано с достаточно запутанными особенностями механизма работы с фокусом в WPF – см. комментарий к п. 2.6 проекта WINDOWS). К сожалению, в момент выполнения обработчика Window_IsVisibleChanged в ситуации, когда свойство IsVisible становится равным true, окно *еще не отображается на экране* (это странное поведение данного обработчика обсуждалось в том же комментарии). Поэтому нам пришлось определить обработчик события, который возникает в тот момент, когда диалоговое окно отображается на экране в первый раз.

2. При определении обработчика Window_Loaded возникает естественное желание просто вызвать в нем *уже имеющийся* обработчик Window_IsVisibleChanged, передав ему пустые параметры:

```
Window_IsVisibleChanged(null, null);
```

К сожалению, такой способ не сработает, так как второй параметр метода Window_IsVisibleChanged имеет тип DependencyPropertyChangedEventArgs, который является *структурой*, а не классом, и поэтому не может принимать значение null. Впрочем, можно поступить по-другому: определить обработчик Window_Loaded так, как указано выше, и вызвать его в методе Window_IsVisibleChanged:

```
private void Window_IsVisibleChanged(object sender,
  DependencyPropertyChangedEventArgs e)
{
    Window_Loaded(null, null);
}
```

Теперь, после исправления всех отмеченных недочетов, поведение списков, содержащих флажки, при управлении ими с помощью клавиатуры становится достаточно естественным.

Заметим, что управление списком флажков с помощью мыши тоже имеет свои особенности. В частности, щелчок на названии флажка или на его маркере *не приводит к выделению этого элемента списка* (выполняется лишь установка или снятие данного флажка). Для выделения флажка как текущего элемента списка надо щелкнуть мышью *справа от надписи* – в той области, которая уже не связана с текстом, но еще не относится к полям списка (см. рисунок, приведенный перед описанием недочета 2).

В общем, надо признать, что более разумным шагом разработчиков библиотеки WPF было бы создание специализированного компонента, реализующего все возможности списка флажков в полном объеме и не требующего от пользователя библиотеки добавления описанных выше «заплаток», которые к тому же не решают всех проблем (например, проблемы, связанной с отображением текущего элемента черным, а не инверсным цветом).

16.2. «Глобальная» установка флажков и использование флажков, принимающих три состояния

Для флажка checkBox1 в окне MainWindow определите обработчик события Click:

```
<CheckBox x:Name="checkBox1" … Click="checkBox1_Click" />
```

```
private void checkBox1_Click(object sender, RoutedEventArgs e)
{
    var cb = e.Source as CheckBox;
    if (cb == null)
        return;
    var i = Grid.GetRow(cb);
    var tb = grid1.Children[i] as TextBlock;
    tb.Text = cb.IsChecked == true ? "Выбраны все пункты" :
      "Пункты не выбраны";
    var lb = win1.grid1.Children[i] as ListBox;
    foreach (var e1 in lb.Items.Cast<CheckBox>())
        e1.IsChecked = cb.IsChecked;
}
```

После этого *переместите* атрибут Click="checkBox1_Click" из элемента CheckBox в его родительский элемент Grid, дополнив имя атрибута префиксом CheckBox:

```
<Grid x:Name="grid1" … CheckBox.Click="checkBox1_Click" >
  …
  <CheckBox x:Name="checkBox1" … Click="checkBox1_Click" />
```

Кроме того, измените обработчик Window_Closing в классе Window1 следующим образом:

```
private void Window_Closing(object sender,
  System.ComponentModel.CancelEventArgs e)
{
    e.Cancel = true;
    Hide();
    for (int i = 0; i < 3; i++)
    {
        ListBox lb = grid1.Children[i] as ListBox;
        var a = lb.Items.Cast<CheckBox>()
          .Where(e1 => (bool)e1.IsChecked);
        var tb = (Owner as MainWindow).grid1.Children[i] as
          TextBlock;
        var cb = (Owner as MainWindow).grid1.Children[3 + i] as
          CheckBox;
        if (a.Count() == lb.Items.Count)
        {
            tb.Text = "Выбраны все пункты";
            cb.IsChecked = true;
        }
        else if (a.Count() == 0)
        {
            tb.Text = "Пункты не выбраны";
            cb.IsChecked = false;
        }
        else
        {
            tb.Text = a.Aggregate("Выбрано:",
              (acc, e1) => acc + " " + e1.Content);
            cb.IsChecked = null;
        }
    }
}
```

Результат. Установка флажка в главном окне обеспечивает выбор всех пунктов соответствующей группы, а его снятие приводит к отмене всех выбранных пунктов. При открытии диалогового окна «Выбор пунктов» его списки флажков корректируются. Аналогичным образом, при установке или снятии в диалоговом окне всех флажков в некотором списке устанавливается или снимается соответствующий флажок в главном окне. Если в списке флажков установить только часть элементов, то соответствующий флажок в главном окне отображается в особом, *третьем* состоянии, которое в разных темах Windows выглядит по-разному. Например,

в случае классической темы третье состояние выглядит как затененная галочка, а для тем Windows 7 – как квадратный маркер (рис. 56). Сам пользователь не может устанавливать флажки главного окна в это особое состояние.

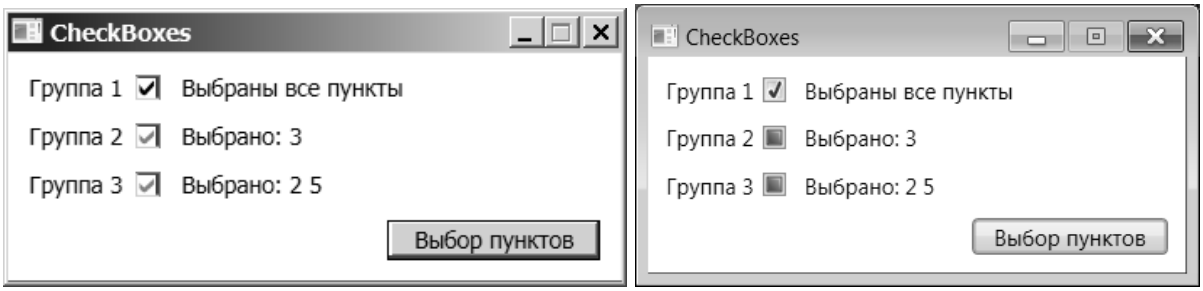


Рис. 56. Вид флажков в третьем состоянии для различных тем Windows 7

Комментарии

1. Для перевода флажка в третье состояние достаточно положить его свойство IsChecked равным null. Если требуется разрешить пользователю выполнять подобное действие, то надо установить свойство флажка IsThreeState равным true. Подчеркнем, что *программную* установку флажка в третье состояние можно выполнять при любом значении свойства IsThreeState.

2. Обратите внимание на способ, с помощью которого в обработчике checkBox1_Click по флажку на главном окне определяются связанные с ним компоненты – метка TextBlock и список флажков в диалоговом окне. Для этого используется *номер строки* компонента Grid, в котором содержится флажок. Для определения номера строки используется *присоединенное свойство* Row компонента Grid (по поводу присоединенных свойств см. проект EVENTS, п. 1.2).

17. Просмотр изображений: IMGVIEW

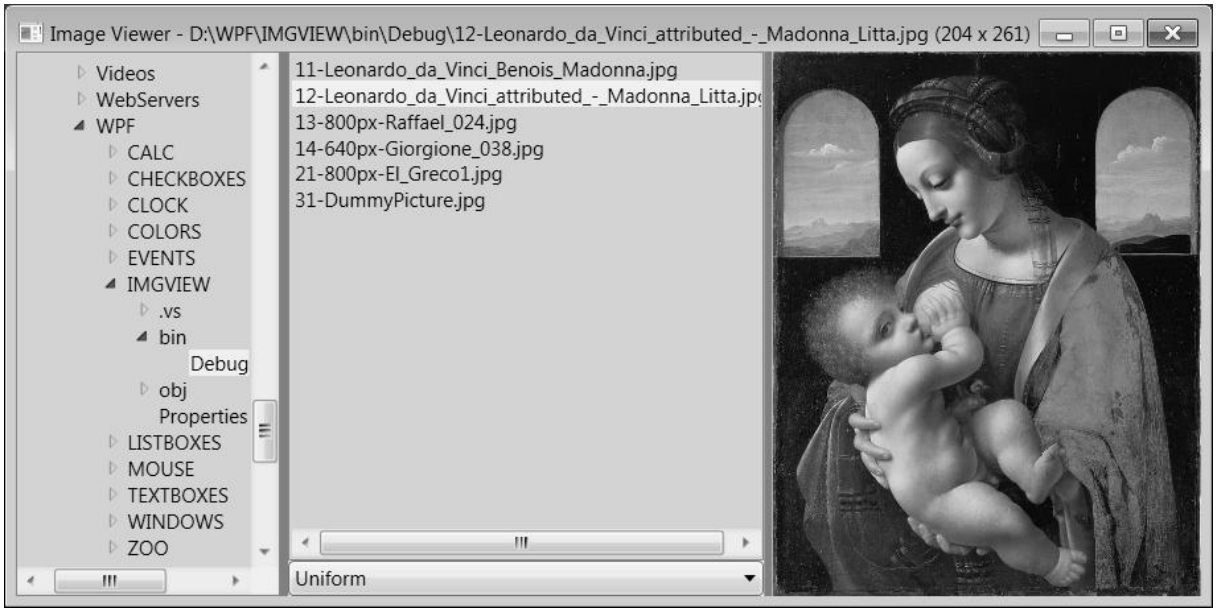


Рис. 57. Окно приложения IMGVIEW

17.1. Иерархический список каталогов

```
<Window x:Class="IMGVIEW.MainWindow"
    …
    Title="Image Viewer" Height="350" Width="700"
    WindowStartupLocation="CenterScreen" >
  <Grid x:Name="grid1" >
      <TreeView x:Name="dirList1" />
  </Grid>
</Window>
```

К списку директив using в начале файле MainWindow.xaml.cs добавьте новую директиву:

```
using System.IO;
```

В классе MainWindow определите два вспомогательных метода:

```
void ExpandItem(TreeViewItem item)
{
    item.Items.Clear();
    if (item.Tag == null)
        foreach (var drive in DriveInfo.GetDrives())
        {
            if (!drive.IsReady)
```

```
                continue;
            TreeViewItem newItem = new TreeViewItem();
            newItem.Tag = drive.RootDirectory;
            newItem.Header = drive.Name;
            if (drive.VolumeLabel != "")
                newItem.Header += " [" + drive.VolumeLabel + "]";
            if (drive.RootDirectory.GetDirectories().Length > 0)
                newItem.Items.Add("*");
            item.Items.Add(newItem);
        }
    else
    {
        try
        {
            foreach (var subDir in (item.Tag as
              DirectoryInfo).GetDirectories())
            {
                try
                {
                    TreeViewItem newItem = new TreeViewItem();
                    newItem.Tag = subDir;
                    newItem.Header = subDir.Name;
                    if (subDir.GetDirectories().Length > 0)
                        newItem.Items.Add("*");
                    item.Items.Add(newItem);
                }
                catch
                { }
            }
        }
        catch
        { }
    }
    item.IsExpanded = true;
}
private void TreeViewItem_Expanded(object sender,
  RoutedEventArgs e)
{
    ExpandItem(e.Source as TreeViewItem);
}
```

В конструктор класса MainWindow добавьте следующие операторы:

```
TreeViewItem item = new TreeViewItem();
dirList1.Tag = item;
item.Tag = null;
item.Header = "Компьютер";
item.Items.Add("*");
dirList1.Items.Add(item);
item.IsSelected = true;
dirList1.Focus();
dirList1.AddHandler(TreeViewItem.ExpandedEvent,
  new RoutedEventHandler(TreeViewItem_Expanded));
```

Результат. При запуске программы в окне отображается начальный пункт «Компьютер» иерархического списка. При его разворачивании появляются все доступные в данный момент диски, а при разворачивании любого диска – все содержащиеся в нем каталоги первого уровня. Если диск имеет метку, то она отображается в квадратных скобках рядом с именем диска.

Если каталог содержит подкаталоги, то его тоже можно развернуть. Рядом с теми элементами списка, которые можно развернуть, отображается специальный *маркер разворачивания*, вид которого зависит от текущей темы Windows: в случае классической темы это квадратик с символом «+», в случае тем Windows 7 – белый треугольник (рис. 58). При разворачивании элемента списка вид маркера изменяется (соответственно на квадратик с символом «–» или на черный наклонный треугольник – см. рисунки, приведенные ниже). Для разворачивания/сворачивания элемента списка достаточно выполнить щелчок мышью на маркере разворачивания или двойной щелчок на имени требуемого элемента.

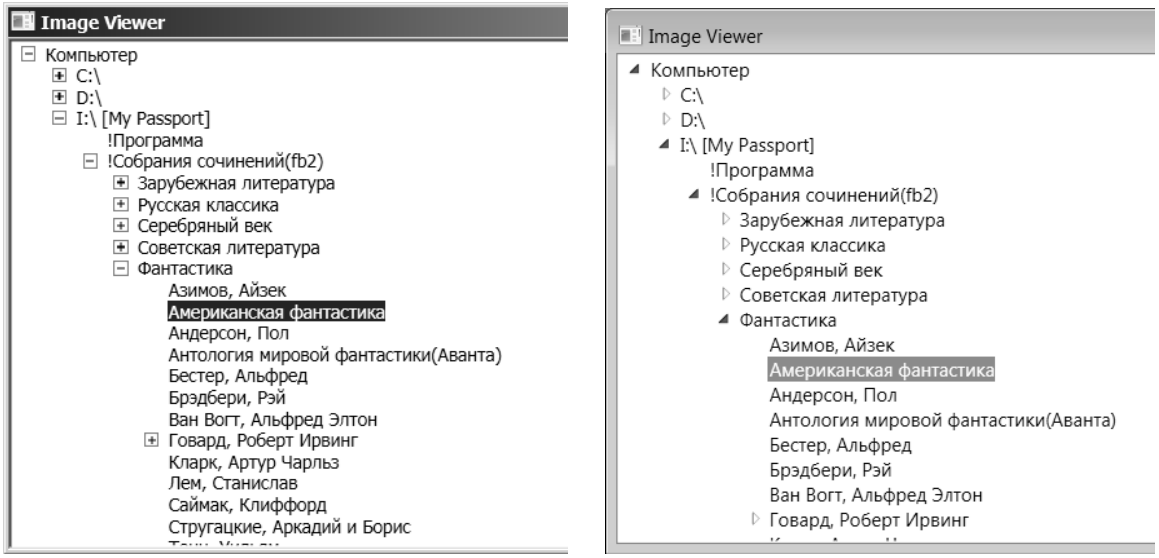


Рис. 58. Вид иерархического списка каталогов для разных тем Windows

Комментарии

1. Главной особенностью компонента TreeView, позволяющего создавать *иерархические списки*, является то, что каждый его элемент (типа TreeViewItem) сам может служить корнем иерархического списка. Основными событиями элементов TreeViewItem являются Expanded, Collapsed, Selected и Unselected. Первые два возникают при разворачивании и, соответственно, сворачивании списка, связанного с данным элементом; это действие можно выполнить программно, изменив свойство IsExpanded. Последние два возникают при установке и, соответственно, снятии выделения для данного элемента; это действие тоже можно выполнить программно, изменив свойство IsSelected. Сам иерархический список TreeView, как и обычные списки ListBox, имеет свойства SelectedItem
и SelectedValue, доступные только для чтения. Свойство SelectedIndex у иерархического списка отсутствует. И у списка TreeView, и у его элементов TreeViewItem имеются свойства Items и ItemsSource, позволяющие настраивать коллекцию их дочерних элементов (первого уровня).

2. В программе используются классы из пространства System.IO, связанного с файловой обработкой, – это класс DriveInfo, позволяющий получить информацию о дисковых устройствах компьютера, и класс DirectoryInfo, позволяющий получить информацию о каталоге (в дальнейшем будут использоваться и другие классы из этого пространства имен).

Из возможностей класса DriveInfo мы использовали статический метод GetDrives, возвращающий коллекцию всех имеющихся дисковых устройств (каждый элемент коллекции тоже имеет тип DriveInfo), и экземплярные свойства IsReady, Name, VolumeLabel и RootDirectory. Свойство IsReady позволяет проверить доступность устройства, Name возвращает его имя, а VolumeLabel – имя метки. Два последних свойства используются для формирования имени элемента, соответствующего корневым каталогам доступных устройств. Свойство RootDirectory имеет тип DirectoryInfo. По объекту DirectoryInfo легко определить все характеристики каталога: его имя (Name), полное имя (FullName), массив типа DirectoryInfo[] с информацией обо всех его подкаталогах (получаемый с помощью метода GetDirectories), массив типа FileInfo[] с информацией обо всех его файлах (получаемый с помощью метода GetFiles) и т. д.

3. При использовании в программе иерархического списка с деревом каталогов не следует сразу строить все дерево, так как количество каталогов на жестких дисках обычно является очень большим. Более эффективным вариантом является построение *части* дерева в тот момент, когда ее нужно отобразить на экране. Дополнительным преимуществом подобного подхода является то, что он позволяет легко *обновить* любой список в со-

ставе дерева в ситуации, когда требуется учесть изменения в наборе его элементов: для этого достаточно свернуть список и еще раз его развернуть.

Для упрощения действий по отображению новых устройств мы создали дерево с единственным корневым элементом «Компьютер», содержащим все *доступные* в данный момент устройства. Если к компьютеру подключается какой-либо новый носитель (например, USB-накопитель) или если некоторое имеющееся устройство становится доступным (например, в результате загрузки CD- или DVD-диска в устройство для их чтения), то для отображения этих новых устройств достаточно свернуть и повторно развернуть корневой элемент «Компьютер».

Обратите внимание на простой способ, позволяющий пометить некоторый элемент как доступный для разворачивания: в коллекцию Items этого элемента добавляется «фиктивный» дочерний элемент «*». Этот дочерний элемент никогда не отображается на экране, так как при разворачивании элемента-родителя он удаляется и выполняется построение списка «настоящих» дочерних элементов. В некоторых книгах предлагается добавлять подобный фиктивный элемент к *любому* создаваемому элементу иерархического списка. В этом случае при попытке развернуть элемент, не имеющий дочерних элементов, просто пропадает значок разворачивания рядом с этим элементом. Мы поступили по-другому: в нашей программе фиктивный дочерний элемент добавляется только в том случае, когда исходный элемент действительно имеет дочерние элементы. Этот способ требует дополнительного расхода времени на анализ дочерних элементов (который мы проводим с помощью метода GetDirectories), но является более удобным для пользователя, так как сразу показывает наличие «настоящих» дочерних элементов. Заметим, что по умолчанию метод GetDirectories возвращает только подкаталоги первого уровня.

Также обратите внимание на использование свойства Tag элемента TreeViewItem для хранения объекта DirectoryInfo, с которым связан данный элемент. С дисками тоже связывается объект DirectoryInfo, а именно: объект, соответствующий корневому каталогу диска. Имеется лишь один элемент списка TreeView, который не связан с каталогом, – это корневой элемент «Компьютер», у которого свойство Tag полагается равным null.

4. В методе ExpandItem используются два оператора try–catch, позволяющие обрабатывать ошибки, связанные с доступом к отдельным каталогам. Внешний блок try предназначен для того, чтобы корректно обрабатывать ситуацию, при которой выполняется попытка доступа к каталогу, который стал недоступным (либо по причине его удаления, либо по причине отсоединения содержащего этот каталог устройства). В такой ситуации разворачивание отменяется, а у этого элемента пропадает маркер раз-

ворачивания. Внутренний блок try (размещенный в цикле foreach) предназначен для того, чтобы корректно обрабатывать ситуацию, при которой какие-либо дочерние каталоги блокируют доступ к их содержимому (в этом случае вызов метода GetDirectories для этих каталогов приведет к исключению). Подобные каталоги просто не включаются в список дочерних каталогов.

Недочет 1. Хотя для компонента TreeView предусмотрена возможность перемещения по элементам с помощью клавиш со стрелками, эта возможность является практически бесполезной, если одновременно не поддерживаются клавиатурные действия по разворачиванию/сворачиванию элементов (а они по умолчанию не поддерживаются).

Исправление. Определите обработчик события PreviewKeyDown для компонента dirList1:

```
<TreeView x:Name="dirList1"
    PreviewKeyDown="dirList1 PreviewKeyDown" />
```

```
private void dirList1_PreviewKeyDown(object sender,
  KeyEventArgs e)
{
    if (e.Key == Key.Space || e.Key == Key.Return)
    {
        var tv = e.Source as TreeViewItem;
        tv.IsExpanded = !tv.IsExpanded;
    }
}
```

Результат. Теперь в программе обеспечена полная поддержка перемещения по списку каталогов с помощью клавиатуры, причем для сворачивания или разворачивания текущего элемента можно использовать либо клавишу Enter, либо клавишу пробела.

Комментарий

Хотя событие определено для иерархического списка dirList1, фактически оно возникает *в одном из его дочерних элементов* типа TreeViewItem, о чем свидетельствует значение свойства e.Source, равное текущему элементу компонента dirList1. Заметим, что вместо использования этого свойства можно было обратиться непосредственно к текущему элементу списка:

```
var tv = dirList1.SelectedItem as TreeViewItem;
```

Недочет 2. При каждом запуске программы пользователю приходится выполнять одни и те же действия для перехода в нужный каталог.

