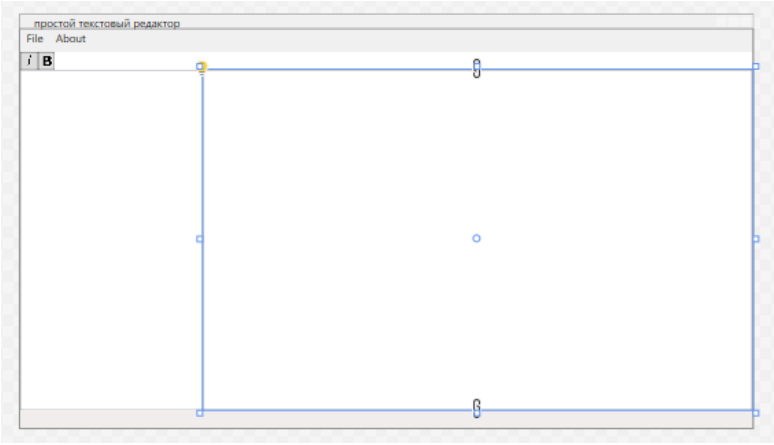
ПЗ 12. Создание проекта с использованием различных управляющих компонентов.

Задание: разработайте проект WPF простого текстового редактора:

- В качестве основного контейнера компоновки используйте DockPanel
- В верхней части DockPanel располагается StackPanel с вложенными элементами menu и wrap panel
- В нижней части располагается StatusBar
- В левой части располагается ListBox
- И остальное место занимает TextBox
- Добавьте в проект папку (ПКМ на имени проекта в обозревателе решений - > добавить -> создать папку), назовите ее icons и переместите туда иконки из архива icons.7z
- Добавьте пункты меню в соответствии со следующей структурой:
    - File
        - open
        - new
        - save
    - Sample
        - Create
        - Load
    - About
        - About program
        - About autors
- Добавьте кнопки в WrapPanel для следующих действий:
    - Курсив
    - Полужирный
    - Подчеркнутый
- При вводе текста:
    - Текст переносится по enter на сл. Строку
    - По нажатию tab производится табулирование
    - Встроенная поддержка проверки орфографии



Теоретическая часть:

Работа с кнопкой.

Элемент Button представляет обычную кнопку:

```
<Button x:Name="button1" Width="60" Height="30" Background="LightGray" />
```

От класса ButtonBase кнопка наследует ряд событий, например, Click, которые позволяют обрабатывать пользовательский ввод.

Чтобы связать кнопку с обработчиком события нажатия, нам надо определить в самой кнопке атрибут Click. А значением этого атрибута будет название обработчика в коде C#. А затем в самом коде C# определить этот обработчик.

Например, код xaml:

```
<Button x:Name="button1" Width="60" Height="30" Content="Нажать"
Click="Button_Click" />
```

И обработчик в коде C#:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Кнопка нажата");
}
```

Либо можно не задавать обработчик через атрибут, а стандартным образом для C# прописать в коде: button1.Click+=Button_Click;

К унаследованным свойствам кнопка имеет такие свойства как IsDefault и IsCancel, которые принимают значения true и false.

Если свойство **IsDefault** установлено в true, то при нажатии клавиши Enter будет вызываться обработчик нажатия этой кнопки.

Аналогично если свойство **IsCancel** будет установлено в true, то при нажатии на клавишу Esc будет вызываться обработчик нажатия этой кнопки.

Например, определим код xaml:

```
<Window x:Class="ControlsApp.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:ControlsApp"
        mc:Ignorable="d"
        Title="Элементы управления" Height="250" Width="300">
    <StackPanel>
        <Button x:Name="acceptButton" Content="OK" IsDefault="True"
Click="acceptButton_Click" />
        <Button x:Name="escButton" Content="Выход" IsCancel="True"
Click="escButton_Click" />
    </StackPanel>
```

```
</Window>
```

А в коде MainWindow.xaml.cs определим следующий код C#:

```
using System.Windows;
namespace ControlsApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        private void acceptButton_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Действие выполнено");
        }
        private void escButton_Click(object sender, RoutedEventArgs e)
        {
            this.Close(); // закрытие окна
        }
    }
}
```

Теперь при нажатии на клавишу Enter будет отображаться сообщение, а при нажатии на Esc будет происходить выход из приложения и закрытие окна.

RepeatButton

Отличительная особенность элемента RepeatButton - непрерывная генерация события Click, пока нажата кнопка. Интервал генерации события корректируется свойствами Delay и Interval.

Сам по себе элемент RepeatButton редко используется, однако он может служить основой для создания ползунка в элементах ScrollBar и ScrollViewer, в которых нажатие на ползунок инициирует постоянную прокрутку.

ToggleButton

Представляет элементарный переключатель. Может находиться в трех состояниях - true, false и "нулевом" (неотмеченном) состоянии, а его значение представляет значение типа bool? в языке C#. Состояние можно установить или получить с помощью свойства IsChecked. Также добавляет три события - Checked (переход в отмеченное состояние), Unchecked (снятие отметки) и Intermediate (если значение равно null). Чтобы отрабатывать все три события, надо установить свойство IsThreeState="True"

ToggleButton, как правило, сам по себе тоже редко используется, однако при этом он служит основой для создания других более функциональных элементов, таких как checkbox и radiobutton.

Работа с компонентом CheckBox

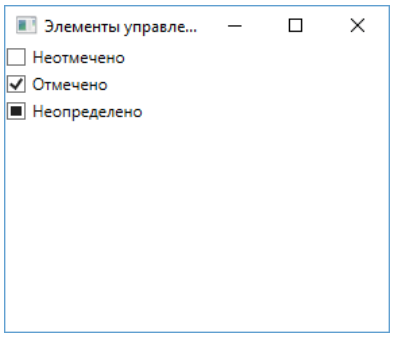
Элемент CheckBox представляет собой обычный флажок. Данный элемент является производным от класса ToggleButton и поэтому может принимать также три состояния: Checked, Unchecked и Intermediate.

Чтобы получить или установить определенное состояние, надо использовать свойство IsChecked, которое также унаследовано от ToggleButton:

```
<StackPanel x:Name="stackPanel">
    <CheckBox x:Name="checkBox1" IsThreeState="True" IsChecked="False"
Height="20" Content="Неотмечено" />
    <CheckBox x:Name="checkBox2" IsThreeState="True" IsChecked="True"
Height="20" Content="Отмечено" />
    <CheckBox x:Name="checkBox3" IsThreeState="True" IsChecked="{x:Null}"
Height="20" Content="Неопределено"/>
</StackPanel>
```

Установка свойства IsChecked="{x:Null}" задает неопределенное состояние для элемента checkbox. Остальные два состояния задаются с помощью True и False. В данном примере также привязан к двум флажкам обработчик события Checked. Это событие возникает при установке checkbox в отмеченное состояние.

А атрибут IsThreeState="True" указывает, что флажок может находиться в трех состояниях.



Ключевыми событиями флажка являются события Checked (генерируется при установке флажка в отмеченное состояние), Unchecked (генерируется при снятии отметки с флажка) и Indeterminate (флажок переведен в неопределенное состояние). Например, определим флажок:

```
<CheckBox x:Name="checkBox" IsChecked="False" Height="20" Content="Флажок"
    IsThreeState="True"
    Unchecked="checkBox_Unchecked"
    Indeterminate="checkBox_Indeterminate"
    Checked="checkBox_Checked" />
```

А в файле кода C# пропишем для него обработчики:

```
private void checkBox_Checked(object sender, RoutedEventArgs e)
{
    MessageBox.Show(checkBox.Content.ToString() + " отмечен");
}

private void checkBox_Unchecked(object sender, RoutedEventArgs e)
```

```
{
    MessageBox.Show(checkBox.Content.ToString() + " не отмечен");
}

private void checkBox_Indeterminate(object sender, RoutedEventArgs e)
{
    MessageBox.Show(checkBox.Content.ToString() + " в неопределенном
состоянии");
}
```

Программное добавление флажка:

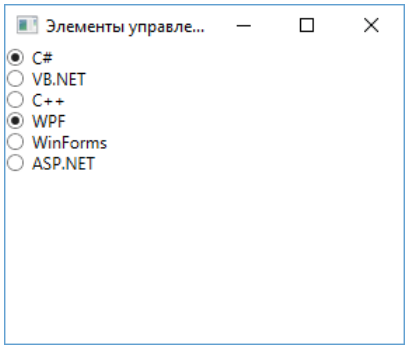
```
using System.Windows;
using System.Windows.Controls;
namespace ControlsApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            // создаем флажок
            CheckBox checkBox2 = new CheckBox { Content = "Новый флажок",
MinHeight = 20, IsChecked=true };
            // установка обработчика
            checkBox2.Checked += checkBox_Checked;
            // добавление в StackPanel
            stackPanel.Children.Add(checkBox2);
        }

        private void checkBox_Checked(object sender, RoutedEventArgs e)
        {
            CheckBox chBox = (CheckBox)sender;
            MessageBox.Show(chBox.Content.ToString() + " отмечен");
        }
    }
}
```

Работа с компонентом RadioButton

Элемент управления, также производный от ToggleButton, представляющий переключатель. Главная его особенность - поддержка групп. Несколько элементов RadioButton можно объединить в группы, и в один момент времени мы можем выбрать из этой группы только один переключатель. Например,

