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

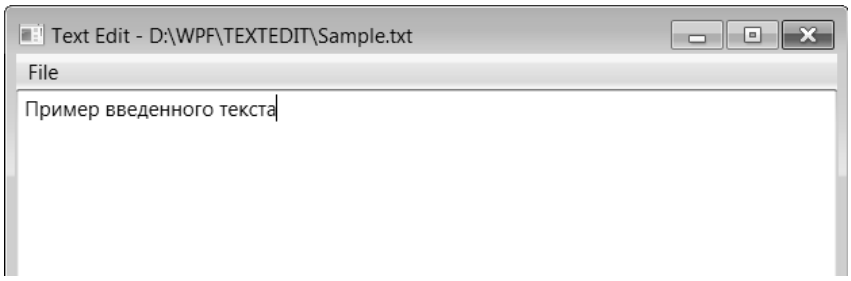

Рис. 1. Верхняя часть окна приложения TEXTEDIT версии 1

1.1. Создание меню

Рис. 2. 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.

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.

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.

1.2. 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.

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.

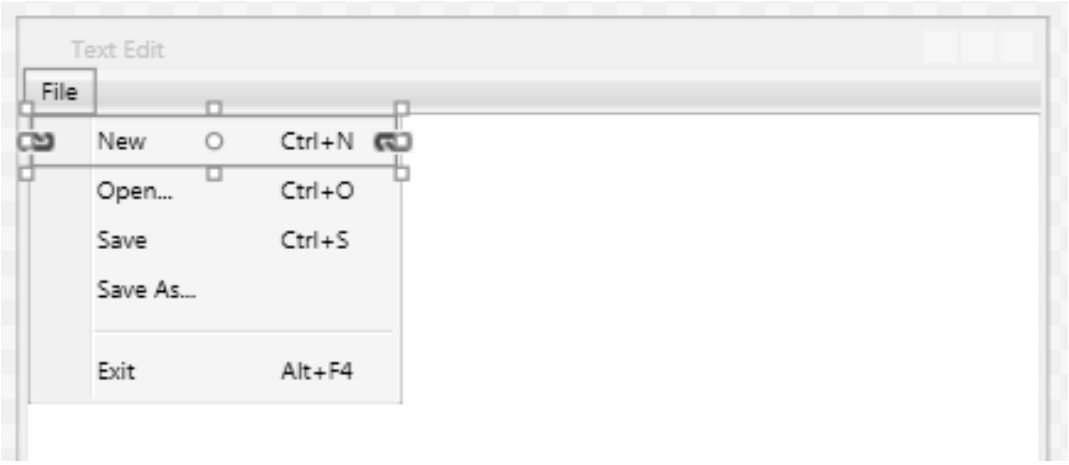

Рис. 3. Макет окна приложения TEXTEDIT версии 1 (второй вариант)

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.

1.3. 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.

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.

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.

1.4. 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.

1.5. 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.

1.6. 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).