Исправление. Добавьте в класс MainWindow новый метод:

```
TreeViewItem InitialExpanding(string fullPath)
{
```

```
    if (!Directory.Exists(fullPath))
        return null;
    var paths = fullPath.Split('\\');
    paths[0] += "\\";
    TreeViewItem rootItem = dirList1.Items[0] as TreeViewItem;
    ExpandItem(rootItem);
    TreeViewItem item = rootItem;
    foreach (var e in paths)
    {
        item = item.Items.Cast<TreeViewItem>()
          .FirstOrDefault(e1 => (e1.Tag as
            DirectoryInfo).Name.ToUpper() == e.ToUpper());
        if (item == null)
            return null;
        ExpandItem(item);
    }
    return item;
}
```

И *после* имеющегося в конструкторе класса MainWindow оператора

```
item.IsSelected = true;
```

добавьте новые операторы:

```
item = InitialExpanding(Directory.GetCurrentDirectory());
if (item != null)
    item.IsSelected = true;
```

Результат. Теперь при запуске программы выполняется автоматический переход к *текущему каталогу* (по умолчанию это подкаталог Debug, входящий в каталог, содержащий разрабатываемый проект, рис. 59).

Рис. 59. Окно приложения IMGVIEW с выделенным текущим каталогом

Комментарии

1. В дальнейшем (см. п. 17.6) при запуске программы будет выполняться переход к тому каталогу, который был выбран перед ее предыдущим закрытием. По этой причине в методе InitialExpanding предусматривается ситуация, когда требуемый каталог отсутствует или части пути не удается найти в существующем дереве каталогов. Во всех таких особых ситуациях метод возвращает null.

2. Для разбиения полного пути fullPath к стартовому каталогу на части использован вариант метода Split класса string с символьным параметром, определяющим разделитель (в нашем случае это символ «\»). Поскольку в имени корневого каталога необходима косая черта, она добавляется к первому элементу сформированного массива.

3. В методе InitialExpanding вначале разворачивается корневой элемент списка dirList1 (это единственный элемент его коллекции Items), а затем выполняется перебор всех каталогов, входящих в полный путь к стартовому каталогу. Очередной подкаталог, входящий в путь стартового каталога, ищется в соответствующей коллекции элементов иерархического списка. После его нахождения он разворачивается, в результате чего в списке появляется новая коллекция элементов, в которой организуется поиск следующей части пути к стартовому каталогу. Для поиска используется запрос FirstOrDefault, возвращающий первый элемент коллекции, который удовлетворяет указанному условию, или значение null, если требуемые элементы в коллекции отсутствуют. Поскольку имена каталогов нечувствительны к регистру, перед их сравнением они преобразуются в верхний регистр методом ToUpper.

4. Обратите внимание на то, что в методе InitialExpanding элементы иерархического списка разворачиваются, но *не выделяются*. Это связано с тем, что событие выделения элемента списка каталогов при последующей модификации программы будет связано с выполнением дополнительных действий, которые могут быть достаточно длительными. Поэтому выделение нового элемента выполняется уже после выхода из метода InitialExpanding (если вызов метода был успешным).

17.2. Список файлов. Компоненты-разделители

```
<Window x:Class="IMGVIEW.MainWindow"
    … >
  <Grid x:Name="grid1">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="160"/>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition />
```

```
    </Grid.ColumnDefinitions>
    <TreeView x:Name="dirList1"
        PreviewKeyDown="dirList1_PreviewKeyDown"
        SelectedItemChanged="dirList1_SelectedItemChanged" />
    <GridSplitter Grid.Column="1" MinWidth ="5"
        HorizontalAlignment="Center" />
    <ListBox x:Name="fileList1" Grid.Column="2" />
  </Grid>
</Window>
```

К списку директив using в начале файла MainWindow.xaml.cs добавьте новую директиву:

```
using IOPath = System.IO.Path;
```

В описание класса MainWindow добавьте новое поле:

```
string[] imageExts = { ".bmp", ".jpeg", ".jpg", ".png", ".gif",
  ".ico", ".wmf", ".emf" };
```

Определите добавленный в xaml-файл обработчик события SelectedItemChanged для компонента dirList1:

```
private void dirList1_SelectedItemChanged(object sender,
  RoutedPropertyChangedEventArgs<object> e)
{
    var dirInfo = (dirList1.SelectedItem as TreeViewItem).Tag as
      DirectoryInfo;
    if (dirInfo == null)
        fileList1.ItemsSource = null;
    else
    {
        try
        {
            var src = dirInfo.GetFiles().Select(e1 => e1.Name)
              .Where(e1 => imageExts.Contains(IOPath
                .GetExtension(e1).ToLower()));
            fileList1.ItemsSource = src;
            if (src.Count() > 0)
                fileList1.SelectedIndex = 0;
        }
        catch
        {
            fileList1.ItemsSource = null;
        }
    }
}
```

В подкаталог Debug, связанный с проектом IMGVIEW, скопируйте несколько графических файлов.

Результат. При выборе какого-либо каталога в иерархическом списке каталогов dirList1 в списке файлов fileList1 отображаются имена *графических файлов* (с расширениями bmp, jpeg, jpg, png, gif, ico, wmf, emf), находящихся в выбранном каталоге, причем имя первого файла из списка становится текущим (рис. 60).

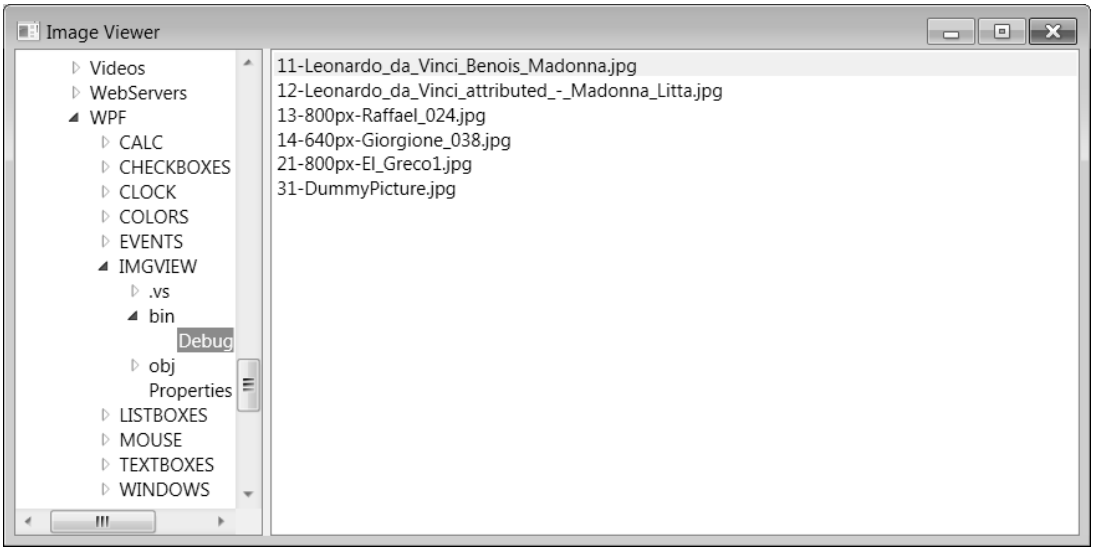


Рис. 60. Окно приложения IMGVIEW со списком файлов

Между списками каталогов и файлов содержится специальный компонент-разделитель GridSplitter, зацепив мышью за который можно изменить ширину одного списка за счет ширины другого (рис. 61).

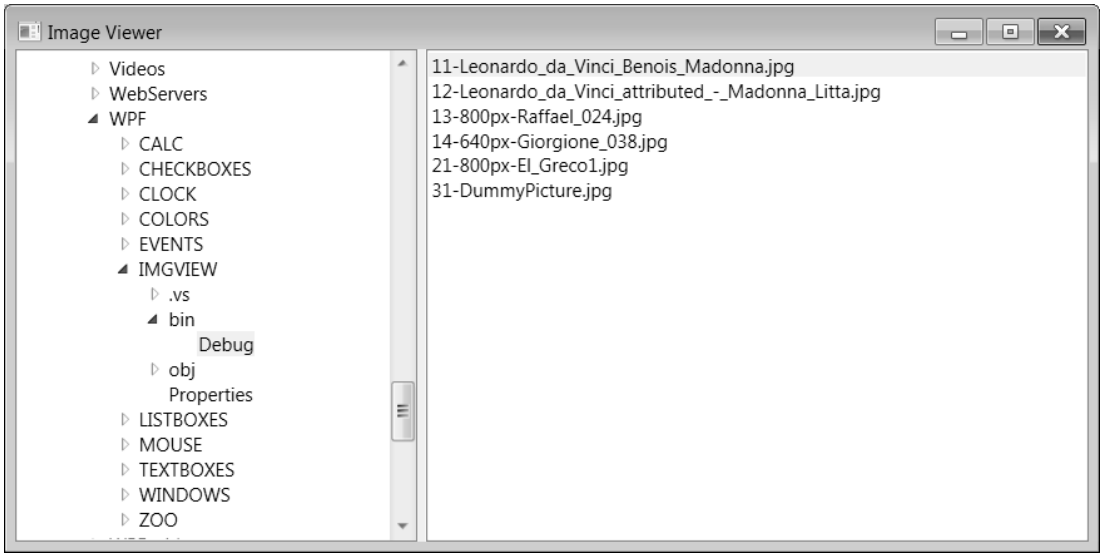


Рис. 61. Окно приложения IMGVIEW после перемещения разделителя

Комментарии

1. В методе dirList1_SelectedItemChanged предусмотрены две особые ситуации: когда элемент списка dirList1 не связан ни с каким каталогом

(таким элементом является корневой элемент списка «Компьютер») и когда при попытке построения списка файлов возбуждается исключение. В каждой из этих ситуаций список файлов делается пустым (пустым он будет и в случае, если в каталоге отсутствуют графические файлы).

2. Для получения расширения из имени файла используется соответствующий метод класса Path из пространства имен System.IO. Хотя это пространство уже указано в списке директив using, попытка использования имени Path приведет к сообщению компилятора о неоднозначности данного имени (так как класс с таким именем имеется также и в пространстве имен System.Windows.Shapes, автоматически подключаемом к любому WPF-приложению). Мы исправили данную ошибку, определив *псевдоним* IOPath для класса System.IO.Path с помощью еще одной директивы using. Разумеется, ее можно было исправить и другими способами, например, удалив директиву using System.Windows.Shapes (если в программе не предполагается использовать классы из этого пространства имен) или указав полное имя System.IO.Path при обращении к классу Path в методе dirList1_SelectedItemChanged.

3. В некоторых распространенных программах, обеспечивающих аналогичную функциональность и включающих два списка, один из которых является иерархическим (например, в Проводнике), нажатие клавиши Enter или пробела в иерархическом списке приводит лишь к обновлению набора элементов из другого списка; при этом развернуть элемент иерархического списка с помощью клавиатуры невозможно.

Недочеты. В стандартных темах Window 7 в случае, если список не имеет фокуса, сложно определить, какой из его элементов является текущим (поскольку фон текущего элемента почти не отличается от фона остальных элементов). Кроме того, после перетаскивания разделителя GridSplitter сам разделитель остается текущим компонентом (имеющим фокус), что является неудобным. При использовании клавиши Tab для переключения между компонентами фокус со списка каталогов вначале переходит на разделитель, а уже после повторного нажатия Tab – на список файлов, что также является неудобным. Наконец, разделитель в окне никак не выделяется, и догадаться о его наличии можно только после наведения на него курсора мыши (курсор при этом изменит вид на горизонтальную двунаправленную стрелку).

Исправления. Все отмеченные недочеты исправляются с помощью дополнительных настроек в xaml-файле:

```
<TreeView x:Name="dirList1"
    PreviewKeyDown="dirList1_PreviewKeyDown"
    SelectedItemChanged="dirList1_SelectedItemChanged"
    Background="LightGray" />
<GridSplitter Grid.Column="1" MinWidth ="5"
```

```
    HorizontalAlignment="Center"
    Background="Gray" Focusable="False" />
<ListBox x:Name="fileList1" Grid.Column="2"
    Background="LightGray" />
```

Результат. Теперь оба списка отображаются на сером фоне, на котором выделяется текущий компонент неактивного списка, поскольку он имеет более светлый фон (рис. 62). Разделитель GridSplitter теперь имеет темно-серый фон и, кроме того, не может получать фокус, поэтому клавиша Tab обеспечивает немедленное переключение между списком каталогов и списком файлов.

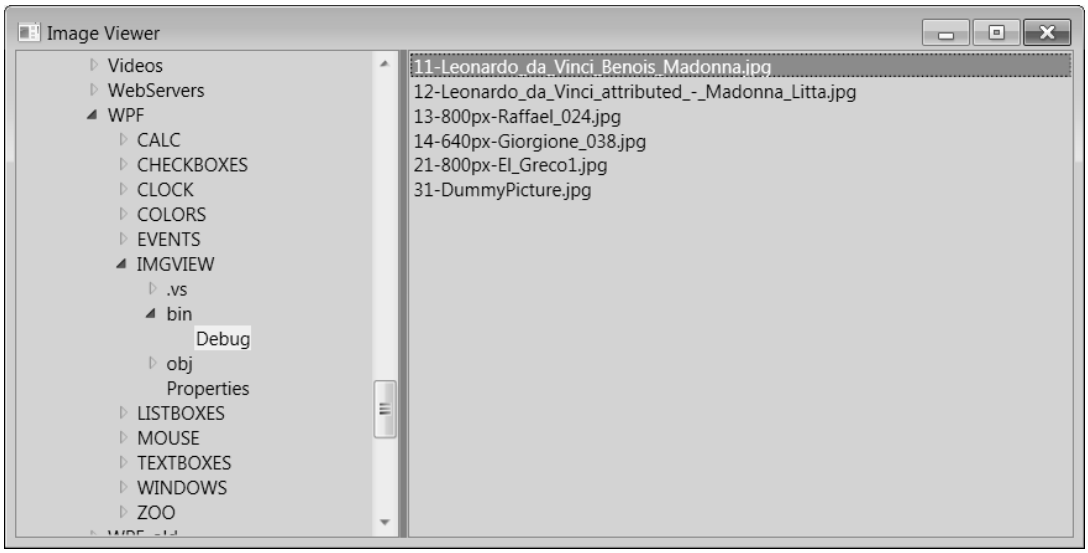


Рис. 62. Окно приложения IMGVIEW со списками каталогов и файлов на сером фоне

Комментарии

1. Наиболее простым и естественным способом настройки компонента-разделителя является помещение его в отдельный столбец компонента Grid с автоматически настраиваемой шириной (если в программе требуется использовать *горизонтальный* разделитель, то его надо поместить в отдельную *строку* с автоматически настраиваемой высотой и вместо свойства HorizontalAlignment задать равным значению Center свойство VerticalAlignment). Путем настройки этих и других свойств компонента GridSplitter можно обеспечить иные варианты его поведения, но они являются менее естественными для пользователя и поэтому применяются редко.

2. Положив свойство Focusable равным false, мы отключили для компонента GridSplitter возможность получения им фокуса. Это свойство имеется и у других визуальных компонентов, причем у некоторых оно по умолчанию равно true, а у остальных – false (изменять значение по умолчанию можно в любом из этих случаев). Родственным свойством является свойство IsTabStop, определяющее, можно ли активизировать данный

компонент с помощью клавиши Tab. Если свойство Focusable равно false, то активизировать компонент нельзя никаким способом, поэтому в этом случае значение свойства IsTabStop игнорируется. Ранее эти свойства обсуждались в проекте TEXTBOXES.

17.3. Компоненты для просмотра изображений и прокрутки содержимого

```
<Window x:Class="IMGVIEW.MainWindow"
    … >
  <Grid x:Name="grid1">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="160"/>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition Width="160"/>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <TreeView x:Name="dirList1" … />
    <GridSplitter Grid.Column="1" … />
    <ListBox x:Name="fileList1" Grid.Column="2"
        Background="LightGray"
        SelectionChanged="fileList1_SelectionChanged" />
    <GridSplitter Grid.Column="3" MinWidth ="5"
        HorizontalAlignment="Center" Background="Gray"
        Focusable="False" />
    <ScrollViewer x:Name="scrollViewer1" Grid.Column="4"
        VerticalScrollBarVisibility="Auto"
        HorizontalScrollBarVisibility="Auto" >
      <Image x:Name="image1" Stretch="None" />
    </ScrollViewer>
  </Grid>
</Window>
```

Определите указанный в xaml-файле обработчик события SelectionChanged для компонента fileList1:

```
private void fileList1_SelectionChanged(object sender,
     SelectionChangedEventArgs e)
{
    string name = (string)fileList1.SelectedValue;
    if (name != null)
    {
        name = ((dirList1.SelectedItem as TreeViewItem).Tag as
```

```
          DirectoryInfo).FullName + "\\" + name;
        Title = "Image Viewer - " + name;
        Mouse.OverrideCursor = Cursors.Wait;
        try
        {
            image1.Source = new BitmapImage(new Uri(name));
            Title += " (" + (int)image1.Source.Width + " x " +
              (int)image1.Source.Height + ")";
        }
        catch
        {
            Title += " (WRONG FORMAT)";
            image1.Source = null;
        }
        Mouse.OverrideCursor = null;
    }
    else
    {
        Title = "Image Viewer";
        image1.Source = null;
    }
}
```

Результат. В правой части окна загружается изображение из текущего графического файла (рис. 63). При этом полное имя файла и размер изображения (в аппаратно-независимых единицах без дробных частей) выводятся в заголовке окна. Если размер изображения больше размера области просмотра, то в области просмотра отображаются полосы прокрутки.

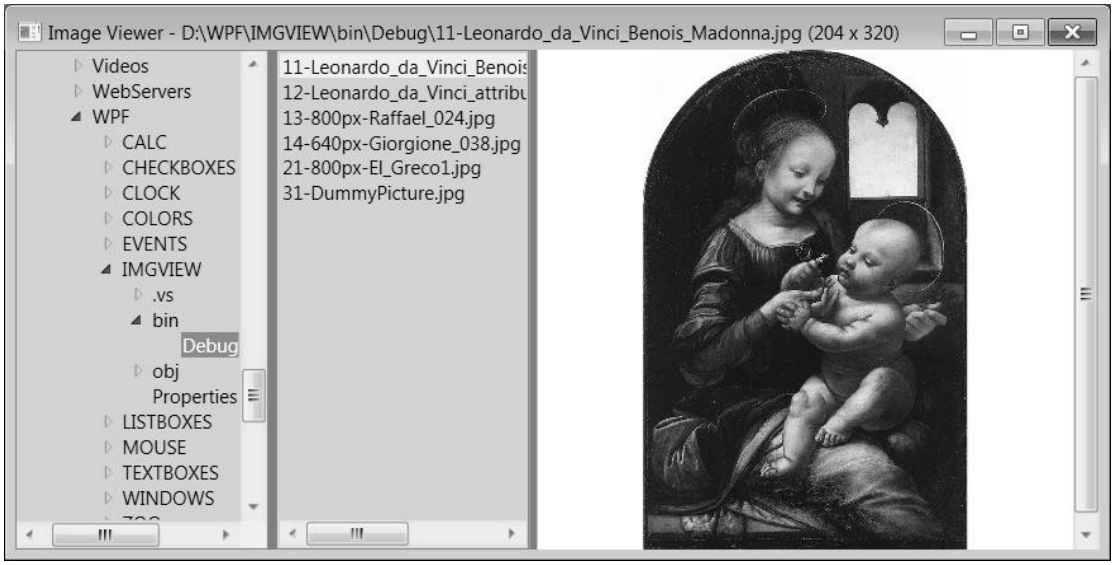


Рис. 63. Окно приложения IMGVIEW с загруженным изображением

Если текущий каталог не содержит графических файлов или если текущий графический файл имеет неверный формат, то область просмотра остается пустой, причем в случае неверного формата соответствующая информация выводится в заголовке окна (для проверки этой возможности в каталог Debug был добавлен пустой файл DummyPicture.jpg, рис. 64).

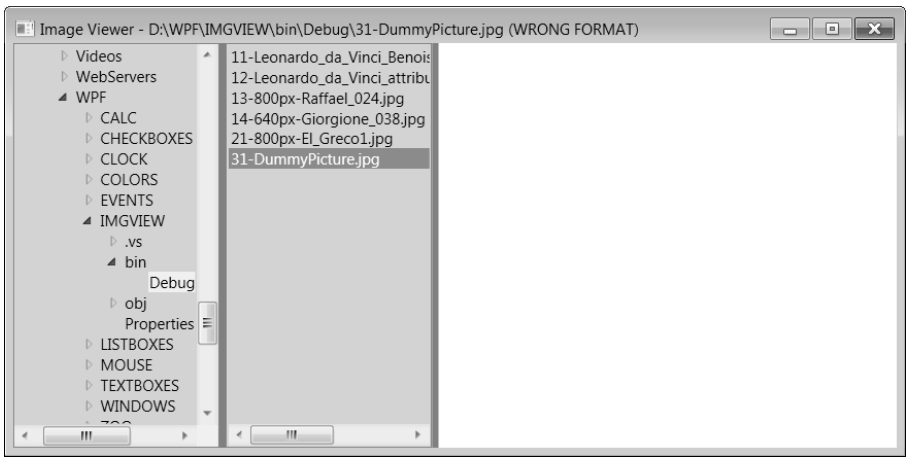


Рис. 64. Вид окна приложения IMGVIEW при выборе ошибочного графического файла

При загрузке изображения вид курсора для всего приложения изменяется на курсор ожидания, после завершения загрузки вид курсора восстанавливается (работа с курсорами подробно обсуждалась ранее в проекте CURSORS).