```
<StackPanel x:Name="stackPanel">
    <RadioButton GroupName="Languages" Content="C#" IsChecked="True" />
    <RadioButton GroupName="Languages" Content="VB.NET" />
    <RadioButton GroupName="Languages" Content="C++" />
    <RadioButton GroupName="Technologies" Content="WPF" IsChecked="True" />
    <RadioButton GroupName="Technologies" Content="WinForms" />
    <RadioButton GroupName="Technologies" Content="ASP.NET" />
</StackPanel>
```



Чтобы включить элемент в определенную группу, используется свойство GroupName. В данном случае у нас две группы - Languages и Technologies. Мы можем отметить не более одного элемента RadioButton в пределах одной группы, зафиксировав тем самым выбор из нескольких возможностей.

Чтобы проследить за выбором того или иного элемента, мы также можем определить у элементов событие Checked и его обрабатывать в коде:

```
<RadioButton GroupName="Languages" Content="VB.NET"
Checked="RadioButton_Checked" />
```

Обработчик в файле кода:

```
private void RadioButton_Checked(object sender, RoutedEventArgs e)
{
    RadioButton pressed = (RadioButton)sender;
    MessageBox.Show(pressed.Content.ToString());
}
```

Программное добавление элемента RadioButton:

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

```
namespace ControlsApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            RadioButton rb = new RadioButton { IsChecked = true, GroupName =
"Languages", Content = "JavaScript" };
            rb.Checked += RadioButton_Checked;
            stackPanel.Children.Add(rb);
        }

        private void RadioButton_Checked(object sender, RoutedEventArgs e)
        {
            RadioButton pressed = (RadioButton)sender;
            MessageBox.Show(pressed.Content.ToString());
        }
    }
}
```

Работа со всплывающими подсказками

ToolTip

Элемент ToolTip представляет всплывающую подсказку при наведении на какой-нибудь элемент. Для определения всплывающей подсказки у элементов уже есть свойство ToolTip, которому можно задать текст, отображаемый при наведении:

```
<Button Content="Tooltip" ToolTip="Всплывающая подсказка для кнопки"
Height="30" Width="80" />
```

Также мы можем более точно настроить всплывающую подсказку с помощью свойства Button.ToolTip:

```
<Button Content="Tooltip" Height="30" Width="80">
    <Button.ToolTip>
        <ToolTip>
            Всплывающая подсказка для кнопки
        </ToolTip>
    </Button.ToolTip>
</Button>
```

Всплывающие подсказки можно применять не только кнопкам, но и ко всем другим элементам управления.

Свойства ToolTip

Некоторые полезные свойства элемента Tooltip:

- HasDropShadow: определяет, будет ли всплывающая подсказка отбрасывать тень.
- Placement: определяет, как будет позиционироваться всплывающая подсказка на окне приложения. По умолчанию ее верхний левый угол позиционируется на указатель мыши.
- HorizontalOffset/VerticalOffset: определяет смещение относительно начального местоположения.
- PlacementTarget: определяет позицию всплывающей подсказки относительно другого элемента управления.

Popup

Элемент Popup также представляет всплывающее окно, только в данном случае оно имеет другую функциональность. Если Tooltip отображается автоматически при наведении и также автоматически скрывается через некоторое время, то в случае с Popup все эти действия нам надо задавать вручную.

- Так, чтобы отразить при наведении мыши на элемент всплывающее окно, нам надо соответственным образом обработать событие MouseEnter.
- Свойство StaysOpen="False". По умолчанию оно равно True, а это значит, что при отображении окна, оно больше не исчезнет, пока мы не установим явно значение этого свойства в False.

Пример:

```
<StackPanel>
    <Button Content="Popup" Width="80" MouseEnter="Button_MouseEnter_1"
HorizontalAlignment="Left" />
    <Popup x:Name="popup1" StaysOpen="False" Placement="Mouse" MaxWidth="180"
         AllowsTransparency="True"  >
        <TextBlock TextWrapping="Wrap" Width="180" Background="LightPink"
Opacity="0.8" >
            Чтобы узнать больше, посетите сайт metanit.com
        </TextBlock>
    </Popup>
</StackPanel>
```

И обработчик наведения курсора мыши на кнопку в коде c#:

```
private void Button_MouseEnter_1(object sender, MouseEventArgs e)
{
    popup1.IsOpen = true;
}
```

Контейнеры

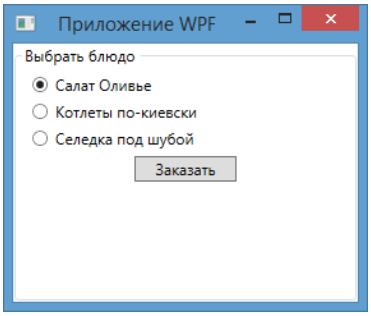
Особая группа элементов управления образована от класса HeaderedContentControl, который является подклассом ContentControl. Эта группа отличается тем, что позволяет задать заголовок содержимому. В эту группу элементов входят GroupBox и Expander.

GroupBox

Элемент GroupBox организует наборы элементов управления в отдельные группы. При этом мы можем определить у группы заголовок:

```
<GroupBox Header="Выбрать блюдо" Padding="5">
    <StackPanel>
        <RadioButton IsChecked="True" Margin="3">Салат Оливье</RadioButton>
        <RadioButton Margin="3">Котлеты по-киевски</RadioButton>
        <RadioButton Margin="3">Селедка под шубой</RadioButton>
        <Button Width="80" Margin="3">Заказать</Button>
    </StackPanel>
</GroupBox>
```

GroupBox включает группу различных элементов, однако, как и всякий элемент управления содержимым, он **принимает внутри себя только один контейнер**, поэтому сначала мы вкладываем в GroupBox общий контейнер, а в него уже все остальные элементы.



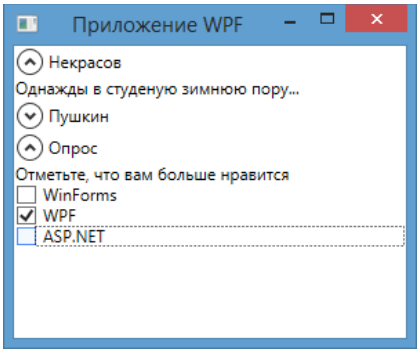
Expander

Представляет скрытое содержимое, раскрывающееся по нажатию мышкой на указатель в виде стрелки. Причем содержимое опять же может быть самым разным: кнопки, текст, картинки и т.д.

С помощью свойства IsExpanded можно задать раскрытие узла при старте приложения. По умолчанию узел скрыт. Пример использования:

```
<StackPanel>
    <Expander Header="Некрасов">
        <TextBlock>Однажды в студеную зимнюю пору...</TextBlock>
    </Expander>
    <Expander Header="Пушкин">
        <TextBlock>Онегин был, по мнению многих, ученый малый, но
...</TextBlock>
    </Expander>
    <Expander Header="Опрос">
        <StackPanel>
```

```
            <TextBlock>Отметьте, что вам больше нравится</TextBlock>
            <CheckBox>WinForms</CheckBox>
            <CheckBox>WPF</CheckBox>
            <CheckBox>ASP.NET</CheckBox>
        </StackPanel>
    </Expander>
</StackPanel>
```



Программное создание Expandera:

```
StackPanel expanderPanel = new StackPanel();
expanderPanel.Children.Add(new CheckBox { Content = "WinForms" });
expanderPanel.Children.Add(new CheckBox { Content = "WPF" });
expanderPanel.Children.Add(new CheckBox { Content = "ASP.NET" });

Expander expander = new Expander();
expander.Header = "Выберите технологию";
expander.Content = expanderPanel;

stackPanel.Children.Add(expander);
```

Работа с текстом

TextBlock

Элемент предназначен для вывода текстовой информации, для создания простых надписей:

```
<StackPanel>
    <TextBlock>Текст1</TextBlock>
    <TextBlock Text="Текст2" />
</StackPanel>
```

Ключевым свойством здесь является свойство Text, которое задает текстовое содержимое. Причем в случае <TextBlock>Текст1</TextBlock> данное свойство задается неявно.

С помощью таких свойств, как FontFamily, TextDecorations и др., мы можем настроить отображение текста. Однако мы можем задать и более сложное форматирование, например:

```
<TextBlock TextWrapping="Wrap">
    <Run FontSize="20" Foreground="Red" FontWeight="Bold">О</Run>
    <Run FontSize="16" Foreground="LightSeaGreen">негин был, по мненью многих...</Run>
</TextBlock>
```

Элементы Run представляют куски обычного текста, для которых можно задать отдельное форматирование.

Для изменения параметров отображаемого текста данный элемент имеет такие свойства, как LineHeight, TextWrapping и TextAlignment.

- Свойство **LineHeight** позволяет указывать высоту строк.
- Свойство **TextWrapping** позволяет переносить текст при установке этого свойства TextWrapping="Wrap". По умолчанию это свойство имеет значение NoWrap, поэтому текст не переносится.
- Свойство **TextAlignment** выравнивает текст по центру (значение Center), правому (Right) или левому краю (Left): <TextBlock TextAlignment="Right">
- Для декорации текста используется свойство **TextDecorations**, например, если TextDecorations="Underline", то текст будет подчеркнут.
- Если нам вдруг потребуется перенести текст на другую строку, то тогда мы можем использовать элемент **LineBreak**

TextBox

Если TextBlock просто выводит статический текст, то этот элемент представляет поле для ввода текстовой информации.

Он также, как и TextBlock, имеет свойства TextWrapping, TextAlignment и TextDecorations.

С помощью свойства **MaxLength** можно задать предельное количество вводимых символов.

```
<TextBox MaxLength="250" TextChanged="TextBox_TextChanged">Начальный текст</TextBox>
```

В коде C# мы можем обработать событие изменения текста:

```
private void TextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    TextBox textBox = (TextBox)sender;
    MessageBox.Show(textBox.Text);
}
```

По умолчанию, если вводимый текст превышает установленные границы поля, то текстовое поле растет, чтобы вместить весь текст. Но визуально это не очень хорошо выглядит. Поэтому, как и в случае с TextBlock, мы можем перенести непомещающийся текст на новую строку, установив свойство TextWrapping="Wrap".

- Чобы переводить по нажатию на клавишу Enter курсор на следующую строку, нам надо установить свойство **AcceptsReturn**="True".
- Также мы можем добавить полю возможность создавать табуляцию с помощью клавиши Tab, установив свойство **AcceptsTab**="True"
- Для отображения полос прокрутки TextBox поддерживает свойства VerticalScrollBarVisibility и HorizontalScrollBarVisibility:

```
<TextBox AcceptsReturn="True" Height="100" VerticalScrollBarVisibility="Auto"
         HorizontalScrollBarVisibility="Auto">Начальный текст</TextBox>
```

Возможно, при создании приложения нам потребуется сделать текстовое поле недоступным для ввода (на время в зависимости от условий или вообще), тогда для этого нам надо установить свойство **IsReadOnly**="True".

Для выделения текста есть свойства **SelectionStart, SelectionLength и SelectionText**. Например, выделим программно текст по нажатию кнопки:

```
<StackPanel>
    <TextBox x:Name="textBox1" Height="100" SelectionBrush="Blue" />
    <Button Content="Выделить текст" Height="30" Width="100"
Click="Button_Click" Margin="10" />
</StackPanel>
```

Обработчик нажатия кнопки:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    textBox1.SelectionStart = 5;
    textBox1.SelectionLength = 10;
    textBox1.Focus();
    // данное выражение эквивалентно
    //textBox1.Select(5, 10);
}
```

Проверка орфографии

TextBox обладает встроенной поддержкой орфографии. Чтобы ее задействовать, надо установить свойство **SpellCheck.IsEnabled="True".** Кроме того, по умолчанию проверка орфографии распространяется только на английский язык, поэтому, если приложение заточено под другой язык, нам надо его явным образом указать через свойство Language:

```
<DockPanel>
    <TextBox SpellCheck.IsEnabled="True" Language="ru-ru">Привет, как
дила?</TextBox>
</DockPanel>
```

Метка (Label)

Главной особенностью меток является поддержка мнемонических команд-клавиш быстрого доступа, которые передают фокус связанному элементу. Например,

```
<Label Target="{Binding ElementName=TextBox1}">_привет</Label>
<TextBox Name="TextBox1" Margin="0 30 0 0" Height="30" Width="100"></TextBox>
```

Теперь, нажав на клавишу "п", мы переведем фокус на связанное текстовое поле. При вызове приложения подчеркивание не отображается, чтобы отображать подчеркивание, надо нажать на клавишу Alt. Тогда чтобы перевести фокус на связанное текстовое поле необходимо будет нажать сочетание Alt + "п". Если не предполагается использование клавиш быстрого доступа, то для вывода обычной текста вместо меток лучше использовать элемент TextBlock.