2. Цвета: COLORS

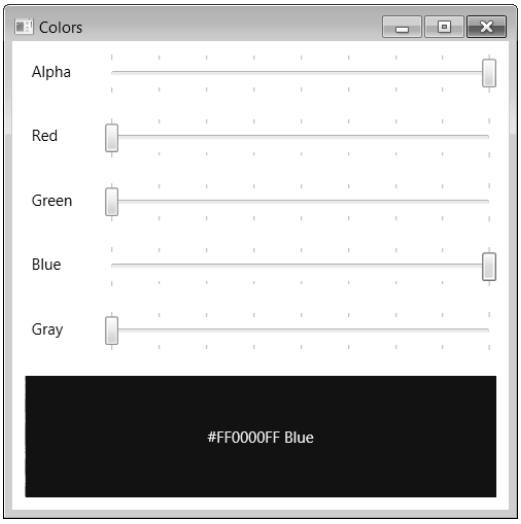

Рис. 4. Окно приложения COLORS

2.1. Начальная настройка макета окна

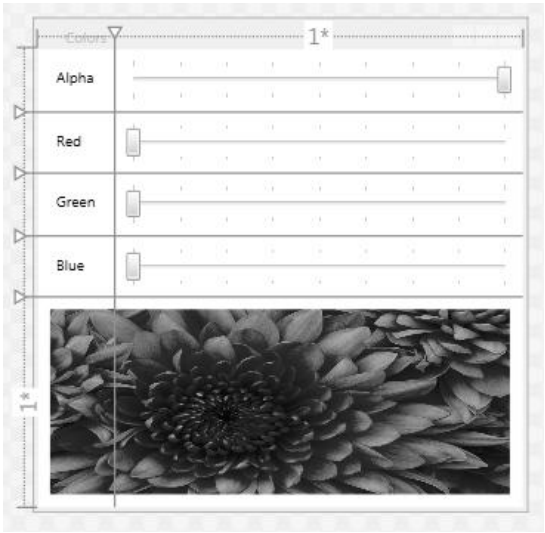

Рис. 5. Макет окна приложения COLORS

```
<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.

2.2. 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.

2.3. 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.

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.

2.4. 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.

2.5. 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».

2.6. 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.

3. Флажки и группы флажков: CHECKBOXES

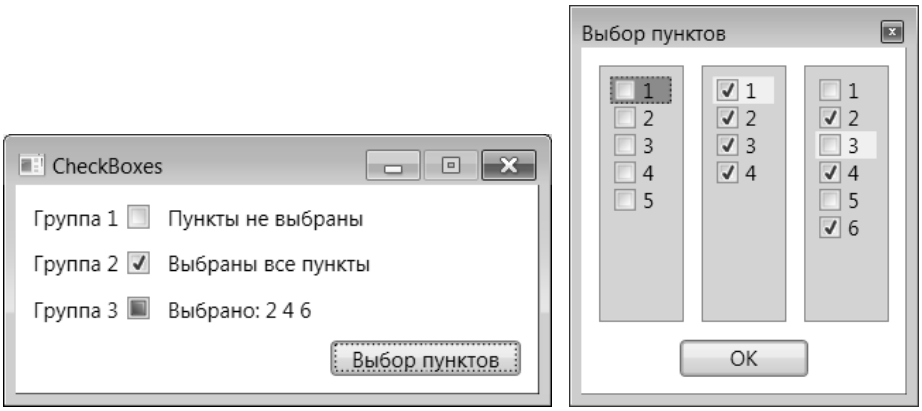

Рис. 6. Окна приложения CHECKBOXES

3.1. 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.

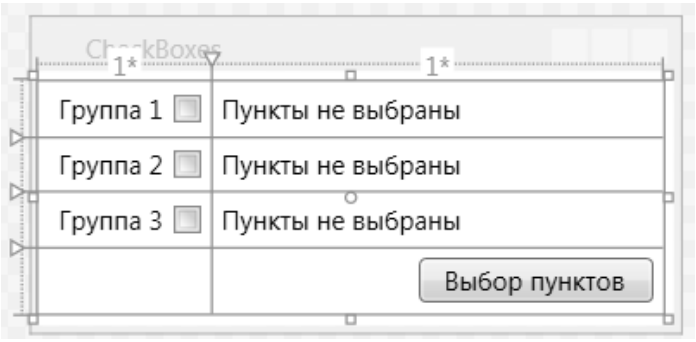

Рис. 7. Макет окна MainWindow приложения CHECKBOXES

```
<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>
```

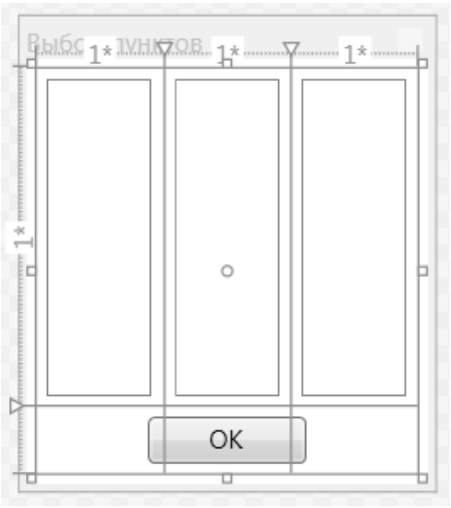

Рис. 8. Макет окна Window1 приложения CHECKBOXES

```
<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. 9. 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.

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.

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 флаж-

ка 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).

3.2. «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.

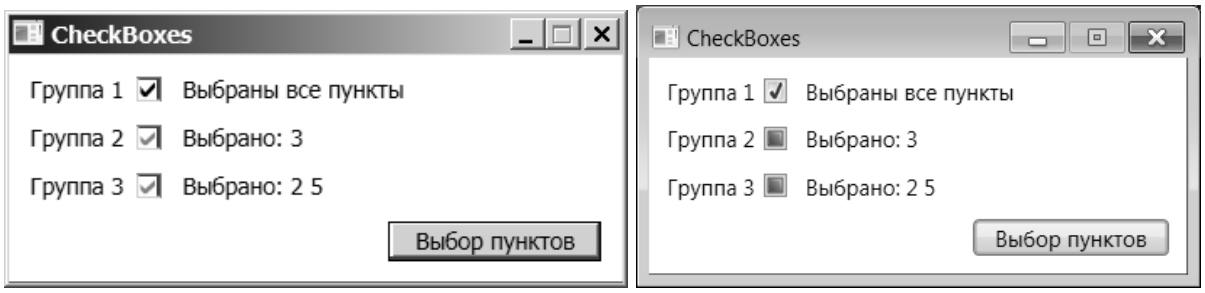

Рис. 10. Вид флажков в третьем состоянии для различных тем Windows 7