1. Панель инструментов: TEXTEDIT, версия 4

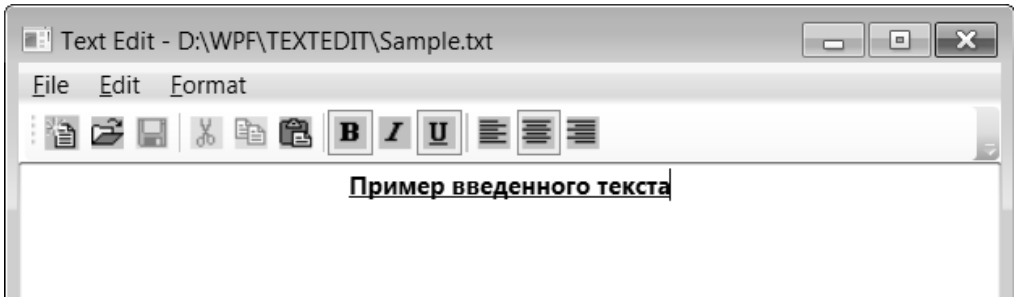

Рис. 1. Верхняя часть окна приложения TEXTEDIT версии 4

1.1. 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.

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. 2. 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>
```

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).

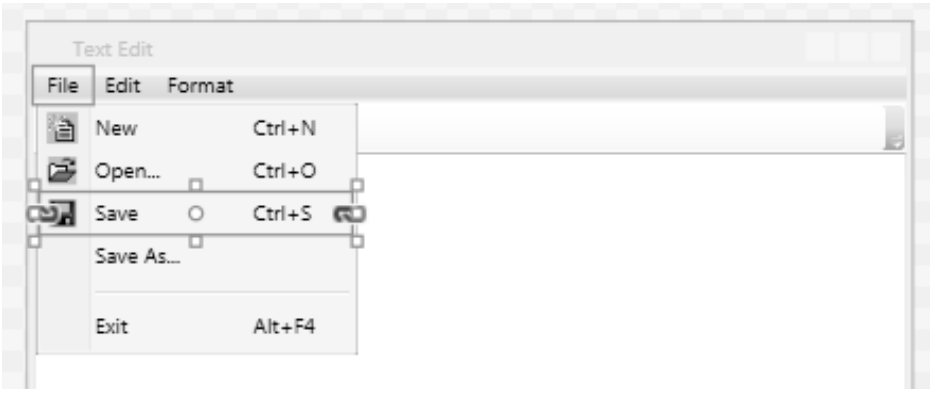

Рис. 3. Макет окна приложения TEXTEDIT версии 4 (меню с изображениями)

```
<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).

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. 4. Окно приложения TEXTEDIT версии 4 с неактивными кнопками

1.2. 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).

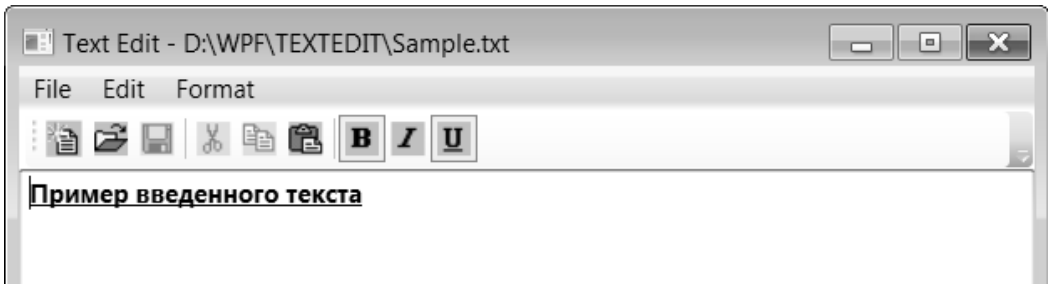

Рис. 5. Окно приложения TEXTEDIT версии 4 с кнопками-переключателями

```
<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.

1.3. 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»).

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 обязательно 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.

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

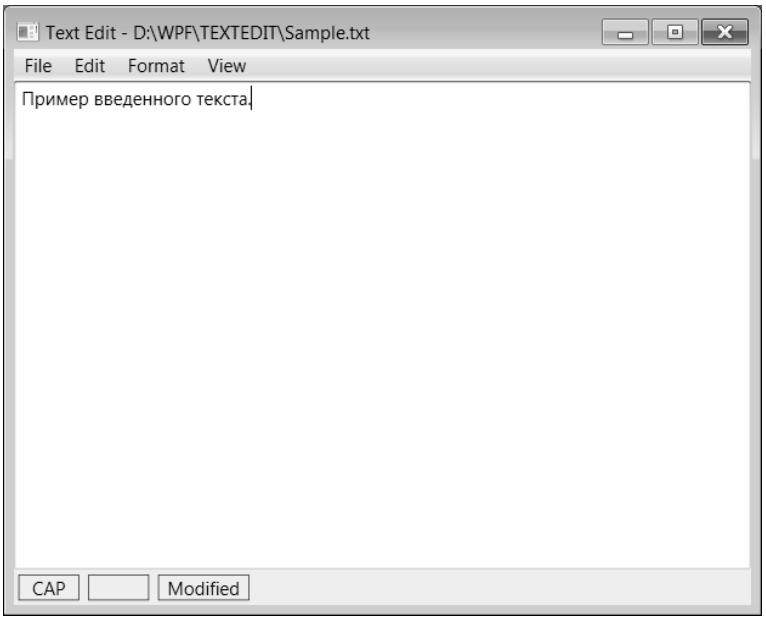

Рис. 6. Окно приложения TEXTEDIT версии 5

2.1. 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); }
}
```

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* с позиционным параметром-ключом ModifConv.

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

2.2. 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).

2.3. 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.