PasswordBox

Элемент предназначен для ввода парольной информации. По сути это тоже текстовое поле, только для ввода символов используется маска. Свойство PasswordChar устанавливает символ маски, отображаемый при вводе пароля. Если это свойство не задано, то по умолчанию для маски символа используется черная точка. Свойство Password устанавливает парольную строку, отображаемую по умолчанию при загрузке окна приложения.

```
<StackPanel>
    <PasswordBox PasswordChar="*" MinHeight="30" />
    <PasswordBox MinHeight="30" />
</StackPanel>
```

Работа с меню

Данный элемент служит для создания стандартных меню:

```
<Menu Height="25" VerticalAlignment="Top">
    <MenuItem Header="File">
        <MenuItem Header="New Project" ></MenuItem>
        <MenuItem Header="Open Project" >
            <MenuItem Header="WinForms"></MenuItem>
            <MenuItem Header="WPF" ></MenuItem>
        </MenuItem>
        <Separator />
        <MenuItem Header="Exit" ></MenuItem>
    </MenuItem>
    <MenuItem Header="Edit" ></MenuItem>
    <MenuItem Header="View" ></MenuItem>
</Menu>
```

Элемент Menu включает набор элементов MenuItem, которые опять же являются элементами управления содержимым и могут включать другие элементы MenuItem и не только. Также мы можем вложить в меню и другие элементы, которые неявно будут преобразованы в MenuItem. Например:

```
<Menu Height="25" VerticalAlignment="Top">
    <MenuItem Header="File">
        <Button Content="Exit" />
```

```
    </MenuItem>
    <MenuItem Header="Edit" ></MenuItem>
    <MenuItem Header="View" ></MenuItem>
    <Button Content="Кнопка в меню" />
</Menu>
```

Также для разделения отдельных пунктов меню можно включать элемент Separator, как в примере выше.

Мы также можем настроить внешний вид отображения, задав свойство MenuItem.Header или использовав свойство Icons:

```
<Menu Height="25" VerticalAlignment="Top" Background="LightGray">
    <MenuItem>
        <MenuItem.Header>
            <StackPanel Orientation="Horizontal">
                <Ellipse Height="10" Width="10" Fill="Black" Margin="0 0 5 0"
/>
                <TextBlock>File</TextBlock>
            </StackPanel>
        </MenuItem.Header>
    </MenuItem>
    <MenuItem Header="Edit">
        <MenuItem.Icon>
            <Image Source="C:\Users\Eugene\Documents\pen.png"></Image>
        </MenuItem.Icon>
    </MenuItem>
    <MenuItem Header="View"></MenuItem>
</Menu>
```

Чтобы обработать нажатие пункта меню и произвести определенное действие, можно использовать событие Click, однако в будущем мы познакомимся с еще одним инструментом под названием команды, который также широко применяется для реакции на нажатие кнопок меню. А пока свяжем обработчик с событием:

```
<MenuItem Header="View"  Click="MenuItem_Click"></MenuItem>
```

И определим сам обработчик в коде C#:

```
private void MenuItem_Click(object sender, RoutedEventArgs e)
{
    MenuItem menuItem = (MenuItem)sender;
    MessageBox.Show(menuItem.Header.ToString());
}
```

ContextMenu

Класс ContextMenu служит для создания контекстных всплывающих меню, отображающихся после нажатия на правую кнопку мыши. Этот элемент также содержит

коллекцию элементов MenuItem. Однако сам по себе ContextMenu существовать не может и должен быть прикреплен к другому элементу управления. Для этого у элементов есть свойство ContextMenu:

```
<ListBox Name="list" Height="145">
    <ListBoxItem Margin="3">MS SQL Server</ListBoxItem>
    <ListBoxItem Margin="3">MySQL</ListBoxItem>
    <ListBoxItem Margin="3">Oracle</ListBoxItem>
    <ListBox.ContextMenu>
        <ContextMenu>
            <MenuItem Header="Копировать"></MenuItem>
            <MenuItem Header="Вставить"></MenuItem>
            <MenuItem Header="Вырезать"></MenuItem>
            <MenuItem Header="Удалить"></MenuItem>
        </ContextMenu>
    </ListBox.ContextMenu>
</ListBox>
```

И при нажатии правой кнопкой мыши на один из элементов отобразится контекстное меню.