При перемещении левого разделителя изменяется ширина списка каталогов и области просмотра (ширина списка файлов остается неизменной, однако изменяется его положение). При перемещении правого разделителя изменяется ширина списка файлов и области просмотра (ширина списка каталогов остается неизменной). При изменении ширины окна изменяется только ширина области просмотра изображения.

Комментарии

1. Для отображения изображений предназначен компонент Image; его свойство Source типа ImageSource содержит загруженное изображение, которое можно прочесть из файла name с помощью конструктора класса BitmapImage с параметром типа Uri, который, в свою очередь, можно создать на основе строки name. Обратившись к свойствам Source.Width и Source.Height, можно определить реальные размеры изображения. Свойство Stretch компонента Image позволяет задать режим просмотра изображения (в нашем случае используется режим без масштабирования None; другие режимы будут описаны в следующем пункте).

Следует иметь в виду, что использованный в нашей программе способ создания объекта класса BitmapImage приводит к блокировке графического файла, связанного с созданным объектом. Если подобная блокировка нежелательна (например, если в программе предполагается *изменять* данный графический файл), то можно связать объект BitmapImage с копи-

ей изображения, загруженной в память, используя свойство StreamSource класса BitmapImage (значение свойства StreamSource необходимо устанавливать между вызовами методов BeginInit и EndInit):

```
var im = new BitmapImage();
im.BeginInit();
im.StreamSource = new MemoryStream(File.ReadAllBytes(name));
im.EndInit();
image1.Source = im;
```

2. Прокрутку компонентов, не умещающихся в выделенной для них области окна, обеспечивает компонент ScrollViewer. Мы установили для него режим, при котором полосы прокрутки (как вертикальная, так и горизонтальная) отображаются только в случае необходимости.

3. Правила, по которым изменяются размеры столбцов компонента Grid при перетаскивании его разделителей, определяются настройками столбцов. В нашем случае столбцы для каталогов и файлов имеют *фиксированную* начальную ширину (равную 160), а для последнего столбца ширина не определена, что по умолчанию означает ширину 1* (т. е. этот столбец будет занимать *всю оставшуюся область компонента* Grid). Если изменение положения разделителя (или изменение размера всего окна) может приводить к изменению ширины либо столбца с явно заданной шириной, либо с шириной, заданной с помощью «звездочек», то меняется только ширина столбца со «звездочками».

Предположим, что мы задали бы настройки столбцов, используя только «звездочки»:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="1*"/>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="1*"/>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="2*"/>
</Grid.ColumnDefinitions>
```

Тогда изменение положения любого разделителя влияло бы только на ширину соседних с ним столбцов, а изменение ширины окна приводило бы к пропорциональному изменению ширины всех столбцов. Заметим, что приведенные значения размеров (1*, 1*, 2*) означают, что в начальный момент список каталогов и список файлов будут иметь одинаковую ширину, а ширина области просмотра изображения будет в два раза больше ширины любого из списков.

Если бы мы задали настройки столбцов следующим образом:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="160"/>
    <ColumnDefinition Width="Auto"/>
```

```
        <ColumnDefinition Width="1*"/>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="2*"/>
    </Grid.ColumnDefinitions>
```

то перемещение *правого* разделителя повлияло бы только на ширину соседних с ним столбцов, а перемещение *левого* разделителя изменило бы ширину всех трех столбцов, причем ширина второго и третьего столбца изменялась бы с сохранением указанной пропорции (1 к 2). Изменение ширины окна влияло бы только на столбцы со «звездочками», причем также пропорциональным образом.

Разумеется, все сказанное выше относится не только к столбцам, но и к строкам компонента Grid.

Недочет. Прокрутку изображения можно выполнять с помощью клавиатуры, однако только после щелчка мышью на изображении. Таким образом, хотя компонент ScrollViewer может принимать фокус, перейти на него клавишей Tab не удается.

Исправление. Добавьте в метод fileList1_SelectionChanged следующие операторы:

```
scrollViewer1.Focusable = image1.Source != null;
scrollViewer1.IsTabStop = image1.Source != null;
```

Результат. Теперь на изображение можно перейти клавишей Tab, после чего использовать клавиши со стрелками (а также Home, End, PgUp и PgDn) для прокрутки большого изображения. Таким образом, нажатие Tab приводит к циклическому перебору списка каталогов, списка файлов и области изображения. В случае пустого списка файлов или пустой области изображения они не могут быть сделаны активными.

17.4. Масштабирование изображений

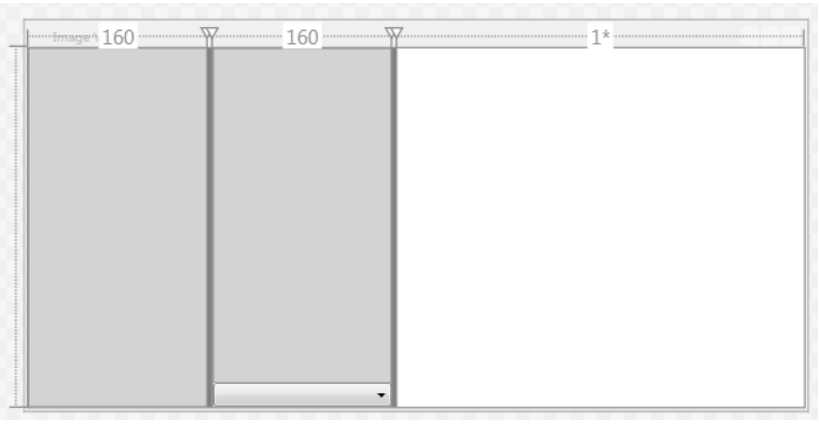


Рис. 65. Макет окна приложения IMGVIEW с добавленным выпадающим списком

```
<Window x:Class="IMGVIEW.MainWindow"
    … >
  <Grid x:Name="grid1">
```

```
    …
    <TreeView x:Name="dirList1" … />
    <GridSplitter Grid.Column="1" … />
    <DockPanel Grid.Column="2">
      <ComboBox x:Name="comboBox1" DockPanel.Dock="Bottom"
          SelectionChanged="comboBox1_SelectionChanged" />
      <ListBox x:Name="fileList1" Background="LightGray"
          Grid.Column="2"
          SelectionChanged="fileList1_SelectionChanged" />
    </DockPanel>
    <GridSplitter Grid.Column="3" … />
    <ScrollViewer x:Name="scrollViewer1" Grid.Column="4"
        VerticalScrollBarVisibility="Auto"
        HorizontalScrollBarVisibility="Auto" >
      <Image x:Name="image1" Stretch="None" />
    </ScrollViewer>
  </Grid>
</Window>
```

В конструктор класса MainWindow добавьте операторы:

```
comboBox1.ItemsSource = Enum.GetValues(typeof(Stretch));
comboBox1.SelectedIndex = 0;
```

Определите добавленный в xaml-файл обработчик события SelectionChanged для компонента comboBox1:

```
private void comboBox1_SelectionChanged(object sender,
  SelectionChangedEventArgs e)
{
    image1.Stretch = (Stretch)comboBox1.SelectedValue;
    if (image1.Stretch == Stretch.None)
    {
        scrollViewer1.HorizontalScrollBarVisibility =
          scrollViewer1.VerticalScrollBarVisibility =
          ScrollBarVisibility.Auto;
        image1.Width = image1.Height = double.NaN;
    }
    else
    {
        scrollViewer1.HorizontalScrollBarVisibility =
          scrollViewer1.VerticalScrollBarVisibility =
          ScrollBarVisibility.Disabled;
        image1.Width = scrollViewer1.ActualWidth;
        image1.Height = scrollViewer1.ActualHeight;
```

```
    }
}
```

Результат. С помощью выпадающего списка comboBox1 можно настраивать режим *масштабирования* изображения (определяемый свойством Stretch). Выпадающий список содержит четыре значения – None, Fill, Uniform и UniformToFill, которые соответствуют вариантам значений свойства Stretch компонента Image. При значении None изображение не масштабируется, при Fill оно масштабируется по размеру компонента Image без сохранения пропорций, при Uniform – с сохранением пропорций, а при UniformToFill – с сохранением пропорций, но при полном заполнении компонента (при этом часть изображения – нижняя или правая – будет отсекаться). На рис. 66 приведены варианты представления одного и того же изображения в разных режимах.

Комментарии

1. Для правильной работы режимов с масштабированием необходимо явно настраивать размеры компонента Image по текущим размерам компонента ScrollView. Наоборот, при отключении масштабирования необходимо отключить режим фиксированных размеров Image, положив свойства Width и Height равными значению NaN; при этом фактические размеры компонента Image будут определяться размерами загруженного изображения. Все эти действия предусмотрены в обработчике comboBox1_SelectionChanged.

2. Благодаря использованию свойства ItemsSource класса ComboBox и метода GetValues класса Enum, нам удалось определить список значений для выпадающего списка с помощью единственного оператора в конструкторе класса MainWindow. Кроме того, это позволило использовать свойство SelectedValue для получения текущего значения comboBox1 в обработчике comboBox1_SelectionChanged.

3. С помощью группирующего компонента DockPanel мы объединили во втором столбце окна список файлов и выпадающий список режимов масштабирования. При этом для выпадающего списка задано присоединенное свойство DockPanel.Dock, равное Bottom, которое обеспечило стыковку компонента comboBox1 к нижней части панели DockPanel (с одновременным растяжением этого компонента по ширине панели). Для второго дочернего компонента панели DockPanel (списка fileList1) свойство DockPanel.Dock не задано, так как по умолчанию последний дочерний компонент панели DockPanel занимает всю оставшуюся свободной часть ее клиентской области (ранее мы использовали компонент DockPanel в проектах TEXTEDIT и COLORS).

Рис. 66. Варианты режимов представления изображения

Недочет 1. При изменении размеров области просмотра размер отмасштабированного изображения не изменяется.

Исправление. Для реакции на изменение размера области просмотра необходимо определить обработчик события SizeChanged компонента scrollViewer1:

```
<ScrollViewer x:Name="scrollViewer1" Grid.Column="4"
    SizeChanged="scrollViewer1_SizeChanged" >
```

```
private void scrollViewer1_SizeChanged(object sender,
  SizeChangedEventArgs e)
{
    if (image1.Stretch != Stretch.None)
    {
        image1.Width = scrollViewer1.ActualWidth;
        image1.Height = scrollViewer1.ActualHeight;
    }
}
```

После определения данного обработчика можно *заменить* операторы настройки размеров компонента image1 в конце метода comboBox1_SelectionChanged на вызов этого обработчика:

```
image1.Width = scrollViewer1.ActualWidth;
image1.Height = scrollViewer1.ActualHeight;
scrollViewer1_SizeChanged(null, null);
```

Результат. Теперь при любом изменении размеров области просмотра (либо за счет изменения размеров окна, либо в результате перетаскивания одного из разделителей) отмасштабированное изображение подстраивается под новые размеры области просмотра.

Недочет 2. Порядок перехода по нажатию клавиши Tab является не вполне естественным: после списка каталогов происходит переход на выпадающий список comboBox1 (расположенный в нижней части второго столбца), затем – на список файлов fileList1 (расположенный в верхней части этого же столбца) и затем – на область просмотра изображения. Это обусловлено тем, что в xaml-файле при перечислении дочерних компонентов панели DockPanel вначале указан компонент comboBox1, а затем – компонент fileList1.

Заметим, что изменить порядок следования компонентов в xaml-файле нельзя, так как дочерний компонент, занимающий всю оставшуюся область панели DockPanel, должен быть указан последним.

Можно было бы явно задать значения свойства TabIndex для компонентов fileList1 (TabIndex="1") и comboBox1 (TabIndex="2"). При этом потребовалось бы определить это свойство и для компонента scrollViewer1 (TabIndex="3"); если этого не сделать, то нажатие Tab будет переводить

фокус со списка каталогов сразу на область просмотра и только потом на компоненты второго столбца.

Однако еще более удобным для пользователя вариантом было бы *исключение* выпадающего списка из последовательности компонентов, которые обходятся по нажатию клавиши Tab, особенно если в программе предусмотрен быстрый способ изменения варианта масштабирования с помощью клавиатуры.

Исправление. Добавьте в xaml-файл два новых атрибута

```
<ComboBox x:Name="comboBox1" DockPanel.Dock="Bottom"
    SelectionChanged="comboBox1_SelectionChanged"
    IsTabStop="False" />
…
<ScrollViewer x:Name="scrollViewer1" Grid.Column="4"
    SizeChanged="scrollViewer1_SizeChanged"
    KeyDown="scrollViewer1_KeyDown" >
```

и определите указанный в xaml-файле обработчик события KeyDown:

```
private void scrollViewer1_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Space || e.Key == Key.Return)
        comboBox1.SelectedIndex = (comboBox1.SelectedIndex + 1) %
          comboBox1.Items.Count;
    else if (e.Key == Key.Back)
        comboBox1.SelectedIndex = (comboBox1.SelectedIndex +
          comboBox1.Items.Count - 1) % comboBox1.Items.Count;
}
```

Результат. Теперь при нажатии клавиши Tab перебираются только списки каталогов, файлов и область просмотра. Кроме того, при активной области просмотра нажатие клавиши Enter или клавиши пробела обеспечивает перебор вариантов масштабирования в прямом порядке, а нажатие клавиши Backspace – в обратном.

Комментарий

Можно также связать с каждым вариантом масштабирования клавишу, нажатие на которую устанавливает этот вариант (например, N –None, F – Fill, U – Uniform, T – UniformToFill).

17.5. Сохранение в реестре Windows информации о состоянии программы

```
<Window x:Class="IMGVIEW.MainWindow"
    …
    Closed="Window_Closed" >
```

В список директив using файла MainWindow.xaml.cs добавьте директиву

```
using Microsoft.Win32;
```

В описание класса MainWindow добавьте поле

```
string regKeyName = "Software\\WPFExamples\\IMGVIEW";
```

и определите добавленный в xaml-файл обработчик:

```
private void Window_Closed(object sender, EventArgs e)
{
    RegistryKey rk = null;
    try
    {
        rk = Registry.CurrentUser.CreateSubKey(regKeyName);
        if (rk == null)
            return;
        rk.SetValue("Width", (int)ActualWidth);
        rk.SetValue("Height", (int)ActualHeight);
        rk.SetValue("DirList",
          (int)grid1.ColumnDefinitions[0].ActualWidth);
        rk.SetValue("FileList",
          (int)grid1.ColumnDefinitions[2].ActualWidth);
        rk.SetValue("Stretch", comboBox1.SelectedIndex);
        var dirInfo = (dirList1.SelectedItem as TreeViewItem).Tag
          as DirectoryInfo;
        rk.SetValue("Path", dirInfo == null ? "" :
          dirInfo.FullName);
        rk.SetValue("File", fileList1.SelectedIndex);
    }
    finally
    {
        if (rk != null)
            rk.Close();
    }
}
```

Результат. Теперь при завершении работы программы сведения о ее текущем состоянии записываются в реестр Windows. Для того чтобы в этом убедиться, следует запустить программу regedit (Редактор реестра). Проще всего это сделать, выбрав в главном меню Windows команду «Выполнить…» и указав в появившемся окне имя программы: regedit. В редакторе реестра надо выбрать раздел «HKEY_CURRENT_USER», а в нем подраздел «Software\WPFExamples\IMGVIEW». В результате на правой панели редактора будут отображены поля данного подраздела и их значе-

ния. Подраздел должен содержать (помимо поля «По умолчанию», которое не будет использоваться в нашей программе) семь полей в алфавитном порядке: DirList, File, FileList, Height, Path, Stretch, Width (поле Path является строковым, остальные – целочисленными). В следующем пункте к программе будет добавлен фрагмент, позволяющий получать из реестра эти данные.

Комментарии

1. Реестр Windows является удобным централизованным хранилищем данных, необходимых для корректной работы программ, в частности, для восстановления их настроек при очередном запуске. Если для каждого пользователя компьютера желательно хранить его собственные настройки, то их следует размещать в разделе HKEY_CURRENT_USER. Если для всех пользователей настройки должны быть одинаковыми, то их надо размещать в разделе HKEY_LOCAL_MACHINE. В любом случае в выбранном разделе необходимо создать подраздел, связанный с данной программой (обычно этот подраздел помещается в подраздел «Software» выбранного раздела).

Для доступа к данным реестра в библиотеке .NET предусмотрены классы Registry и RegistryKey, определенные в пространстве имен Microsoft.Win32.

Класс Registry позволяет выбрать один из разделов реестра первого уровня и получить связанный с ним объект типа RegistryKey. Для получения разделов предусмотрены статические свойства класса Registry, доступные только для чтения. Разделу HKEY_CURRENT_USER соответствует свойство CurrentUser, а разделу HKEY_LOCAL_MACHINE – свойство LocalMachine.

Получив с помощью класса Registry объект типа RegistryKey, можно с его помощью *создавать* новые подразделы и *открывать* существующие подразделы в режимах чтения и/или записи. Метод CreateSubKey открывает подраздел в режиме записи; если данный раздел не существует, то метод предварительно создает его. Метод OpenSubKey открывает *существующий* подраздел в режиме «только для чтения». В обоих методах в качестве строкового параметра надо указать полный путь к данному подразделу. Имеется также перегруженный вариант метода OpenSubKey, позволяющий открывать существующий подраздел в режиме «чтение и запись»; для этого в методе надо указать второй, дополнительный параметр, равный true. Если требуемая операция выполнена успешно, то оба метода возвращают объект типа RegistryKey, связанный с открытым подразделом. Если операцию выполнить не удалось, то либо возвращается значение null (например, в случае попытки открыть методом OpenSubKey несуществующий подраздел), либо возбуждается исключение (например, если программа имеет недостаточно прав для доступа к указанному под-

разделу в требуемом режиме). Любой успешно открытый подраздел надо закрыть с помощью метода Close.

Метод SetValue класса RegistryKey позволяет добавлять или изменять поля у открытого подраздела. Он имеет два параметра – имя поля и его новое значение (в качестве типа значений следует использовать string, int или массив байтов). Метод для получения значений полей будет описан в следующем пункте.

2. Для отладки фрагментов программы, связанных с реестром, необходимо использовать редактор реестра regedit, поскольку он позволяет просматривать и редактировать его содержимое. Если обнаружится, что подраздел или какие-либо его поля созданы с ошибками, то с помощью редактора реестра их можно будет легко удалить. Если при загруженном редакторе реестра тестируемая программа будет запущена повторно, то для обновления информации в окне редактора реестра достаточно нажать клавишу F5. При работе с реестром вначале следует отладить часть программы, отвечающую за запись данных в реестр, а затем – ту часть, которая отвечает за считывание этих данных.

3. В методах класса RegustryKey не предусмотрено удобного способа для хранения вещественных данных. Поэтому все сохраняемые в реестре размеры мы приводим к типу int. При этом, разумеется, дробная часть размера окажется потерянной, но эта потеря будет незаметна для пользователя, поскольку составит менее 1/96 дюйма. Обратите также внимание на то, как определяется фактическая ширина столбцов компонента Grid.

4. Используя появившуюся в версии C# 6.0 null-условную операцию «.?» (см. п. 8.1 проекта CURSORS, комментарий 6), оператор

```
rk.SetValue("Path", dirInfo == null ? "" : dirInfo.FullName);
```

можно записать более кратко:

```
rk.SetValue("Path", dirInfo?.FullName ?? "");
```

17.6. Восстановление из реестра Windows информации о состоянии программы

Измените конструктор класса MainWindow следующим образом:

```
public MainWindow()
{
    InitializeComponent();
    TreeViewItem item = new TreeViewItem();
    item.Tag = null;
    item.Header = "Компьютер";
    item.Items.Add("*");
    dirList1.Items.Add(item);
    item.IsSelected = true;
    item = InitialExpanding(Directory.GetCurrentDirectory());
```

```
    if (item != null)
        item.IsSelected = true;
    comboBox1.ItemsSource = Enum.GetValues(typeof(Stretch));
    comboBox1.SelectedIndex = 0;
    string s = "";
    int i = 0;
    RegistryKey rk = null;
    try
    {
        rk = Registry.CurrentUser.OpenSubKey(regKeyName);
        if (rk != null)
        {
            Width = (int)rk.GetValue("Width", (int)Width);
            Height = (int)rk.GetValue("Height", (int)Height);
            grid1.ColumnDefinitions[0].Width =
              new GridLength((int)rk.GetValue("DirList",
              (int)grid1.ColumnDefinitions[0].Width.Value));
            grid1.ColumnDefinitions[2].Width =
              new GridLength((int)rk.GetValue("FileList",
              (int)grid1.ColumnDefinitions[2].Width.Value));
            comboBox1.SelectedIndex = (int)rk.GetValue("Stretch",
              comboBox1.SelectedIndex);
            s = (string)rk.GetValue("Path", "");
            i = (int)rk.GetValue("File", 0);
        }
    }
    finally
    {
        if (rk != null)
            rk.Close();
    }
    if (!Directory.Exists(s))
        s = Directory.GetCurrentDirectory();
    item = InitialExpanding(s);
    if (item != null)
        item.IsSelected = true;
    if (fileList1.Items.Count == 0)
        i = -1;
    else
        if (i >= fileList1.Items.Count || i == -1)
```

```
            i = 0;
    fileList1.SelectedIndex = i;
    dirList1.Focus();
    dirList1.AddHandler(TreeViewItem.ExpandedEvent,
        new RoutedEventHandler(TreeViewItem_Expanded));
    comboBox1.ItemsSource = Enum.GetValues(typeof(Stretch));
    comboBox1.SelectedIndex = 0;
}
```

Результат. При запуске программы данные из реестра Windows используются для восстановления ее состояния. Если данные в реестре отсутствуют, то устанавливаются настройки по умолчанию. Поскольку с момента последнего сохранения данных в реестре состояние дисковой системы могло измениться, при считывании поля Path проверяется наличие указанного в нем каталога, и если он не найден, то используется текущий каталог. Кроме того, прочитанное поле File при необходимости корректируется таким образом, чтобы в случае непустого списка файлов обязательно был выделен какой-либо его элемент.

При отображении на экране окно центрируется с учетом его новых размеров, считанных из реестра.

Комментарии

1. Для считывания значений полей подраздела реестра предназначен метод GetValue класса RegistryKey; при этом нужный подраздел достаточно открыть только на чтение. Наиболее удобным является вариант метода GetValue с *двумя* параметрами: первый параметр содержит имя поля, а второй – значение по умолчанию (метод возвращает значение по умолчанию, если указанное поле в подразделе реестра не найдено). Вариант метода с одним параметром (именем поля) менее удобен, так как при отсутствии данного поля он возвращает значение null. Важно учитывать, что метод GetValue имеет тип object, поэтому его возвращаемое значение необходимо явно привести к типу указанного поля (int или string).

2. Редактор реестра regedit особенно полезен при отладке фрагмента программы, отвечающего за *считывание* данных из реестра. С его помощью можно удалять поля подраздела и сам подраздел, а также вносить изменения в поля, что позволяет протестировать работу программы в различных ситуациях.

18. Табличное приложение с заставкой: TRIGFUNC

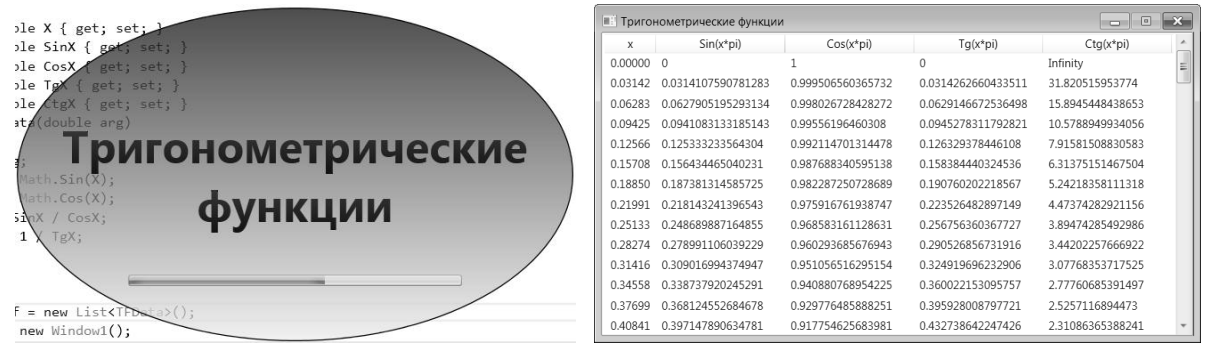


Рис. 67. Окна приложения TRIGFUNC

18.1. Формирование таблицы значений тригонометрических функций

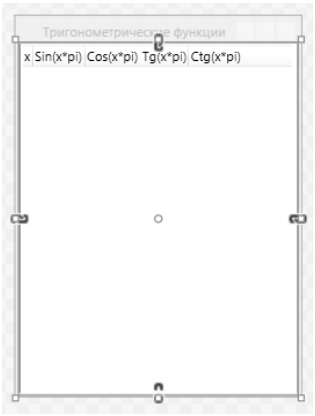


Рис. 68. Макет окна MainWindow приложения TRIGFUNC

```
<Window x:Class="TRIGFUNC.MainWindow"
    …
    Title="Тригонометрические функции" Height="350"
    SizeToContent="Width" ResizeMode="CanMinimize"
    WindowStartupLocation="CenterScreen"
    Loaded="Window_Loaded" >
  <ListView x:Name="listView1" >
    <ListView.View >
      <GridView x:Name="gridView1" >
        <GridViewColumn Header="x"
            DisplayMemberBinding="{Binding Path=X,
              StringFormat='{}{0:f5}'}" />
        <GridViewColumn Header="Sin(x*pi)"
```

```
            DisplayMemberBinding="{Binding Path=SinX}"/>
        <GridViewColumn Header="Cos(x*pi)"
            DisplayMemberBinding="{Binding Path=CosX}"/>
        <GridViewColumn Header="Tg(x*pi)"
            DisplayMemberBinding="{Binding Path=TgX}"/>
        <GridViewColumn Header="Ctg(x*pi)"
            DisplayMemberBinding="{Binding Path=CtgX}"/>
      </GridView>
    </ListView.View>
  </ListView>
</Window>
```

В класс MainWindow добавьте описание вспомогательного вложенного класса TFData и поля tf, содержащего список объектов класса TFData:

```
class TFData
{
    public double X { get; set; }
    public double SinX { get; set; }
    public double CosX { get; set; }
    public double TgX { get; set; }
    public double CtgX { get; set; }
    public TFData(double arg)
    {
        X = arg;
        SinX = Math.Sin(X);
        CosX = Math.Cos(X);
        TgX = SinX / CosX;
        CtgX = 1 / TgX;
    }
}
List<TFData> tf = new List<TFData>();
```

Для окна MainWindow определите обработчик события Loaded, указанный в xaml-файле:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    int n = 7, nMax = 1000001;
    string[] args = Environment.GetCommandLineArgs();
    if (args.Length > 1)
        try
        {
            int n0 = int.Parse(args[1]);
```

```
            if (n0 < 2 || n0 > nMax)
                throw new Exception();
            else
                n = n0;
        }
        catch
        {
            string s = string.Format("Неверный параметр: {0}\n" +
              "Допустимые значения: от 2 до {1}", args[1], nMax);
            MessageBox.Show(s, "Ошибка", MessageBoxButton.OK,
              MessageBoxImage.Error);
            Close();
            return;
        }
    double step = 1.0 / (n - 1);
    for (int i = 0; i < n; i++)
        tf.Add(new TFData(Math.PI * i * step));
    listView1.ItemsSource = tf;
}
```

Результат. Программа заполняет компонент-таблицу listView1 значениями тригонометрических функций с аргументами от 0 до π радиан на сетке из n равноотстоящих точек. Число точек n можно указать в качестве параметра командной строки; допускаются значения от 2 до 1000001. При запуске программы из среды Visual Studio параметры можно указать в поле «Command line arguments» группы настроек Debug в окне свойств проекта (напомним, что данное окно вызывается командой «Project | <*Имя проекта*> Properties»). Если параметр указан неверно, то выводится сообщение об ошибке и программа немедленно завершает работу. Если параметр не указан, то количество точек полагается равным 7.

Хотя размеры окна изменять нельзя, допустимо менять ширину отдельных столбцов, зацепив мышью за разделительную линию между их заголовками и переместив ее в нужном направлении (при этом ширина окна автоматически изменится; это обеспечивается благодаря настройке окна SizeToContent="Width"). Более того, зацепив мышью за заголовок некоторого столбца, можно перетащить его на новую позицию, поменяв тем самым порядок следования столбцов.

Комментарии

1. В данном проекте используется еще один специализированный представитель компонентов типа списка. Это *табличный список* ListView, который является непосредственным потомком «обычного» списка ListBox. Как и ListBox, класс ListView содержит свойство Items для хра-

нения своих элементов, а также свойство ItemsSource, позволяющее сразу задать набор элементов в виде некоторой коллекции. Основным отличием класса ListView от его предка ListBox является наличие дополнительного свойства View типа ViewBase, которое определяет способ отображения элементов (если свойство View равно null, то компонент ListView выглядит так же, как и обычный список).

Единственным потомком класса ViewBase, включенным в библиотеку WPF, является класс GridView, позволяющий отображать элементы класса ListView *в нескольких столбцах с заголовками*. Важнейшим свойством класса GridView является Columns – коллекция типа GridViewColumnCollection с элементами типа GridViewColumn. Объект GridViewColumn определяет, в частности, заголовок столбца (свойство Header), ширину столбца (свойство Width) и то свойство элементов табличного списка ListView, которое должно отображаться в этом столбце (свойство DisplayMemberBinding). Таким образом, для возможности отображения элементов табличного списка ListView в виде набора столбцов необходимо, чтобы эти элементы имели набор свойств, каждое из которых будет связано с некоторым столбцом списка ListView. Для связывания свойства DisplayMemberBinding с нужным свойством элемента табличного списка используется механизм привязки, который наиболее просто реализовать в xaml-файле.

Если свойство Width для объекта GridViewColumn не указано явно, то по умолчанию для столбца устанавливается ширина, достаточная для отображения его фактического содержимого (хотя пользователь может в дальнейшем изменить ширину любого столбца). Если свойство AllowsColumnReorder объекта GridView имеет значение true (это значение по умолчанию), то пользователь может также изменять порядок следования столбцов.

Для заполнения табличного списка надо определить класс со свойствами, которые будут отображаться в таблице (в нашем случае это класс TFData). Затем создается коллекция объектов этого класса (в нашем случае List<TFData>), и эта коллекция связывается со свойством ItemsSource табличного списка. Дополнительно в xaml-файле надо указать, с какими свойствами надо связать каждый столбец (это можно сделать и в коде, но настройки xaml-файла являются более короткими и наглядными).

2. Поскольку ни при записи, ни при чтении свойств класса TFData не требуется предусматривать особых действий, эти свойства сделаны *автоматическими* (ранее автоматическое свойство Modified было использовано нами в проекте TEXTEDIT версии 1 – см. п. 9.5).

3. Обратите внимание на возможность указания *формата строки* при связывании (мы использовали эту возможность для столбца аргументов). Благодаря кавычкам в формат можно добавлять дополнительный

текст. Скобки {} в начале формата нужны в случае, если текст формата начинается с фигурной скобки. Для остальных столбцов формат не указывался, поэтому для них используется числовой формат по умолчанию (так называемый *общий формат* – general), в котором выбирается наиболее краткое из возможных представлений вещественного числа. Для столбца аргументов мы использовали формат f5 – формат с *фиксированным* числом дробных знаков (в данном случае – с пятью знаками после запятой). Первая часть формата (вида «0:») означает, что полученная строка будет иметь ширину, минимально необходимую для хранения строкового представления числа.

4. Для доступа к параметрам командной строки был использован метод GetCommandLineArgs класса Environment, возвращающий строковый массив. Первый элемент этого массива (с индексом 0) содержит *полное имя исполняемого файла приложения*, а начиная со следующего элемента, в массиве размещаются параметры командной строки.

5. Указанный при запуске программы параметр командной строки может оказаться недопустимым по двум причинам: во-первых, он может быть строкой, которую нельзя преобразовать к целому числу, во-вторых, даже если подобное преобразование возможно, полученное число может лежать вне допустимого диапазона. Для того чтобы обрабатывать все ошибки в одном месте программы, мы разместили соответствующий фрагмент в разделе catch try-блока, где делается попытка преобразовать параметр командной строки в целое число. При невозможности подобного преобразования возбуждается исключение, которое сразу передает управление в раздел catch. Если же преобразование прошло успешно, однако число находится вне допустимого диапазона, то для перехода в раздел catch программа явным образом генерирует исключение, используя для этого оператор throw (тип исключения в данном случае роли не играет, поэтому возбуждается исключение типа Exception – общего предка всех классов-исключений).

6. Для немедленного завершения приложения достаточно вызвать метод Close его главного окна. Следует иметь в виду, что даже после вызова данного метода текущий метод проработает до конца. Поэтому после вызова Close необходимо сразу выйти из текущего метода с помощью оператора return.

7. При выполнении вычислений с типом double можно не беспокоиться о возможном переполнении. В частности, при вычислении значения функции ctg в нуле ошибки не произойдет, а в соответствующей ячейке таблицы будет выведено значение «Infinity» («бесконечность»).

8. Начиная с C# версии 6.0 (Visual Studio 2015) строковое выражение, содержащее дополнительные данные, можно получить не только с помощью функции Format

```
    string.Format(
      "Неверный параметр: {0}\nДопустимые значения: от 2 до {1}",
      args[1], nMax);
```

но и применяя появившиеся в этой версии *интерполированные строки* (обратите внимание на символ $ перед открывающей кавычкой):

```
    $"Неверный параметр: {args[1]}\nДопустимые значения: от 2 до {nMax}"
```

Как и в форматных настройках функции Format, в интерполированных строках допускается использовать атрибуты форматирования, которые указываются в тех же фигурных скобках *после* форматируемого выражения, например {x,15:f5} (это означает, что вещественное число x надо вывести в 15 экранных позициях, используя формат f5).

Недочет 1. Поскольку для столбцов мы не указали значения Width, их ширина определяется по содержимому, однако только по тому содержимому, которое *отображается в окне в начале программы*. Если ниже (в невидимой части) присутствуют более длинные значения, то это никак не будет учитываться. В качестве примера на рис. 69 приводится завершающая часть таблицы, построенной по 10000 точкам, в которой ширина трех последних столбцов недостаточна для отображения всех цифр в полученных числах.

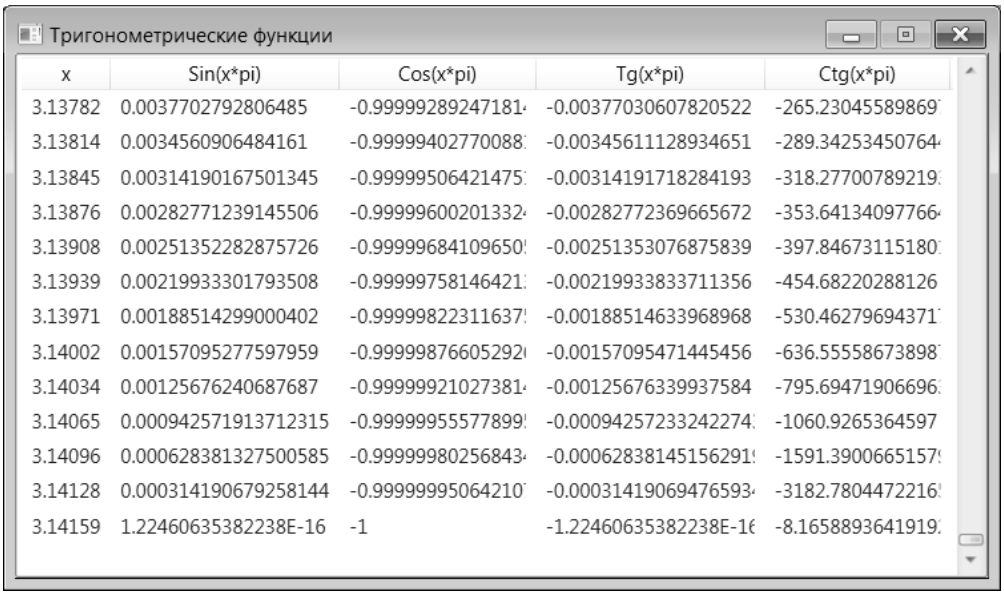


Рис. 69. Вид окна приложения TRIGFUNC с недостаточной шириной столбцов

Разумеется, указанный недочет не является серьезным, так как пользователь может самостоятельно изменять ширину столбцов. Однако все-таки опишем способ его исправления, поскольку он может оказаться полезным в аналогичных ситуациях.

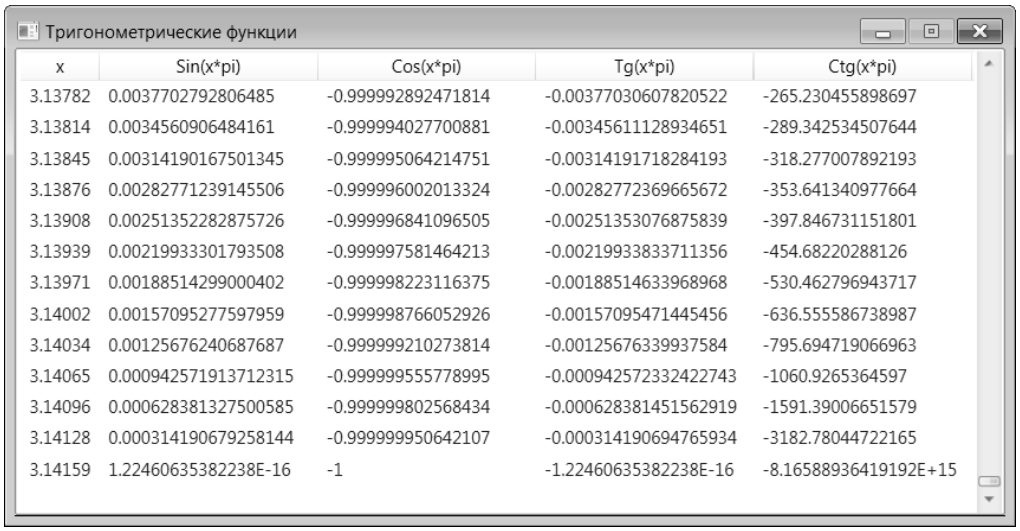
Исправление. Определите обработчик события LayoutUpdated для окна:

```
<Window x:Class="TRIGFUNC.MainWindow"
```

```
    …
    LayoutUpdated="Window_LayoutUpdated" >
```

```
private void Window_LayoutUpdated(object sender, EventArgs e)
{
    double maxWidth = gridView1.Columns
      .Select(e1 => e1.ActualWidth).Max() + 10;
    for (int i = 1; i < gridView1.Columns.Count; i++)
        gridView1.Columns[i].Width = maxWidth;
    LayoutUpdated -= Window_LayoutUpdated;
}
```

Результат. После определения реальных размеров каждого столбца (при этом, как было отмечено выше, учитываются только те данные, которые сразу будут отображаться на экране) ширина столбцов со второго по последний *пересчитывается еще раз*: она полагается равной максимальному из ранее определенных значений ширины *плюс* 10 аппаратно-независимых единиц. Этого достаточно, чтобы все данные полностью отображались в таблице (рис. 70). Кроме того, все столбцы со значениями тригонометрических функций теперь имеют одинаковую ширину. Указанное действие выполняется только после *первого* определения реальных размеров компонентов окна. Заметим, что центрирование окна выполняется правильно, т. е. с учетом увеличившейся ширины окна.



x	Sin(x*pi)	Cos(x*pi)	Tg(x*pi)	Ctg(x*pi)
3.13782	0.0037702792806485	-0.999992892471814	-0.00377030607820522	-265.230455898697
3.13814	0.0034560906484161	-0.999994027700881	-0.00345611128934651	-289.342534507644
3.13845	0.00314190167501345	-0.999995064214751	-0.00314191718284193	-318.277007892193
3.13876	0.00282771239145506	-0.999996002013324	-0.00282772369665672	-353.641340977664
3.13908	0.00251352282875726	-0.999996841096505	-0.00251353076875839	-397.846731151801
3.13939	0.00219933301793508	-0.999997581464213	-0.00219933833711356	-454.68220288126
3.13971	0.00188514299000402	-0.999998223116375	-0.00188514633968968	-530.462796943717
3.14002	0.00157095277597959	-0.999998766052926	-0.00157095471445456	-636.555586738987
3.14034	0.00125676240687687	-0.999999210273814	-0.00125676339937584	-795.694719066963
3.14065	0.000942571913712315	-0.999999555778995	-0.000942572332422743	-1060.9265364597
3.14096	0.000628381327500585	-0.999999802568434	-0.000628381451562919	-1591.39006651579
3.14128	0.000314190679258144	-0.999999950642107	-0.000314190694765934	-3182.78044722165
3.14159	1.22460635382238E-16	-1	-1.22460635382238E-16	-8.16588936419192E+15

Рис. 70. Вид окна приложения TRIGFUNC после корректировки ширины столбцов

Комментарий

Событие LayoutUpdated возникает каждый раз после пересчета размеров компонента. В частности, при его первом возникновении в программе будут доступны реальные размеры компонента, учитывающие содержащиеся в нем данные. Следует заметить, что впервые это событие возникает *после* события Loaded и перед *первым* событием Activated. Таким обра-

зом, исправить выявленный недочет, добавив аналогичные операторы в обработчик события Loaded, не удастся. Можно было бы использовать обработчик для события Activated, но тогда возникли бы проблемы с центрированием окна на экране по горизонтали (с этими проблемами мы еще столкнемся в следующем пункте).

Необходимо, чтобы указанные в обработчике Window_LayoutUpdated действия были выполнены *единственный раз* – после первого вычисления реальных размеров столбцов. Это обеспечивает последний оператор обработчика, который *отсоединяет* обработчик Window_LayoutUpdated от события LayoutUpdated. Заметим, что при отсутствии этого оператора произошло бы зависание программы, так как любое действие по изменению размеров компонентов приводит к возникновению события LayoutUpdated и, таким образом, действия в методе Window_LayoutUpdated приводили бы к повторному вызову этого метода, и так до бесконечности.

18.2. Отображение окна-заставки при загрузке программы

Добавьте к проекту новое окно Window1, выполнив действия, описанные в начале раздела, посвященного проекту WINDOWS (п. 2.1).

Рис. 71. Макет окна Window1 приложения TRIGFUNC (первый вариант)

Window1.xaml

```
<Window x:Class="TRIGFUNC.Window1"
    …
    Title="Window1" Height="300" AllowsTransparency="True"
    Width="500" WindowStartupLocation="CenterScreen"
    WindowStyle="None" Background="Transparent"
    ResizeMode="NoResize" ShowInTaskbar="False" >
  <Grid x:Name="grid1" >
    <Ellipse Stroke="Black" >
```

```
      <Ellipse.Fill>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
          <GradientStop Color="#BF0000FF" Offset="0"/>
          <GradientStop Color="#BFFFFF00" Offset="1"/>
        </LinearGradientBrush>
      </Ellipse.Fill>
    </Ellipse>
    <TextBlock TextWrapping="Wrap"
        Text="Тригонометрические функции" FontSize="40"
        TextAlignment="Center" VerticalAlignment="Center"
        FontWeight="Bold" >
      <TextBlock.Foreground>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
          <GradientStop Color="Red" Offset="0"/>
          <GradientStop Color="Blue" Offset="1"/>
        </LinearGradientBrush>
      </TextBlock.Foreground>
    </TextBlock>
  </Grid>
</Window>
```

Файл Window1.xaml.cs изменять не требуется.

В класс MainWindow добавьте поле

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

В *начало* метода Window_Loaded добавьте операторы:

```
LayoutUpdated -= Window_LayoutUpdated;
Mouse.OverrideCursor = Cursors.AppStarting;
win1.Owner = this;
win1.Show();
```

В *конец* метода Window_Loaded добавьте операторы:

```
win1.Hide();
Mouse.OverrideCursor = null;
```

Результат. В течение загрузки главного окна и заполнения табличного списка listView1 на экране отображается окно Window1 – *заставка,* свидетельствующая о том, что программа запущена и в данный момент выполняет инициализирующие действия. Заставка имеет форму эллипса, не содержит заголовка и является полупрозрачной; ее нельзя перемещать по экрану. Для фона заставки используется вертикальная градиентная заливка (от синего цвета к желтому); такой же вариант заливки, только с другими цветами (от красного к синему), использован для текста заставки. Курсор мыши на заставке принимает вид курсора ожидания окончания загрузки программы. При отображении главного окна заставка исчезает.

Комментарий

При оформлении окна-заставки мы использовали средства, относящиеся к сильным сторонам библиотеки WPF: градиентные кисти и возможность простого создания окон произвольной формы.

Чтобы скрыть рамку и заголовок окна, достаточно установить его свойство WindowStyle равным None. Чтобы части окна, не занятые его содержимым, не отображались на экране, необходимо установить два свойства, связанные с прозрачностью, – AllowsTransparency="True" и Background="Transparent".

Содержимым окна-заставки является компонент Grid, в единственной ячейке которого размещены эллипс и текстовый компонент TextBlock. По умолчанию эллипс занимает все пространство ячейки, а текстовый компонент растягивается по ее ширине. Центрирование текстового компонента по вертикали и центрирование самого текста по горизонтали надо настраивать явным образом.

Для фона эллипса и цвета текста компонента TextBlock используются градиентные кисти типа LinearGradientBrush. Свойства StartPoint и EndPoint определяют начальную и конечную точку отрезка, в пределах которого будет происходить изменение цвета, причем с левой верхней границей компонента, для которого определяется градиентная кисть, связывается координата (0, 0), а с его правой нижней границей – координата (1, 1). Указав в качестве концов отрезка точки (0.5, 0) и (0.5, 1), мы тем самым задали *вертикальное направление* градиента («сверху вниз»; вместо данных точек можно было указать, например, (0, 0) и (0, 1)). Кроме того, для градиентной кисти надо указать не менее двух точек на отрезке с требуемыми значениями цветов (цвета других точек будут вычисляться автоматически). Эти точки задаются в свойствах GradientStop, причем кроме цвета Color следует указать значение Offset – смещение от начальной точки отрезка (считается, что конечная точка имеет смещение 1).

Все эти настройки для объектов LinearGradientBrush можно установить и программно, однако более наглядно это выглядит в xaml-разметке. Здесь мы встречаемся с ситуацией, когда свойство компонента (Background для компонента Ellipse и Foreground для компонента TextBlock) *нельзя* задать в виде единственного атрибута. В этом случае свойство оформляется в виде *дочернего XML-элемента* данного компонента и получает составное имя, включающее имя компонента и имя свойства (в нашем случае Ellipse.Background и TextBlock.Foreground). Напомним, что подобные элементы xaml-файла называются *элементами-свойствами* (см. проект WINDOWS, последний комментарий в п. 2.1). В элементе-свойстве можно задавать произвольное количество настроек –

как в виде атрибутов-свойств, так и в виде дочерних элементов-свойств следующего уровня (в нашем случае это элементы GradientStop).

Недочет 1. Требуемые вычисления обычно выполняются так быстро, что рассмотреть заставку не удается из-за краткого времени ее отображения (некоторое замедление в ходе начальных вычислений возникает только для числа точек, близкого к максимальному).

Исправление. В методе Window_Loaded *перед* оператором

```
win1.Hide();
```

вставьте оператор

```
System.Threading.Thread.Sleep(1000);
```

Результат. После завершения инициализирующих действий, но перед скрытием окна-заставки программа приостанавливает работу на 1 секунду.

Комментарий

Метод Sleep класса Thread, описанного в пространстве имен System.Threading, приостанавливает выполнение программы на указанное число миллисекунд.

Недочет 2. Второй недочет оказывается неожиданным: ширина столбцов в главном окне стала пересчитываться неправильно!

Причем, судя по виду окна, в котором даже начальные данные отображаются не полностью (рис. 72), это означает, что обработчик Window_LayoutUpdated сработал еще *до того*, как были определены требуемые размеры столбцов. Эксперименты с программой показывают, что, действительно, из-за вызова win1.Show() возникает дополнительное (первое по счету) событие LayoutUpdated для главного окна, при котором фактические размеры столбцов таблицы *еще определяются неверно*.

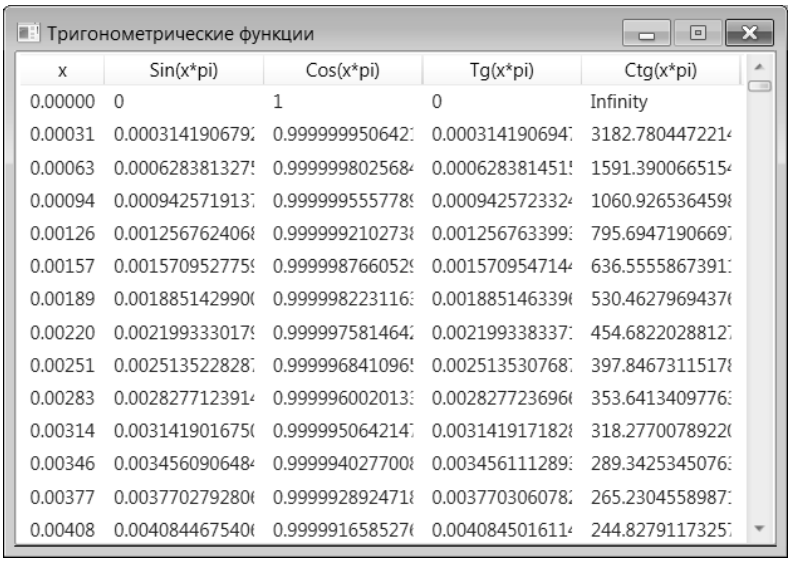


Рис. 72. Вид окна приложения TRIGFUNC с недостаточной шириной столбцов

Исправление. Поскольку проблема возникла после добавления в метод Window_Loaded вызова, связанного с отображением окна-заставки,

возникает мысль *отсоединить* на это время обработчик Window_LayoutUpdated от события LayoutUpdated, чтобы обусловленное этим вызовом событие LayoutUpdated не привело к (неправильному) пересчету размеров столбцов. Итак, добавим в начало метода Window_Loaded оператор

```
LayoutUpdated -= Window_LayoutUpdated;
```

а в конец этого же метода – оператор

```
LayoutUpdated += Window_LayoutUpdated;
```

Результат. Теперь главное окно отображается со столбцами правильной ширины, однако неожиданно возникает новая проблема.

Недочет 3. Главное окно неверно центрируется по горизонтали. Не доискиваясь до причин этого недочета, попытаемся его исправить простейшим способом. У нас имеется обработчик Window_LayoutUpdated, в котором правильно устанавливаются размеры столбцов, а вместе с ними и новые размеры окна. Используя эти новые размеры окна, мы можем *самостоятельно* обеспечить его центрирование.

Исправление. Добавьте в конец метода Window_LayoutUpdated следующий оператор:

```
Left = (SystemParameters.WorkArea.Width - ActualWidth) / 2;
```

Указанное добавление еще не исправляет недочет, однако причины этого понятны: для того чтобы обновилось значение свойства ActualWidth, используемое в добавленном операторе, необходимо, чтобы *еще раз* сработало событие LayoutUpdated, учитывающие те изменения размеров, которые мы сделали в методе Window_LayoutUpdated.

Завершающая часть исправления. Добавьте в класс MainWindow вспомогательный метод

```
public static void DoEvents()
{
    Application.Current.Dispatcher.Invoke(System.Windows
      .Threading.DispatcherPriority.Background,
      new Action(delegate { }));
}
```

и вызовите его *после* оператора, отсоединяющего обработчик от события, но *перед* оператором изменения свойства Left, который был ранее добавлен в метод Window_LayoutUpdated:

```
LayoutUpdated -= Window_LayoutUpdated;
DoEvents();
Left = (SystemParameters.WorkArea.Width - ActualWidth) / 2;
```

Результат. Теперь центрирование выполняется правильно.

Комментарии

1. При вычислении новой позиции окна мы использовали статическое свойство WorkArea.Width стандартного класса SystemParameters, позволяющее определить ширину *клиентской области экрана*, т. е. той его части, которая не занята панелью задач (напомним, что панель задач, хотя обычно и располагается у нижней границы экрана, может быть пристыкована к любой из его границ). Свойство WorkArea.Height позволяет определить высоту клиентской области экрана. Заметим, что класс SystemParameters содержит огромное количество свойств, позволяющих определить самые разные настройки компьютера, в том числе размеры стандартных визуальных элементов (например, высоту заголовка окна или толщину его границ).

2. Вызов метода DoEvents обеспечивает немедленную обработку всех событий, возникших к данному моменту в приложении. В библиотеке Windows Forms данный метод входил в состав стандартного класса Application, однако в библиотеке WPF он был признан устаревшим и поэтому не реализован. Мы использовали собственный вариант данного метода, основанный на следующей идее: вызывается диспетчер приложения и ему дается задача выполнить «пустой» делегат, причем с низким приоритетом. Пока не будут завершены все основные действия по обработке предыдущих событий (в частности, пока не будут пересчитаны размеры окна с учетом новых значений ширины столбцов), этот делегат не будет выполнен. А если он выполнен и произошел возврат из метода DoEvents, то это означает, что все размеры окна обновились, и их можно использовать для центрирования.

18.3. Отображение индикатора прогресса при загрузке программы

Рис. 73. Макет окна Window1 приложения TRIGFUNC (второй вариант)

```
<Window x:Class="TRIGFUNC.Window1"
```

```
    … >
  <Grid x:Name="grid1" >
    <Ellipse … >
        …
    </Ellipse>
    <TextBlock … >
    </TextBlock>
    <ProgressBar x:Name="progressBar1"
        HorizontalAlignment="Center" Margin="0,50"
        Height="10" VerticalAlignment="Bottom" Width="300"
        Background="#00000000" Foreground="Blue" />
  </Grid>
</Window>
```

Измените цикл в методе Window_Loaded для класса MainWindow следующим образом:

```
for (int i = 0; i < n; i++)
{
    win1.progressBar1.Value = 100 * i / (n - 1);
    tf.Add(new TFData(Math.PI * i * step));
}
```

Результат. Внесенные изменения предназначены для отображения в окне-заставке *индикатора прогресса* (компонент ProgressBar), показывающего, какая часть вычислений выполнена к текущему моменту.

Комментарий

Компонент ProgressBar позволяет отображать информацию о ходе выполнения некоторого процесса. Вид индикатора зависит от значений трех вещественных свойств – Value, Minimum (значения обоих свойство по умолчанию равны 0) и Maximum (значение по умолчанию 100). Значение свойства Value лежит в диапазоне от Minimum до Maximum и определяет, какая часть индикатора будет выделена.

Ошибка 1. Даже для максимального количества точек, равного 1000001, состояние индикатора прогресса на экране не изменяется. Данная ошибка связана с тем, что изменение индикатора прогресса связано с его перерисовкой, которая не выполняется, пока не произойдет выход из метода Window_Loaded.

Исправление. Добавьте в цикл вызов метода DoEvents:

```
for (int i = 0; i < n; i++)
{
    win1.progressBar1.Value = 100 * i / (n - 1);
    DoEvents();
    tf.Add(new TFData(Math.PI * i * step));
```

```
}
```

Ошибка 2. Теперь индикатор прогресса обновляется, но время работы метода Window_Loaded существенно увеличилось. Это связано с тем, что вызов DoEvents требует определенного времени на выполнение, а в данном варианте программы он выполняется на *каждой* итерации цикла.

Исправление. Добавьте в цикл условный оператор:

```
for (int i = 0; i < n; i++)
{
    if (win1.progressBar1.Value != 100 * i / (n - 1))
    {
        win1.progressBar1.Value = 100 * i / (n - 1);
        DoEvents();
    }
    tf.Add(new TFData(Math.PI * i * step));
}
```

Результат. Теперь вызов метода DoEvents выполняется только на тех итерациях, на которых *изменяется* свойство Value компонента ProgressBar (т. е. когда действительно требуется выполнить его перерисовку).

Комментарий

Завершая описание данного проекта, отметим, что в библиотеке WPF предусмотрен класс SplashScreen, специально предназначенный для отображения заставки – графического изображения, которое берется из ресурсов приложения (имя файла, содержащего графическое изображение, является параметром конструктора класса SplashScreen). Для отображения заставки предназначен метод Show с логическим параметром autoClose (если параметр равен true, то заставка автоматически исчезает при отображении на экране главного окна). Для скрытия заставки, показанной методом Show с параметром false, предназначен метод Close с параметром fadeoutDuration типа TimeSpan (параметр определяет промежуток времени, в течение которого заставка будет постепенно «выцветать» на экране перед своим полным исчезновением). Разумеется, никакие визуальные компоненты, вроде индикатора прогресса, в заставках такого типа использовать нельзя. Зато подобные заставки не создают проблем, с которыми мы столкнулись в п. 18.2 (см. описание недочета 2).

19. Создание компонентов во время выполнения программы: HTOWERS

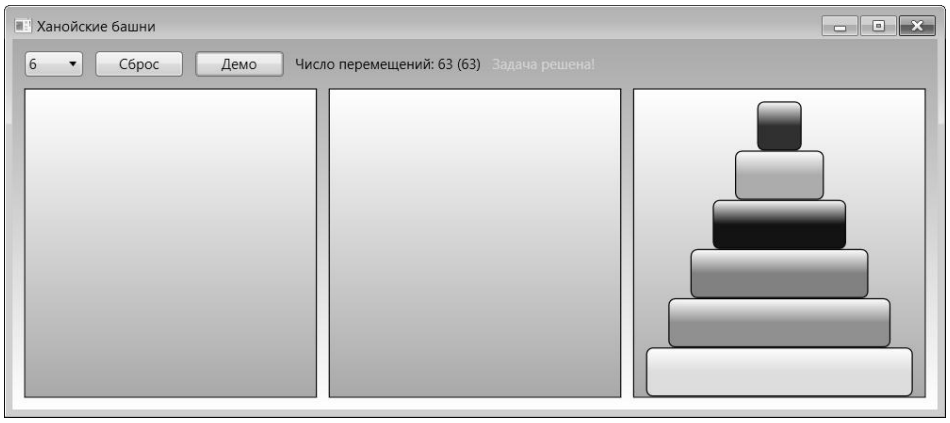


Рис. 74. Окно приложения HTOWERS

Программа, которая будет разработана в данном примере, представляет собой компьютерную реализацию известной логической задачи «Ханойские башни». Эта задача состоит в следующем: имеются три колышка, на один из которых нанизано (в порядке уменьшения размера) несколько дисков, образующих «башню»; требуется переместить всю башню на один из пустых колышков, пользуясь другим пустым колышком как вспомогательным. Переносить можно по одному диску, причем *больший диск нельзя помещать на меньший*. Заметим, что для решения задачи с N дисками минимальное число переносов равно 2^N–1 (см., например, [6, гл. 1]).

В нашей программе вместо колышков будут использоваться три прямоугольные области (компоненты DockPanel), а вместо дисков между этими областями будут перемещаться прямоугольные блоки (компоненты Rectangle).

19.1. Настройка начальной позиции

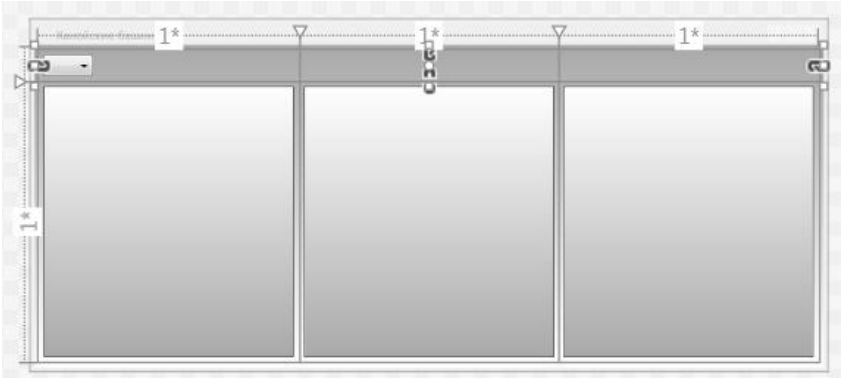


Рис. 75. Макет окна приложения HTOWERS

```
<Window x:Class="HTOWERS.MainWindow"
    …
    Title="Ханойские башни" Height="350" Width="800"
    WindowStartupLocation="CenterScreen" Loaded="Window_Loaded" >
  <Window.Resources>
    <LinearGradientBrush x:Key="GrayBrush" EndPoint="0.5,1"
        StartPoint="0.5,0" >
      <GradientStop Color="White" Offset="0" />
      <GradientStop Color="DarkGray" Offset="1" />
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="GrayBrush1" EndPoint="0.5,1"
        StartPoint="0.5,0" >
      <GradientStop Color="DarkGray" Offset="0" />
      <GradientStop Color="White" Offset="1" />
    </LinearGradientBrush>
  </Window.Resources>
  <Window.Background>
    <StaticResource ResourceKey="GrayBrush1"/>
  </Window.Background>
  <Grid Margin="5" >
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <StackPanel Grid.ColumnSpan="3" Orientation="Horizontal" >
      <ComboBox x:Name="comboBox1" Width="50" Margin="5"
          SelectionChanged="comboBox1_SelectionChanged" />
    </StackPanel>
    <Border BorderBrush="Black" BorderThickness="1" Grid.Row="1"
        Margin="5">
      <DockPanel x:Name="panel1" LastChildFill="False"
          Background="{StaticResource ResourceKey=GrayBrush}" />
    </Border>
    <Border BorderBrush="Black" BorderThickness="1" Grid.Row="1"
        Grid.Column="1" Margin="5">
```

```
      <DockPanel x:Name="panel2" LastChildFill="False"
          Background="{StaticResource ResourceKey=GrayBrush}" />
    </Border>
    <Border BorderBrush="Black" BorderThickness="1" Grid.Row="1"
        Grid.Column="2" Margin="5">
      <DockPanel x:Name="panel3" LastChildFill="False"
          Background="{StaticResource ResourceKey=GrayBrush}" />
    </Border>
  </Grid>
</Window>
```

В класс MainWindow добавьте два поля:

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

В конструктор класса MainWindow добавьте два оператора:

```
comboBox1.ItemsSource = Enumerable.Range(2, 9)
  .Select(e => e.ToString());
comboBox1.Focus();
```

И определите обработчики события Loaded для окна и события SelectionChanged для компонента combobox1 (эти обработчики уже указаны в xaml-файле):

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    comboBox1.SelectedIndex = 8;
}
private void comboBox1_SelectionChanged(object sender,
  SelectionChangedEventArgs e)
{
    total = int.Parse(comboBox1.SelectedItem.ToString());
    panel1.Children.Clear();
    panel2.Children.Clear();
    panel3.Children.Clear();
    var w = (panel1.ActualWidth - 20) / (2 * total);
    for (int i = 0; i < total; i++)
    {
        var r = new Rectangle();
        r.Width = panel1.ActualWidth - (20 + i * 2 * w);
        r.Height = (panel1.ActualHeight - 10) / total;
        r.Stroke = Brushes.Black;
        r.StrokeThickness = 1;
        DockPanel.SetDock(r, Dock.Bottom);
        LinearGradientBrush b = new LinearGradientBrush();
```

```
        b.StartPoint = new Point(0.5, 0);
        b.EndPoint = new Point(0.5, 0.5);
        Color c1 = Color.FromRgb((byte)rnd.Next(256),
          (byte)rnd.Next(256), (byte)rnd.Next(256));
        b.GradientStops.Add(new GradientStop(Colors.White, 0));
        b.GradientStops.Add(new GradientStop(c1, 1));
        r.Fill = b;
        r.RadiusX = r.Height / 8;
        r.RadiusY = r.Height / 8;
        panel1.Children.Add(r);
    }
}
```

Результат. С помощью выпадающего списка comboBox1 можно выбирать количество блоков (от двух до десяти). При запуске программы устанавливается количество блоков, равное десяти, и эти блоки изображаются на первой панели (рис. 76). Обратите внимание на то, что размеры окна можно изменять; при этом изменяются размеры панелей, однако сохраняются размеры «старых» блоков (созданных до изменения размеров окна).

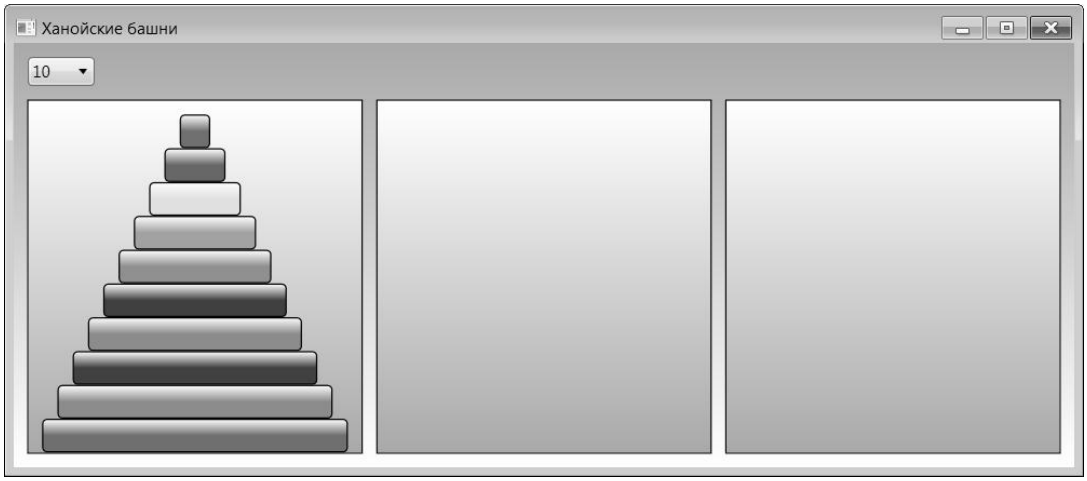


Рис. 76. Окно приложения HTOWERS после запуска

Комментарии

1. В данной программе, как и в предыдущей, мы активно используем *градиентные кисти*. Они устанавливаются в качестве фона для всего окна, каждой панели и каждого прямоугольного блока. При этом кисти для окна и панелей определяются в xaml-файле, а кисть для блоков – программно, в обработчике comboBox1_SelectionChanged. В кисти для блоков вертикальный отрезок градиента начинается на вершине блока, а заканчивается в его *центре*, поэтому градиентный переход изображается только в верхней половине блока.

При определении градиентных кистей для фона окна и панелей были использованы *статические ресурсы*. Это позволило избежать копирования всех настроек кистей для каждой панели.

2. Панель DockPanel удобна для наших целей тем, что упрощает размещение блоков: каждый последующий блок автоматически размещается на предыдущем, так как для всех блоков установлено присоединенное свойство DockPanel.Dock, равное Bottom. Для того чтобы последний дочерний компонент панели также мог иметь такое значение свойства Dock, для самой панели необходимо установить свойство LastChildFill равным false (если этого не сделать, то последний компонент будет захватывать всю свободную область панели). Вокруг каждой панели рисуется рамка – компонент Border.

3. Компонент Rectangle, используемый нами для отображения блоков, позволяет изображать не только обычные прямоугольники, но и прямоугольники со скругленными вершинами, чем мы также воспользовались, задав свойства RadiusX и RadiusY. Для отображения созданного прямоугольника на панели его необходимо поместить в коллекцию Children этой панели.

19.2. Перетаскивание блоков на новое место

Для каждого элемента DockPanel в xaml-файле добавьте новый атрибут:

```
AllowDrop="True"
```

В конец цикла for метода combobox1_SelectionChanged добавьте оператор:

```
r.MouseDown += R_MouseDown;
```

Заметим, что после ввода начальной части этого оператора r.MouseDown += появится приглашение завершить ввод, нажав клавишу Tab. Если это сделать, то будет не только завершен ввод данного оператора, но и создана заготовка для обработчика R_MouseDown следующего вида:

```
private void R_MouseDown(object sender, MouseButtonEventArgs e)
{
    throw new NotImplementedException();
}
```

Измените этот обработчик:

```
private void R_MouseDown(object sender, MouseButtonEventArgs e)
{
    if (e.ChangedButton == MouseButton.Left)
        DragDrop.DoDragDrop(sender as Rectangle, sender,
          DragDropEffects.Move);
}
```

В классе MainWindow определите два вспомогательных метода:

```
DockPanel GetPanel(object trg)
{
    var panel = trg as DockPanel;
    if (panel != null)
        return panel;
    var r = trg as Rectangle;
    if (r != null)
        return r.Parent as DockPanel;
    return null;
}
void MoveRect(Rectangle r, DockPanel panel)
{
    (r.Parent as DockPanel).Children.Remove(r);
    panel.Children.Add(r);
}
```

Используя xaml-файл, определите для компонента panel1 обработчики событий DragEnter и Drop:

```
<DockPanel x:Name="panel1" … DragEnter="panel1_DragEnter"
    Drop="panel1_Drop" />
```

```
private void panel1_DragEnter(object sender, DragEventArgs e)
{
    var panel = GetPanel(e.Source);
    if (panel == null)
        return;
    var r = e.Data.GetData(typeof(Rectangle)) as Rectangle;
    var oldPanel = r.Parent as DockPanel;
    e.Effects = oldPanel.Children.IndexOf(r) ==
      oldPanel.Children.Count - 1 ?
      DragDropEffects.Move : DragDropEffects.None;
    e.Handled = true;
}
private void panel1_Drop(object sender, DragEventArgs e)
{
    var panel = GetPanel(e.Source);
    if (panel == null)
        return;
    var r = e.Data.GetData(typeof(Rectangle)) as Rectangle;
    if (panel == r.Parent)
        return;
```

```
    rect_Move(r, panel);
}
```

После создания обработчиков *переместите* в xaml-файле связанные с ними атрибуты DragEnter="panel1_DragEnter" и Drop="panel1_Drop" в элемент Grid и дополнительно укажите в элементе Grid атрибут, связывающий событие DragOver с уже имеющимся обработчиком panel1_DragEnter:

```
<Grid Margin="5" DragEnter="panel1_DragEnter"
    DragOver="panel1_DragEnter" Drop="panel1_Drop" >
  …
  <Border … >
    <DockPanel x:Name="panel1" … DragEnter="panel1_DragEnter"
        Drop="panel1_Drop" />
```

Результат. Верхний блок (компонент Rectangle) любой башни можно переместить на другую панель DockPanel, причем перемещенный блок всегда будет располагаться на вершине башни. Блоки, расположенные под верхним блоком, перемещать нельзя.

Комментарии

1. Механизм перетаскивания Drag & Drop был подробно рассмотрен в проекте ZOO, п. 7.1.

Чтобы упростить действия по перетаскиванию, следует разрешить отпускать перетаскиваемый блок не только над «пустым» пространством панели, но и над уже имеющимися на ней блоками. Однако в последней ситуации приемником, реагирующим на события перетаскивания, будет не панель, а компонент Rectangle, для которого дополнительно потребуется определить его родителя, чтобы в любом случае получить ту панель, на которую надо переместить блок-источник. Указанные действия мы оформили в виде вспомогательного метода GetPanel.

Вспомогательный метод MoveBlock обеспечивает «смену родителя» для перемещаемого блока. В дальнейшем он будет дополнен новыми операторами.

2. Если использовать появившуюся в языке C# версии 6.0 null-условную операцию «.?» (см. проект CURSORS, п. 8.1, комментарий 6), то тело метода GetPanel можно представить в виде *единственного* оператора:

```
DockPanel GetPanel(object trg)
{
    return (trg as DockPanel) ??
      (trg as Rectangle)?.Parent as DockPanel;
}
```

Осталось учесть *дополнительное условие задачи*: блок может перемещаться либо на пустое место, либо на башню с верхним блоком *большего* размера. Для этого откорректируйте метод panel1_DragEnter:

```
private void panel1_DragEnter(object sender, DragEventArgs e)
{
    var panel = GetPanel(e.Source);
    if (panel == null)
        return;
    var r = e.Data.GetData(typeof(Rectangle)) as Rectangle;
    var oldPanel = r.Parent as DockPanel;
    var c = panel.Children.Count;
    var k = c > 0 ? (panel.Children[c - 1] as Rectangle).Width :
      double.MaxValue;
    e.Effects = oldPanel.Children.IndexOf(r) ==
      oldPanel.Children.Count - 1 && r.Width <= k ?
      DragDropEffects.Move : DragDropEffects.None;
    e.Handled = true;
}
```

Результат. При перемещении блока учитывается дополнительное условие.

Комментарий

В переменную k записывается ширина верхнего блока панели-приемника или максимальное значение типа double, если панель-приемник не содержит блоков. В начале перетаскивания (когда источник находится над панелью-родителем) курсор не является запрещающим благодаря тому, что была использована операция нестрогого сравнения <=.

19.3. Восстановление начальной позиции, подсчет числа перемещений блоков и контроль за решением задачи

```
<Window x:Class="HTOWERS.MainWindow" … >
  …
    <StackPanel Grid.ColumnSpan="3" Orientation="Horizontal" >
      <ComboBox x:Name="comboBox1" Width="50" Margin="5"
          SelectionChanged="comboBox1_SelectionChanged" />
      <Button x:Name="button1" Content="Сброс" Width="75"
          Margin="5" Click="button1 Click" />
      <TextBlock x:Name="label1" Text="" Margin="5"
          VerticalAlignment="Center" />
      <TextBlock x:Name="label2" Text="Задача решена!" Margin="5"
          Foreground="LightGreen" VerticalAlignment="Center" />
```

```
    </StackPanel>
  …
</Window>
```

В описание класса MainWindow добавьте новые поля

```
int count;
int minCount;
```

и вспомогательный метод:

```
void Info()
{
    label1.Text = string.Format("Число перемещений: {0} ({1})",
      count, minCount);
}
```

Перед циклом for в методе comboBox1_SelectionChanged добавьте новые операторы:

```
count = 0;
minCount = (int)Math.Round(Math.Pow(2, total)) - 1;
Info();
label2.Visibility = Visibility.Hidden;
```

В метод MoveRect добавьте следующие операторы:

```
count++;
Info();
if (panel2.Children.Count == total ||
  panel3.Children.Count == total)
    label2.Visibility = Visibility.Visible;
```

И определите обработчик события Click для кнопки button1, указанный в xaml-файле:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    comboBox1_SelectionChanged(null, null);
}
```

Результат. 1. Для восстановления начальной позиции с тем же количеством блоков следует нажать кнопку «Сброс». Если при восстановлении начальной позиции требуется изменить число блоков, то по-прежнему достаточно указать новое значение в компоненте comboBox1 (нажимать кнопку «Сброс» в этой ситуации не требуется).

2. Информация о числе перемещений блоков выводится в тексте метки label1. Там же, в скобках, указывается количество перемещений, минимально необходимое для решения задачи с данным числом блоков n (это количество равно $2^n - 1$).

3. Задача считается решенной (и об этом выводится сообщение на экран), если размер башни в конечной позиции (на панели panel2 или panel3) равен общему числу блоков.

Комментарии

1. Обратите внимание на то, что перемещения блока в пределах *одного и того же* компонента DockPanel при подсчете числа перемещений не учитываются.

2. Для нахождения величины 2^n была использована функция Pow класса Math. Поскольку она возвращает результат типа double, полученное значение должно преобразовываться к целому типу. Так как при преобразовании (int) происходит *отбрасывание* дробной части, мы «на всякий случай» предварительно выполняем округление полученного числа до ближайшего целого с помощью функции Round. Заметим, что необходимость в преобразовании (int) сохраняется, так как функция Round возвращает значение типа double (хотя и с нулевой дробной частью).

Недочет. После решения задачи по-прежнему разрешено перемещение блоков.

Исправление. Добавьте в начало метода panel1_DragEnter следующий фрагмент:

```
if (label2.IsVisible)
{
    e.Effects = DragDropEffects.None;
    e.Handled = true;
    return;
}
```

Результат. Теперь после решения задачи блоки нельзя перемещать.

19.4. Демонстрационное решение задачи

```
<Window x:Class="HTOWERS.MainWindow" … >
  …
    <StackPanel Grid.ColumnSpan="3" Orientation="Horizontal" >
      <ComboBox x:Name="comboBox1" Width="50" Margin="5"
          SelectionChanged="comboBox1_SelectionChanged" />
      <Button x:Name="button1" Content="Сброс" Width="75"
          Margin="5" Click="button1_Click" />
      <Button x:Name="button2" Content="Демо" Width="75"
          Margin="5" Click="button2_Click" />
      …
    </StackPanel>
  …
</Window>
```

В начало метода R_MouseDown добавьте следующий фрагмент:

```
if (!button1.IsEnabled)
    return;
```

В описание класса MainWindow добавьте два вспомогательных метода:

```
public static void DoEvents()
{
    Application.Current.Dispatcher.Invoke(System.Windows
      .Threading.DispatcherPriority.Background,
      new Action(delegate { }));
}
private void Step(int k, DockPanel src, DockPanel dst,
  DockPanel tmp)
{
    if (k == 0)
        return;
    Step(k - 1, src, tmp, dst);
    if (button1.IsEnabled)
        return;
    MoveRect(src.Children[src.Children.Count - 1] as Rectangle,
      dst);
    DoEvents();
    System.Threading.Thread.Sleep(1500 / (total - 1));
    Step(k - 1, tmp, dst, src);
}
```

И определите обработчик события Click для кнопки button2, уже указанный в xaml-файле:

```
private void button2_Click(object sender, RoutedEventArgs e)
{
    comboBox1.IsEnabled = button1.IsEnabled = !button1.IsEnabled;
    if (!button1.IsEnabled)
    {
        if (panel1.Children.Count != total)
            comboBox1_SelectionChanged(null, null);
        Step(total, panel1, panel3, panel2);
        comboBox1.IsEnabled = button1.IsEnabled = true;
    }
}
```

Результат. При нажатии на кнопку «Демо» программа переходит в *демо-режим*, показывающий правильное решение задачи с указанным числом блоков (блоки перемещаются автоматически). В демо-режиме бло-

кируется компонент comboBox1 и кнопка «Сброс»; кроме того, в нем запрещено перетаскивание блоков. Выход из демо-режима происходит после завершения решения задачи, а также при повторном нажатии на кнопку «Демо» (в последнем случае после выхода из демо-режима можно продолжать решать задачу самостоятельно).

Комментарий

Для демонстрационного решения задачи используется рекурсивный алгоритм, реализованный в методе Step. Чтобы за ходом решения можно было проследить, после каждого автоматического перемещения блока наступает пауза, длительность которой зависит от общего количества блоков (пауза тем короче, чем больше количество блоков). Обратите внимание на вызов метода DoEvents (ранее этот метод использовался в проекте TRIGFUNC). При отсутствии метода DoEvents в окне не будет выполняться перерисовка компонентов в ходе выполнения рекурсивного алгоритма, и на экране появится только заключительная конфигурация блоков на третьей панели.

Ошибка. При попытке завершить программу, находящуюся в демо-режиме, возникает исключение, связанное с тем, что ранее вызванные методы Step продолжают выполняться уже после того, как компоненты окна были разрушены в результате завершающих действий программы.

Исправление. Определите обработчик события Closed для окна:

```
<Window x:Class="HTOWERS.MainWindow"
    … Closed="Window_Closed" >
```

```
private void Window_Closed(object sender, EventArgs e)
{
    button1.IsEnabled = true;
}
```

Результат. Теперь при закрытии окна кнопка button1 делается доступной, что позволяет практически немедленно завершить все рекурсивные вызовы метода Step благодаря присутствующему в методе Step условному оператору.

20. Учебные задания

20.1. Проект DIALOGS: взаимодействие между окнами

Общие указания. Для диалоговых (*модальных*) окон необходимо настраивать стандартную работу клавиш Enter и Esc (клавиша Enter всегда связывается к кнопкой «ОК» или ее аналогом, клавиша Esc – с клавишей «Отмена» или ее аналогом). При повторном открытии диалогового окна в нем должна либо сохраняться предыдущая информация, либо выводиться новая информация, если таково условие задачи. В любом случае необходимо обеспечивать активизацию первого компонента диалогового окна. Диалоговое окно не должно содержать доступных кнопок минимизации и максимизации. Главное окно должно содержать доступную кнопку минимизации; максимизация должна быть доступна только для главного окна, которое может изменять свой размер. В качестве меток можно использовать компоненты TextBlock. По умолчанию (если в задании не требуется, чтобы диалоговое окно могло изменять размер) диалоговое окно должно иметь фиксированный размер.

1. Главное окно (фиксированного размера) содержит поле ввода TextBox с заголовком-меткой «Пароль» и кнопку Button «Открыть защищенное окно». Начальный пароль – «qwerty». Если пароль введен правильно, то при нажатии кнопки (или клавиши Enter) появляется модальное окно с заголовком «Защищенное окно», содержащее два поля ввода с общим заголовком «Новый пароль» (вначале эти поля содержат прежний пароль)
и две модальные кнопки «ОК» и «Отмена». Кнопка «ОК» доступна, если оба поля ввода в модальном окне содержат одинаковый непустой текст; при закрытии модального окна кнопкой «ОК» или клавишей Enter данный текст становится новым паролем. При закрытии модального окна кнопкой «Отмена» или клавишей Esc пароль не меняется. При вводе пароля вместо набранных символов должны отображаться символы «*» (звездочки); для этого следует использовать свойство PasswordChar компонента TextBox. При закрытии приложения сохранять пароль не требуется.

2. Главное окно (фиксированного размера) содержит кнопку Button «Изменить масштабирование» (кнопка размещена у левого верхнего угла окна). При нажатии кнопки появляется модальное окно с заголовком «Масштабирование», содержащее поле ввода с заголовком «Новый масштаб (%)» и три кнопки: «ОК», «Применить» и «Отмена». Кнопки «ОК» и «Применить» доступны, если поле ввода содержит целое число в диапа-

зоне от 10 до 300. Начальное значение данного поля равно 100; в это поле можно вводить только цифры. При нажатии кнопок «ОК» или «Применить» главное окно и содержащаяся на нем кнопка изменяют свои размеры в соответствии с новым масштабным множителем (например, если множитель равен 200, то размеры удваиваются); в случае нажатия кнопки «ОК» дополнительно происходит закрытие модального окна. Масштабирование всегда выполняется относительно *начальных* размеров окна. При нажатии кнопки «Отмена» модальное окно закрывается без выполнения масштабирования. При повторном открытии модального окна в поле ввода должно отображаться текущее значение масштабного множителя.

3. Главное окно (изменяемого размера) содержит многострочное поле ввода. При попытке закрыть окно появляется модальное окно с кнопками: «Сохранить», «Не сохранять», «Отмена». При нажатии кнопки «Сохранить» введенный текст сохранятся в файле text.txt. При нажатии кнопки «Отмена» закрытие главного окна отменяется. При повторном открытии окна, в случае наличия файла text.txt, его содержимое загружается в поле ввода.

4. Главное окно (изменяемого размера) содержит кнопку «Show». При нажатии на эту кнопку появляется модальное окно с двумя полями ввода, содержащими текущие значения координат левого верхнего угла кнопки «Show» (в первом поле выводится координата x, во втором – координата y), и кнопками «ОК», «Применить» и «Отмена». При нажатии кнопок «ОК» или «Применить» для кнопки «Show» устанавливаются новые координаты; в случае нажатия кнопки «ОК» дополнительно происходит закрытие модального окна. При нажатии кнопки «Отмена» модальное окно закрывается без выполнения дополнительных действий. Если поля ввода содержат недопустимые значения, то кнопки «ОК» и «Применить» должны быть недоступны (значение считается допустимым, если его можно преобразовать в неотрицательное число, причем после перемещения на указанную позицию кнопка «Show» будет хотя бы частично видна на экране). **Указание**. В главном окне используйте компонент Canvas.

5. Главное окно (фиксированного размера) содержит кнопку «Близнец». При нажатии на нее появляется модальное окно того же размера с кнопками «ОК» и «Отмена». Размеры модального окна можно изменять. Кнопка «ОК» (и клавиша Enter) закрывает модальное окно и устанавливает для главного окна размеры модального окна, кнопка «Отмена» (и клавиша Esc) – тоже закрывает, но при этом не изменяет размеры главного окна.

6. Главное окно (изменяемого размера) содержит кнопку «Добавить надпись». При нажатии на нее появляется модальное окно с двумя полями ввода для координат новой надписи, полем ввода для текста этой надписи и кнопками «ОК», «Применить» и «Отмена». При нажатии кнопок «ОК»

или «Применить» в указанной пользователем позиции главного окна создается новая метка Label; в случае нажатия кнопки «ОК» дополнительно происходит закрытие модального окна. При нажатии кнопки «Отмена» модальное окно закрывается без выполнения дополнительных действий. Если модальное окно было закрыто по нажатию кнопки «ОК», то при его повторном открытии координаты для новой надписи должны увеличиться на 20 единиц. **Указание**. В главном окне используйте компонент Canvas.

7. Главное окно (фиксированного размера) содержит кнопку «Бросить монету» и метки с отображением статистики «Суммарное количество бросков = » и «Процент орлов = ». При нажатии на кнопку «Бросить монету» появляется модальное окно с полем ввода количества бросков и кнопками «ОК», «Применить» и «Отмена». При нажатии кнопок «ОК» или «Применить» после смоделированных бросков монеты в главном окне обновляются метки статистики; в случае нажатия кнопки «ОК» дополнительно происходит закрытие модального окна. При нажатии кнопки «Отмена» модальное окно закрывается без выполнения дополнительных действий. Для имитации подбрасываний монеты используйте класс Random.

8. Главное окно (фиксированного размера) содержит три метки. При щелчке на одной из меток появляется модальное окно с полем ввода и кнопками «ОК», «Применить» и «Отмена». Поле ввода должно содержать старый текст метки. При нажатии кнопок «ОК» или «Применить» в главном окне обновляется содержимое метки; в случае нажатия кнопки «ОК» дополнительно происходит закрытие модального окна. При нажатии кнопки «Отмена» модальное окно закрывается без выполнения дополнительных действий. Если новый текст метки не умещается в главном окне, то ширина окна должна увеличиться.

9. Главное окно (фиксированного размера) содержит три поля ввода и метку (не кнопку!) «Изменить режим редактирования». В начале программы первое поле ввода имеет фокус, а все поля доступны для редактирования. При щелчке на метке появляется модальное окно с текстом «Вы уверены?» и кнопками «Да» и «Нет». При нажатии кнопки «Да» происходит изменение режима редактирования для поля ввода, имеющего фокус (обычный режим переводится в режим «Только для чтения» и наоборот). При нажатии на кнопку «Нет» режим редактирования не изменяется. В любом случае модальное окно закрывается. Модальное окно следует реализовать самостоятельно, не используя класс MessageBox.

10. Главное окно (фиксированного размера) содержит кнопку «Изменить фон». При нажатии на эту кнопку появляется модальное окно с тремя полями для ввода интенсивностей красной, зеленой и синей цветовых составляющих и кнопками «ОК», «Применить» и «Отмена». При нажатии кнопок «ОК» или «Применить» в главном окне изменяется цвет фона;

в случае нажатия кнопки «ОК» дополнительно происходит закрытие модального окна. При нажатии кнопки «Отмена» модальное окно закрывается без выполнения дополнительных действий. Если хотя бы одно из полей содержит текст, отличный от числа из диапазона 0–255, то кнопки «ОК» и «Применить» должны быть недоступны. **Указание**. Используйте структуру Color и ее метод FromRGB, а также класс SolidColorBrush.

11. Главное окно (переменного размера) содержит кнопку «Изменить выравнивание». При нажатии на эту кнопку появляется модальное окно с двумя выпадающими списками, имеющим надписи: «Горизонтальное выравнивание» и «Вертикальное выравнивание». Каждый список содержит три варианта: «По левому краю», «По центру», «По правому краю». Кроме того, модальное окно содержит кнопки «ОК», «Применить» и «Отмена». При нажатии кнопок «ОК» или «Применить» главное окно изменяет свое положение в соответствии со значениями выпадающих списков; в случае нажатия кнопки «ОК» дополнительно происходит закрытие модального окна. При нажатии кнопки «Отмена» модальное окно закрывается без выполнения дополнительных действий. При выравнивании необходимо учитывать *текущие* размеры главного окна. **Указание**. Используйте свойство SystemParameters.WorkArea.

20.2. Проект SYNC: синхронизация компонентов

Общие указания. Во всех заданиях требуется настраивать однотипные компоненты, задавая для каждой группы однотипных компонентов один метод-обработчик. Этот обработчик должен связываться с событием *родительского компонента* настраиваемой группы компонентов; связывание может выполняться в xaml-файле или с помощью метода AddHandler. Операторы if (типа if (n==1), if (n==2), if (n==3), ...) или switch в обработчиках использоваться не должны. Допустимо использовать свойства Tag компонентов, а также метод FindName, позволяющий обратиться к компоненту по его имени. По поводу совместного использования обработчиков событий см. проект CALC.

1. Главное окно содержит шесть полей ввода (компоненты TextBox) с текстом «1»–«6» и кнопку «Show». При нажатии кнопки «Show» появляется второе (немодальное) окно, содержащее шесть флажков (компоненты CheckBox). Флажки имеют подписи «1»–«6»; вначале ни один из них не установлен. При установке любого флажка текст соответствующего поля ввода выделяется полужирным шрифтом, при снятии флажка полужирное выделение соответствующего поля ввода отменяется. При изменении текста какого-либо поля ввода должна изменяться подпись соответствующего флажка.

2. Главное окно содержит шесть полей ввода (компоненты TextBox) с текстом «1»–«6» и кнопку «Show»; текст первого из полей ввода должен

быть выделен полужирным шрифтом. При нажатии кнопки «Show» появляется второе (немодальное) окно, содержащее шесть радиокнопок (компоненты RadioButton). Радиокнопки имеют подписи «1»–«6»; выбранной должна быть радиокнопка, соответствующая полю ввода с полужирным шрифтом. При выборе другой радиокнопки полужирное выделение автоматически переносится на текст соответствующего поля ввода. При изменении текста какого-либо поля ввода должна изменяться подпись соответствующей радиокнопки.

3. Главное окно содержит кнопку «Show», а также панель инструментов ToolBar с шестью кнопками ToggleButton. Кнопки на панели имеют заголовки «1»–«6», вначале ни одна из кнопок не является нажатой. При нажатии кнопки «Show» появляется второе (немодальное) окно, содержащее 6 флажков (компоненты CheckBox). Флажки имеют подписи «1»–«6»; вначале ни один из них не установлен. При установке/снятии любого флажка автоматически нажимается/освобождается соответствующая кнопка панели инструментов и наоборот, при нажатии/освобождении кнопки на панели инструментов автоматически устанавливается/снимается соответствующий флажок.

4. Главное окно содержит кнопку «Show», а также панель инструментов ToolBar с шестью кнопками ToggleButton. Кнопки на панели имеют заголовки «1»–«6», одна из кнопок является нажатой (вначале это кнопка «1»). При нажатии кнопки «Show» появляется второе (немодальное) окно, содержащее 6 радиокнопок (компоненты RadioButton). Радиокнопки имеют подписи «1»–«6»; выбранной должна быть радиокнопка, соответствующая нажатой кнопке ToggleButton главного окна. При выборе другой радиокнопки автоматически нажимается соответствующая кнопка ToggleButton (а ранее нажатая кнопка освобождается) и наоборот, при нажатии на кнопку ToggleButton автоматически выбирается соответствующая радиокнопка.

5. Главное окно содержит шесть красных меток с текстом «Color» и кнопку «Show». При нажатии кнопки «Show» появляется второе (немодальное) окно, содержащее шесть панелей с тремя радиокнопками каждая. Радиокнопки имеют подписи – «Red», «Green», «Blue» (подписи можно сделать общими для всех панелей, разместив их левее первой панели); вначале в каждой панели выбрана радиокнопка «Red». При переключении радиокнопок цвет текста соответствующей метки корректируется. При щелчке на любой метке главного окна ее цвет изменяется циклически (с красного на зеленый, с зеленого на синий, с синего на красный); при этом изменении должна автоматически выбираться соответствующая радиокнопка на той панели второго окна, которая связана с данной меткой.

6. Главное окно содержит семь полей ввода с текстом «0» и кнопку «Show». При нажатии кнопки «Show» появляется второе (немодальное) окно с семью ползунками (компонентами TrackBar). При перемещении ползунка должно автоматически изменяться число в соответствующем поле ввода. При указании другого числа в поле ввода должно автоматически измениться положение соответствующего ползунка. Границы изменения ползунков и значений полей ввода совпадают и лежат в диапазоне от 0 до 100. Синхронизацию ползунка с полем ввода следует проводить только в случае, когда в поле ввода указывается число из допустимого диапазона.

7. Главное окно содержит семь индикаторов прогресса (компоненты ProgressBar), кнопку «Default» и кнопку «Show». При нажатии кнопки «Show» появляется второе (немодальное) окно с семью ползунками (компоненты TrackBar). При перемещении ползунков должно автоматически изменяться наполнение соответствующих индикаторов ProgressBar. При щелчке на одном из компонентов ProgressBar соответствующий ползунок переходит из доступного состояния в недоступное и наоборот. При нажатии кнопки «Default» все индикаторы и ползунки возвращаются в исходное (нулевое) положение; доступность ползунков при этом не изменяется.

8. Главное окно содержит семь полей ввода с текстом «text1»–«text7» и кнопку «Show». При нажатии кнопки «Show» появляется второе (немодальное) окно с семью ползунками (компоненты TrackBar) и метками, содержащими тот же текст, что и поля ввода в главном окне. При перемещении ползунков должна автоматически изменяться ширина соответствующих полей ввода в главном окне. При изменении текста в полях ввода должен автоматически меняться текст соответствующих меток во втором окне.

9. Главное окно содержит семь ползунков (компоненты TrackBar), семь меток и кнопку «Show». Позиция ползунков по умолчанию – крайняя левая. При нажатии кнопки «Show» появляется второе (немодальное) окно с семью полями ввода. Содержимое полей ввода должно быть синхронизовано с текстом меток главного окна. При перемещении ползунков размер шрифта для меток меняется в соответствии с позицией ползунка в диапазоне от 10 до 30. При изменении текста в полях ввода должен изменяться текст в соответствующих метках.

10. Главное окно содержит семь полей ввода и кнопку «Show». При нажатии кнопки «Show» появляется второе (немодальное) окно с семью полями ввода, которые должны быть синхронизированы с соответствующими полями главного окна. Синхронизация должна выполняться при каждом изменении текста в любом поле ввода. Во втором окне также содержится кнопка «Остановить синхронизацию». При нажатии на нее синхронизация прекращается, а название кнопки заменяется на «Возобновить

синхронизацию». При нажатии кнопки еще раз синхронизация возобновляется, причем для синхронизации используется текст, указанный в полях ввода главного окна. При закрытии и повторном открытии второго окна состояние кнопки «Остановить синхронизацию» не изменяется.

11. Главное окно содержит семь полей ввода и кнопку «Show». При нажатии кнопки «Show» появляется второе (немодальное) окно с семью метками, содержимое которых совпадает с содержимым полей ввода. При щелчке на любой метке соответствующее поле ввода изменяет фон с белого на серый и наоборот. При изменении текста в поле ввода с белым фоном это изменение сразу проявляется в соответствующей метке, при изменении текста в поле ввода с серым фоном метка не изменяется. Однако при изменении фона поля ввода с серого на белый немедленно выполняется синхронизация текста поля ввода и соответствующей метки.

12. Главное окно содержит семь полей ввода текста и кнопку Button с заголовком «Show». При нажатии кнопки «Show» появляется второе (немодальное) окно с семью ползунками (компоненты TrackBar), позиция которых соответствует длине текста в соответствующем поле ввода. При редактировании текста автоматически меняется позиция ползунка, а при изменении позиции ползунка текст в поле ввода сокращается или удлиняется (удлинение текста выполняется за счет добавления символов «*»). Ползунки могут принимать значения от 0 до 25; в поля ввода нельзя ввести текст, длина которого превышает 25 символов.

20.3. Проект DRAGDROP: режим Drag & Drop

Общие указания. Окно приложения должно допускать возможность изменения размеров, причем размер и положение компонентов окна (за исключением высоты однострочных полей) должны соответствующим образом корректироваться (используйте компонент Grid со строками и столбцами, изменяющими свой размер в зависимости от размера окна). Режим Drag & Drop подробно обсуждался в проекте ZOO; кроме того, он использовался в проектах LISTBOXES и HTOWERS. Работа с меню рассматривалась в проекте TEXTEDIT, версии 1 и 2.

1. Окно содержит три многострочных поля ввода. Реализуйте возможность перетаскивания на них текстовых файлов из Проводника и автоматическую загрузку содержимого этих файлов в соответствующие поля (обрабатываться должны только файлы с расширением txt). Кроме того, требуется предусмотреть возможность перетаскивания *непустого* содержимого одного поля ввода на другое (при нажатой клавише Ctrl). При любом варианте перетаскивания новое содержимое должно добавляться к прежнему содержимому поля ввода. Меню окна содержит одно подменю «Command» с командами «Clear» и «Exit». Команда «Clear» удаляет текст из того поля ввода, которое имеет фокус.

2. Окно содержит три многострочных поля ввода, над каждым из которых указывается метка (вначале метки содержат текст «<No file>»). Реализуйте возможность перетаскивания текстовых файлов *на метки* и автоматическую загрузку содержимого этих файлов в соответствующие поля (обрабатываться должны только файлы с расширением txt); при этом в соответствующей метке должно отображаться полное имя файла. Перетащить файл можно только на метку с текстом «<No file>». Меню окна содержит одно подменю «Command» с командами «Save», «Clear» и «Exit». Команды «Save» и «Clear» действуют на то поле ввода, которое имеет фокус; при этом команда «Save» сохраняет новое содержимое поля ввода в том же файле, а команда «Clear» очищает поле ввода вместе со связанной с ним меткой (на метке при этом опять отображается текст «<No file>»).

3. Окно содержит четыре «обычные» метки с буквами нуклеотидов A, C, T, G и три «широкие» метки, каждая из которых имеет рамку и содержит генные последовательности (вначале широкие метки являются пустыми; их ширина не зависит от размеров текста и определяется шириной окна). Реализуйте перетаскивание символов-нуклеотидов из обычных меток на широкие; при этом нуклеотиды добавляются в конец текста широкой метки. Кроме того, реализуйте перетаскивание мышью текста из одной широкой метки в другую, в результате чего в конец текста метки-приемника добавляется весь текст метки-источника. Меню окна содержит одно подменю «Command» с командами «Clear 1», «Clear 2», «Clear 3» и «Exit». Команды «Clear» очищают широкую метку с указанным номером. При реализации этих команд следует использовать единственный общий обработчик.

4. Окно содержит шесть меток Label с заголовками «label1»–«label6» и шесть полей ввода TextBox с текстом «textBox1»–«textBox6». При перетаскивании мышью поля ввода на метку (при нажатой клавише Ctrl) заголовок метки меняется на содержимое поля ввода, причем изменяется как содержимое, так и шрифт (положение поля ввода не меняется). Перемещать поле ввода на метку можно несколько раз. Меню окна содержит одно подменю «Command» с командами «Bold», «Italic», и «Exit». Команды «Bold» и «Italic» действуют как переключатели, устанавливая или отменяя режим полужирного и курсивного шрифта для поля ввода, имеющего фокус.

5. Окно содержит шесть кнопок Button с заголовками «button1»–«button6» и шесть пустых списков ListBox. При перетаскивании мышью кнопки на список в *указанную* позицию списка добавляется заголовок этой кнопки (положение кнопки не меняется). Перемещать кнопку на список можно несколько раз. Меню окна содержит одно подменю «Command» с командами «Clear» и «Exit». Команда «Clear» очищает содержимое спи-

ска, имеющего фокус; если фокус имеет кнопка, то никаких действий не выполняется. **Указание**. В WPF отсутствует стандартный метод IndexFromPoint, позволяющий по позиции мыши в списке определить номер пункта; вариант реализации такого метода приведен в проекте LISTBOXES, п. 15.4.

6. Окно содержит шесть радиокнопок RadioButton с заголовками «color1»–«color6» (заголовки имеют различные цвета) и шесть прямоугольников (компоненты Rectangle) с белым фоном. При перетаскивании мышью радиокнопки на прямоугольник цвет фона прямоугольника изменяется в соответствии с цветом заголовка перетаскиваемой радиокнопки (положение радиокнопки не меняется). Меню окна содержит одно подменю «Command» с командами «Color» и «Exit». Команда «Color» приводит к появлению диалогового окна ColorDialog, с помощью которого можно изменить цвет заголовка выбранной радиокнопки. **Указание**. К проекту необходимо подключить компонент ColorDialog из библиотеки Windows Forms (детали описаны в проекте TEXTEDIT версии 2, п. 10.3).

7. Окно содержит компонент Canvas и две кнопки – «New» и «Trash». Нажатие на кнопку «New» ведет к появлению в случайном месте компонента Canvas новой метки (компонента TextBlock) со случайной заглавной латинской буквой. Реализуйте возможность перетаскивания появившихся меток на новые позиции компонента Canvas. При перетаскивании метки на кнопку «Trash» метка удаляется. При перетаскивании одной метки на другую метка-источник удаляется, а к тексту метки-приемника добавляется текст метки-источника. Меню окна содержит одно подменю «Command» с командами «Clear» и «Exit». Команда «Clear» удаляет все метки. Все метки удаляются также при нажатии на кнопку «Trash». **Указание**. По поводу создания компонентов во время работы приложения см. проект HTOWERS.

8. Окно содержит четыре панели (компоненты Canvas) и группу из четырех радиокнопок для выбора текущей панели. На панелях в случайных местах располагаются по 3 метки. Реализуйте возможность перетаскивания меток с текущей панели на любую другую (кроме того, для текущей панели можно перетаскивать метки в пределах этой панели). Метки, расположенные на других панелях, перетаскивать нельзя. Меню окна содержит одно подменю «Command» с командой «Color» и «Exit». Команда «Color» приводит к появлению диалогового окна ColorDialog, с помощью которого можно изменить цвет всех меток активной панели. **Указание**. К проекту необходимо подключить компонент ColorDialog из библиотеки Windows Forms (детали описаны в проекте TEXTEDIT версии 2, п. 10.3). По поводу перетаскивания компонентов между групповыми компонентами см. проект HTOWERS.

9. Реализуйте приложение для тестирования перетаскивания. Окно содержит четыре многострочных поля ввода, метку с текстом «Label» и метку с текстом «String». Любую из меток, а также любые объекты из других программ, в частности, Проводника, можно перетащить на любое *незаполненное* поле ввода. В результате в поле ввода выводится подробная информация об объекте перетаскивания (в том числе о доступных форматах, которые можно определить с помощью метода GetFormats). При перетаскивании метки «Label» объектом перетаскивания является сама метка, а при перетаскивании метки «String» объектом перетаскивания является строка с ее названием. Меню окна содержит подменю «Command» с командами «Clear» и «Exit». Команда «Clear» очищает поле ввода, имеющее фокус.

10. Окно содержит список, в который пользователь может добавлять имена файлов, перетаскивая их из Проводника (полное имя файла добавляется в конец списка). Кроме того, окно содержит метку «Trash». При перетаскивании списка на метку (при нажатой клавише Ctrl) выполняется удаление текущего элемента списка (если список пустой, то перетаскивание запрещено, т. е. курсор перетаскивания имеет запрещающий вид). Позиция списка при перетаскивании не изменяется. Меню окна содержит подменю «Command» с командами «Clear» и «Exit». Команда «Clear» очищает список.

20.4. Проект TIMER: программы, управляемые таймером

Общее указание. Работа с секундомером рассматривается в проекте CLOCK, п. 4.2.

1. Окно содержит выпадающий список ComboBox с комментариями (вначале в нем имеется только комментарий «–»), пустой список ListBox с меткой «Результаты», табло секундомера – метку с текстом «0:0», а также кнопку Button с заголовком «Старт», используемую для запуска секундомера (при запуске секундомера заголовок кнопки меняется на «Стоп»). При остановке секундомера полученный результат вместе с текущим комментарием добавляется в конец списка «Результаты», а табло секундомера очищается. Для добавления нового комментария достаточно ввести его в поле списка ComboBox и нажать Enter. Секундомер регистрирует время с точностью до десятых долей секунды. Меню окна содержит подменю «Command» с пунктами «Clear» и «Exit». Команда «Clear» приводит к очистке списка «Результаты».

2. Окно содержит метку, состоящую из одного символа – строчной русской буквы «а», и два поля ввода (только для чтения) с заголовками-метками «Время» и «Очки». В верхней части окна располагается панель инструментов с четырьмя кнопками: «Старт», «10 с», «30 с», «60 с», причем последние три образуют группу, обязательно содержащую нажатую

кнопку (вначале это «10 с»). Кроме того, окно содержит список ListBox с заголовком-меткой «Лучшие результаты», в котором указываются 10 лучших результатов для выбранного времени (упорядоченных по возрастанию длительности). При нажатии кнопки «Старт» в строке «Время» начинается обратный отсчет времени (в десятых долях секунды), строка «Очки» обнуляется, а символ-буква в метке изменяется. Требуется нажать клавишу с указанной буквой. При правильном нажатии клавиши счетчик в строке «Очки» увеличивается на 1, а буква в метке снова изменяется. При неправильном нажатии клавиши счетчик «Очки» уменьшается на 1. Продолжительность одного тренировочного сеанса (в секундах) определяется нажатой кнопкой быстрого доступа. После завершения тренировочного сеанса при необходимости корректируется список «Лучшие результаты»; если полученный результат заносится в данный список, то об этом сообщается во вспомогательном информационном окне (с помощью функции MessageBox.Show). Списки лучших результатов для каждого режима хранятся в файлах «res10.dat», «res30.dat», «res60.dat» и считываются из них при смене режима и при запуске программы.

3. Окно содержит метку и два поля ввода (только для чтения) с заголовками-метками «Время» и «Очки». В верхней части окна располагается панель инструментов с пятью кнопками: «Старт», «+», «–», «*», «/», причем последние четыре образуют группу, обязательно содержащую нажатую кнопку (вначале это «+»). Кроме того, окно содержит список ListBox с заголовком-меткой «Лучшие результаты», где указываются 10 лучших результатов для выбранного режима, и панель с пятью компонентами RadioButton, имеющими пустые подписи. При нажатии кнопки «Старт» в строке «Время» начинается обратный отсчет времени (в десятых долях секунды), строка «Очки» обнуляется, а в метке появляется числовое выражение с выбранной операцией (например, «34 + 78 =»). В группе радиокнопок выводятся 5 вариантов ответа. Требуется щелкнуть мышью на радиокнопке с правильным вариантом. При правильном ответе счетчик в строке «Очки» увеличивается на 1, при неправильном – уменьшается на 1. В любом случае в метке появляется новое выражение. Продолжительность одного тренировочного сеанса – 30 секунд. После завершения тренировочного сеанса при необходимости корректируется список лучших результатов; если полученный результат заносится в данный список, то об этом сообщается во вспомогательном информационном окне (с помощью функции MessageBox.Show). Лучшие результаты для каждого режима хранятся в файлах «add.dat», «sub.dat», «mult.dat», «div.dat» и считываются из них при смене режима и при запуске программы. Для режимов «+» и «–» в выражениях должны использоваться числа от 1 до 100, для режима «*» – от 1 до 10, для режима «/» первый операнд должен быть двузначным, второй – однозначным, а результат должен быть целым числом.

4. Окно содержит метку с текстом, отображающим текущее системное время компьютера в формате «чч:мм», и три компонента-флажка CheckBox, каждый из которых связан с отдельным будильником. Текст рядом с флажком указывает время срабатывания будильника и также имеет формат «чч:мм». Срабатывание происходит, если к указанному моменту будильник является включенным (т. е. связанный с ним флажок содержит «галочку»); при этом в течение 10 с воспроизводится ранее выбранный звуковой сигнал, а флажок переходит в промежуточное (третье) состояние. Для досрочного отключения сигнала достаточно снять «галочку» с соответствующего флажка, переведя его тем самым в отключенное состояние. Если досрочное отключение не выполнено, то флажок автоматически переходит в отключенное состояние через 10 с. При установке любого флажка во включенное состояние должно отображаться диалоговое окно с двумя выпадающими списками ComboBox, предназначенными для задания нового времени (часов и минут) срабатывания будильника; по умолчанию устанавливается время, ранее связанное с данным будильником. Кроме того, в диалоговом окне можно выбрать, с помощью списка ListBox, один из предустановленных вариантов сигнала, который начнет воспроизводиться в момент срабатывания будильника (сигналы хранятся в виде набора wav-файлов). Требуется также предусмотреть сохранение текущих настроек приложения в текстовом файле и их последующее восстановление при повторном запуске приложения. **Указание**. Для воспроизведения wav-файлов используйте класс System.Media.SoundPlayer.

5. Окно содержит поле ввода (только для чтения) с заголовком-меткой «Время» и начальным значением «0,0», пустой список ListBox с заголовком-меткой «Результаты», а также панель (компонент Canvas), на которой в произвольном порядке расположены 6 меток квадратного размера с заголовками «1»–«6». Меню окна содержит подменю «Command» с командами «Start» и «Exit». При выборе команды «Start» все метки изменяют свое положение на панели и заливаются красным цветом, а в строке «Время» начинается отсчет времени с точностью до десятых долей секунды. Требуется быстро нажать на все метки в указанном порядке. Нажатие на правильную метку делает ее фон зеленым. Как только все 6 меток окажутся нажатыми, отсчет времени прекращается. Если при этом в строке «Время» содержится значение, меньшее «10,0» (менее 10 с), то выводится сообщение «Вы выиграли» и полученный результат заносится в начальную строку списка результатов; в противном случае выводится сообщение «Вы проиграли». Очередной выбор команды «Start» приводит к случайному изменению расположения меток на компоненте Canvas, обнулению строки «Время» и началу нового отсчета. Команда «Start» должна быть доступна только при остановленной игре. Список результатов должен храниться в файле на диске и считываться из этого файла при каждом запуске программы.

При изменении расположения меток на панели необходимо, чтобы никакие из двух меток не пересекались.

6. Окно содержит два поля ввода (только для чтения) с метками «Время» и «Добыча» и панель (компонент Canvas). Кроме того, окно содержит список ListBox с заголовком-меткой «5 лучших результатов». Вначале в строке «Время» содержится число «30», а в строке «Добыча» – «0». Меню окна содержит одно подменю «Command» с командами «Start» и «Exit». При выполнении команды «Start» начинается обратный отсчет времени в строке «Время» (от 30 до 0 в секундах) и на панели в случайном месте появляется метка с текстом «50» (размеры метки – 10 × 10 аппаратно-независимых единиц, внешние и внутренние поля равны 0), причем через каждую десятую долю секунды число на метке уменьшается на 1. При щелчке мышью на метке число в поле ввода «Добыча» увеличивается на значение метки (или уменьшается, если текст метки изображает отрицательное число), а метка отображается в другом месте панели (вновь с заголовком «50»). При щелчке мыши вне метки из поля «Добыча» вычитается число 10. Через 30 секунд игра заканчивается. Если в строке «Добыча» содержится положительное число, то выводится сообщение «Вы выиграли», в противном случае выводится сообщение «Вы проиграли»; кроме того, при необходимости корректируется список лучших результатов. При запущенной игре команда «Start» должна быть недоступна. Список лучших результатов должен храниться в файле на диске и считываться из этого файла при каждом запуске программы.

7. Окно содержит панель (компонент Canvas), кнопку с заголовком «Старт», три поля ввода (только для чтения) с метками «Точность», «Время» и «Результат» (вначале поля содержат нули) и список ListBox с заголовком-меткой «5 лучших результатов». При нажатии на кнопку «Старт» ее заголовок изменяется на «Стоп», начинается отсчет времени в строке «Время» (с точностью до десятых долей секунды), а в одном из углов панели изображается метка квадратного размера без заголовка (угол панели выбирается случайным образом). Требуется переместить эту метку мышью точно в центр панели и нажать кнопку «Стоп» (при перемещении метка должна следовать за мышью). После этого в строке «Точность» выводятся два числа: отклонение от верной позиции по горизонтали и по вертикали, а в строке «Результат» выводится вещественное число, вычисленное следующим образом: к полученному времени прибавляется 1, и это число умножается на сумму модулей отклонений. Если число в строке «Результат» меньше 50, то выводится сообщение «Вы выиграли», в противном случае выводится сообщение «Вы проиграли»; кроме того, при необходимости корректируется список лучших результатов, который должен храниться

в файле на диске и считываться из этого файла при каждом запуске программы.

8. Окно содержит два поля ввода (только для чтения) с метками-заголовками «Ракеты» и «Время до взрыва», панель (компонент Canvas) и односимвольную метку, изображающую самолет (шрифт Wingdings, символ «Q») и находящуюся в левом верхнем углу панели. Курсор мыши на панели имеет вид перекрестия. Меню окна содержит подменю «Command» с командами «Start» и «Exit». При выполнении команды «Start» метка самолета начинает прямолинейное движение на панели (приращения свойств метки Left и Top должны лежать в диапазоне 1–3; они определяются случайным образом перед началом движения и производятся каждую десятую долю секунды), а в строке «Ракеты» появляется число «4». Щелчок мышью на панели означает пуск ракеты, которая взрывается через 2 с (в строке «Время до взрыва» начинается отсчет времени в десятых долях секунды – от 2,0 до 0,0); точку пуска ракеты на экране следует пометить красным кругом (компонент Ellipse) радиуса 2 аппаратно-независимых единицы. В момент взрыва радиус круга увеличивается до 15 единиц (зона поражения). Если в момент взрыва самолет окажется в зоне поражения (т. е. расстояние от центра метки-самолета до точки пуска ракеты будет не более 15 единиц), то появляется сообщение «Самолет сбит, вы выиграли», и игра заканчивается. В противном случае ничего не происходит. Запускать новую ракету до взрыва ранее запущенной нельзя. Если все ракеты израсходованы безрезультатно или самолет достиг границы панели, то выводится сообщение «Вы проиграли». Команда «Start» должна быть доступна только при остановленной игре.

20.5. Проект REGISTRY: диалоги и работа с реестром

Общие указания. Работа с реестром описывается в проекте IMGVIEW. При сохранении в реестре размеров окна и других характеристик приложения вещественного типа следует отбрасывать их дробные части и сохранять в реестре полученные целочисленные значения (как это сделано в проекте IMGVIEW). Работа с диалоговым окном OpenFileDialog описана в проекте TEXTEDIT, версия 1. Для организации диалогов, связанных с выбором шрифта и цвета, следует использовать компоненты FontDialog и ColorDialog из библиотеки Windows Forms (пример работы с компонентом ColorDialog приводится в проекте TEXTEDIT, версия 2).

1. Окно содержит компонент Grid из двух столбцов одинаковой ширины. В каждом столбце размещается метка и многострочное поле ввода TextBox. Вначале фокус имеет левое поле ввода; нажатие клавиши Tab позволяет переключаться между компонентами TextBox. При изменении ширины окна пропорционально увеличиваются размеры компонентов TextBox (оба столбца всегда имеют одинаковую ширину). Меню окна

содержит подменю «File» с командами «Open...», «Save», «Compare» и «Exit». Команда «Open...» выводит диалоговое окно OpenFileDialog; с его помощью можно выбирать не только существующие текстовые файлы, но и создавать новые. При выборе требуемого файла его содержимое загружается в активное поле ввода, а полное имя файла отображается в метке, расположенной над этим полем ввода. Команда «Compare» доступна, если в обоих компонентах TextBox содержатся загруженные данные; она сравнивает содержимое левого и правого поля ввода и устанавливает курсор перед первым отличающимся символом в *левом* поле ввода (при нажатии Tab курсор должен переходить на первый отличающийся символ в *правом* поле ввода). Если содержимое поля ввода изменено пользователем, то перед именем файла в метке указывается символ *. Команда «Save» сохраняет в файле содержимое того поля ввода, в котором отображается курсор; при этом символ * в метке исчезает. При завершении работы программы имена файлов сохраняются в реестре, а при последующем запуске программы считываются из реестра (если запись существует); восстанавливаются также размеры окна и позиции курсора в каждом компоненте TextBox. При попытке закрыть программу, не сохранив измененное содержимое файлов, выводится стандартное диалоговое окно с запросом о сохранении измененного файла и вариантами ответа «Да», «Нет», «Отмена». Если требуется сохранить два файла, то последовательно выводятся два диалоговых окна (за исключением случая, когда в первом диалоговом окне был выбран вариант «Отмена»).

2. Окно содержит многострочное поле ввода (компонент TextBox), размеры которого автоматически меняются при изменении размеров окна, выпадающий список вариантов конвертации систем счисления («10→2» – вариант выбран по умолчанию, «10→16», «2→10», «16→10», «16→2», «2→16») и кнопку «Конвертировать». Во время первого запуска программы в начале ее работы поле ввода недоступно для редактирования. Меню окна содержит одно подменю «File» с командами «Open...», «Save» и «Exit». Команда «Open...» выводит диалоговое окно OpenFileDialog; с его помощью пользователь открывает текстовый файл, содержимое которого загружается в поле ввода (имя загруженного файла отображается в заголовке окна); при этом поле ввода становится доступным для редактирования. Кнопка «Конвертация» переводит выделенное пользователем в поле ввода число из одной системы счисления в другую (выделенное число заменяется на полученное число, причем полученное число остается выделенным). Если ничего не выделено или выделение содержит некорректные данные, то ничего не происходит. Если текст загруженного файла изменен, то в заголовке окна после имени файла отображается символ *. Команда «Save» сохраняет содержимое поля ввода; при этом символ * из заголовка удаляется. При завершении работы программы имя текущего открытого

файла сохраняется в реестре, а при последующем запуске программы считывается из реестра (если запись существует); восстанавливаются также последний вариант конвертации, размер и положение окна и позиция курсора в поле ввода. При попытке закрыть программу, не сохранив измененное содержимое файла, выводится стандартное диалоговое окно с запросом о сохранении измененного файла и вариантами ответа «Да», «Нет», «Отмена».

3. Окно содержит список (компонент ListBox) с заголовком-меткой «Playlist», кнопки «Play/Stop», «Up» и «Down», поле ввода с подписью-меткой «Time» и меню «Command» с командами «Add file», «Add folder», «Clear» и «Exit». Размеры списка автоматически меняются при изменении размеров окна. При пустом списке кнопки неактивны. Команды «Add file» и «Add folder» приводят к появлению диалогового окна OpenFileDialog с помощью которого пользователь выбирает один из wav-файлов; команда «Add file» добавляет в список выбранный файл, команда «Add folder» – все wav-файлы из папки (файлы добавляются в конец списка). Последний добавленный в список элемент становится текущим. После добавления в список хотя бы одного элемента кнопка «Play/Stop» становится доступной; если список содержит более одного элемента, то доступными становятся кнопки «Up» и «Down». Поле ввода «Time» позволяет указать время воспроизведения каждого файла (в секундах); по умолчанию время равно 10 с (если продолжительность файла меньше указанного времени, то файл воспроизводится циклически). Кнопки «Up» и «Down» позволяют перемещать текущий элемент списка вверх или вниз. Когда время воспроизведения заканчивается, автоматически начинает воспроизводиться файл, расположенный в списке после текущего (при этом данный элемент списка становится текущим). Файлы воспроизводятся циклически. При завершении работы программы список воспроизведения сохраняется в реестре. При последующих запусках программа должна восстанавливать сохраненное состояние (в том числе список файлов, позицию текущего элемента, время воспроизведения и размеры окна).

4. Окно содержит поле ввода с заголовком-меткой «Код символа» и панель GroupBox с меткой, содержащей один символ размером 100 аппаратно-независимых единиц (код этого символа указывается в поле ввода). В начале работы программа настроена на шрифт Wingdings; название шрифта указывается в заголовке компонента GroupBox. Значения кодов символов в поле ввода можно изменять в диапазоне от 33 до 255; если поле ввода содержит другой текст, то метка не отображается. Меню окна содержит одно подменю «Command» с командами «Font...» и «Exit». Команда «Font...» приводит к появлению диалогового окна FontDialog, позволяющего изменить название просматриваемого шрифта (прочие изменения, сделанные в окне FontDialog, не учитываются). Размер шрифта дол-

жен изменяться автоматически при изменении размеров окна; при этом начальные размеры окна нельзя уменьшать. Отображаемый символ должен быть отцентрирован по вертикали и горизонтали относительно границы панели GroupBox. При завершении работы программы текст в поле ввода, название шрифта и размеры окна сохраняются в реестре. При последующих запусках программа должна восстанавливать сохраненное состояние.

5. Окно содержит поле ввода с заголовком-меткой «Номер цвета» и панель GroupBox с меткой. При изменении размера окна пропорционально увеличивается размер панели. При вводе в поле названия одного из стандартных цветов, входящих в класс Colors, панель GroupBox закрашивается этим цветом, а название цвета выводится в заголовке панели инверсным цветом (регистр символов во введенном тексте не учитывается; если введенный текст не соответствует ни одному из стандартных цветов, то фон панели GroupBox не изменяется). Меню окна содержит одно подменю «Command» с командами «Color...» и «Exit». Команда «Color...» приводит к появлению диалогового окна ColorDialog, позволяющего выбрать цвет для фона панели GroupBox. Если выбранный цвет является одним из стандартных цветов, то панель закрашивается этим цветом, а в заголовке панели указывается его название; если выбранный цвет не является стандартным цветом, то об этом выводится сообщение в стандартном информационном окне, а фон панели не изменяется. При завершении работы программы текст поля ввода, текущий цвет фона панели, ее заголовок, а также размеры окна сохраняются в реестре. При последующих запусках программа должна восстанавливать сохраненные данные. **Указание**. Использование класса Colors описывается в проекте COLORS.

6. Окно содержит пустое многострочное поле (компонент TextBox) с серым фоном, недоступное для редактирования, кнопку «Открыть» и 3 компонента-ползунка TrackBar, каждый из которых может принимать 10 значений – от 0 до 9. Ползунки ориентированы вертикально и располагаются в левой части окна. Над каждым ползунком отображается метка, содержащая текущее значение этого ползунка (число от «0» до «9»). Кнопка «Открыть» находится под ползунками, многострочный текстовый компонент занимает оставшуюся (правую) часть окна по всей его высоте. При изменении размеров окна должна изменяться ширина и высота многострочного текстового компонента, а также высота ползунков; минимальные допустимые размеры окна надо настроить таким образом, чтобы они обеспечивали отображение всех ее компонентов. При правильной настройке с помощью ползунков трехзначного кода замка и нажатии кнопки «Открыть» (или клавиши Enter) в компонент TextBox загружается содержимое записной книжки из файла notebook.txt (если этот файл существует), фон компонента TextBox становится белым, а заголовок кнопки изменяется на «Закрыть» (при неправильном вводе кода вид компонента TextBox

и кнопки не меняется). Правильный код хранится в реестре; если запись в реестре отсутствует, то код равен «000». При открытой записной книжке можно редактировать ее содержимое, а также устанавливать новый код с помощью ползунков. При нажатии кнопки «Закрыть» текст сохраняется в файле notebook.txt, код – в реестре, компонент TextBox очищается, его фон становится серым, заголовок кнопки меняется на «Открыть», а значения ползунков изменяются случайным образом. При завершении работы программы в реестре дополнительно сохраняются размеры окна, которые восстанавливаются при ее последующем запуске.

7. Окно содержит компонент Grid с тремя столбцами, причем средний столбец отводится под разделитель (компонент GridSplitter). В первом столбце окна размещается список ListBox, в третьем – многострочное поле ввода (компонент TextBox). При изменении ширины окна увеличивается ширина компонента TextBox; границу между списком и полем ввода можно изменять путем перетаскивания разделителя. Клавиша Tab позволяет переключаться между списком и полем ввода. Меню окна содержит одно подменю «File» с командами «Open...», «Close» и «Exit». Команда «Open...» выводит диалоговое окно OpenFileDialog; с его помощью можно выбирать не только существующие текстовые файлы, но и создавать новые. При выборе требуемого файла его полное имя добавляется в конец списка ListBox (становясь текущим), а его содержимое загружается в компонент TextBox. В дальнейшем для загрузки данного файла в поле ввода достаточно выбрать его имя в списке ListBox. Если при выполнении команды «Open...» было выбрано имя, уже имеющееся в списке, то это имя в списке делается текущим. Когда элемент списка теряет выделение, связанный с ним текст автоматически сохраняется в соответствующем файле. Команда «Close» удаляет имя текущего файла из списка файлов; при этом также происходит автоматическое сохранение текста в данном файле. При удалении элемента списка выделяется следующий за ним элемент, а при его отсутствии – предыдущий элемент. Если список ListBox является пустым, то компонент TextBox недоступен для редактирования; также недоступной в этой ситуации является команда меню «Close». При завершении работы программы содержимое списка файлов сохраняется в реестре, а при последующем запуске программы считывается из реестра; сохраняется и восстанавливается также номер текущего элемента списка, позиция курсора в компоненте TextBox, размер окна и позиция разделителя.

Литература

1. *Абрамян М. Э.* Программирование в Delphi на примерах. 2-е изд., доп. – Ростов-на-Дону: Изд-во ЦВВР, 2003. – 75 с.

2. *Абрамян М. Э.* Delphi 7. Карманный справочник с примерами. – М.: Изд-во «КУДИЦ-ОБРАЗ», 2006. – 288 с.

3. *Абрамян М. Э.* Visual C# на примерах. – СПб.: Изд-во БХВ-Петербург, 2008. – 482 с.

4. *Абрамян М. Э.* Платформа .NET: Основные типы стандартной библиотеки. Работа с массивами, строками, файлами. Объекты, интерфейсы, обобщения. Технология LINQ. – Ростов-на-Дону: Изд-во ЮФУ, 2014. – 218 с.

5. *Албахари Дж., Албахари Б.* C# 6.0. Справочник. Полное описание языка. 6-е изд. – М.: «И. Д. Вильямс», 2016. – 1040 с.

6. *Грэхэм Р., Кнут Д., Паташник О.* Конкретная математика. Основание информатики. – М.: Мир, 1998.

7. *Петцольд Ч.* Microsoft Windows Presentation Foundation. Базовый курс. – М.: Изд-во «Русская Редакция»; СПб.: Питер, 2008. – 944 с.

8. *Мак-Дональд М.* WPF 4: Windows Presentation Foundation в .NET 4.0 с примерами на C# 2010 для профессионалов. – М.: «И. Д. Вильямс», 2011. – 1024 с.

9. *Натан А.* WPF 4. Подробное руководство. – СПб.: Символ-Плюс, 2011. – 880 с.

Указатель

В указателе приведены все компоненты, вспомогательные классы, свойства, события и методы библиотеки WPF, которые описываются в комментариях к какому-либо этапу разработки проекта. Кроме того, указатель содержит ссылки на комментарии, где рассматриваются основные концепции библиотеки WPF. В группе «Проекты» перечислены все проекты, представленные в книге.

Учебное издание

АБРАМЯН Анна Владимировна,
АБРАМЯН Михаил Эдуардович

Разработка пользовательского интерфейса на основе технологии Windows Presentation Foundation

Подписано в печать 26.10.2017.
Формат 60×84 1/16. Усл. печ. л. 17,44. Уч.-изд. л. 13,48.
Бумага офсетная. Тираж 40 экз. Заказ № 5997.

Издательство Южного федерального университета

Отпечатано в отделе полиграфической, корпоративной и сувенирной продукции
Издательско-полиграфического комплекса КИБИ МЕДИА ЦЕНТРА ЮФУ
344090, г. Ростов-на-Дону, пр. Стачки, 200/1. Тел. (863) 247-80-51.