1. Работа с несколькими окнами: WINDOWS

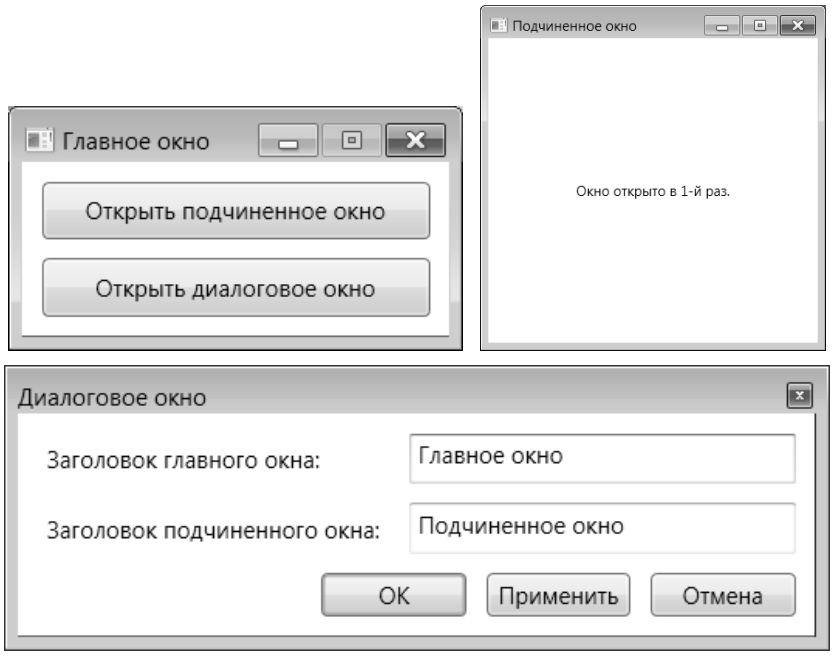

Рис. 1. Окна приложения WINDOWS

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

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

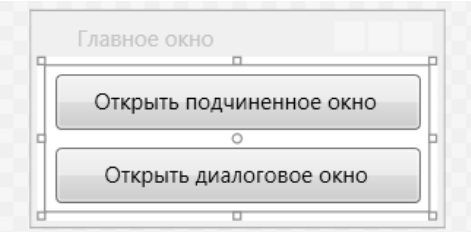

Рис. 2. Макет окна MainWindow приложения WINDOWS

MainWindow.xaml (рис. 8):

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

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

Window1.xaml:

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

  </Grid>
</Window>
```

Window2.xaml:

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

  </Grid>
</Window>
```

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

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

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

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

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

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

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

1.2. Решение проблем, возникающих при повторном открытии подчиненных окон

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

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

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

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

Window1.xaml.cs и Window2.xaml.cs:

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

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

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

Для окна MainWindow измените обработчик button1_Click:

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

Для окна Window1 определите обработчик события IsVisibleChanged:

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

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

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

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

1.4. Окно с содержимым в виде обычного текста

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

В начало описания класса Window1 добавьте поле

```
int count;
```

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

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

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

1.5. Модальные и обычные кнопки диалогового окна

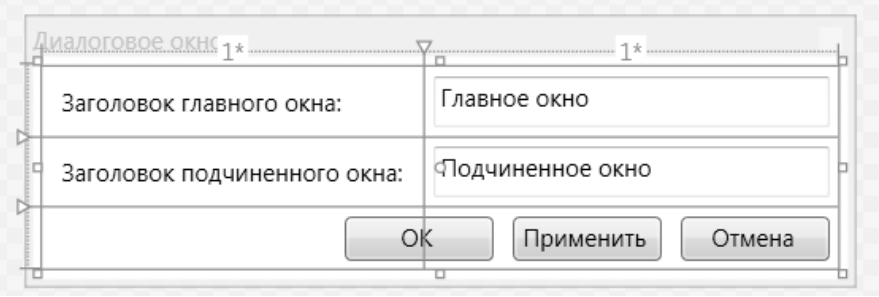

Рис. 3. Макет окна Window2 приложения WINDOWS

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

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

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

```
bool dialogRes;

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

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

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

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

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

В классе MainWindow дополните обработчик button2_Click:

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

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

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

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

В классе Window2 добавьте в метод Window_IsVisibleChanged следующий оператор:

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

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

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

1.7. Запрос на подтверждение закрытия окна

В классе Window1 измените обработчик Window_Closing:

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

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

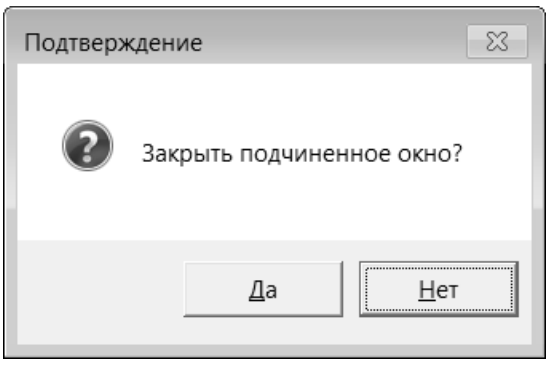

Рис. 4. Стандартное диалоговое окно

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

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

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

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

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

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

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

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

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

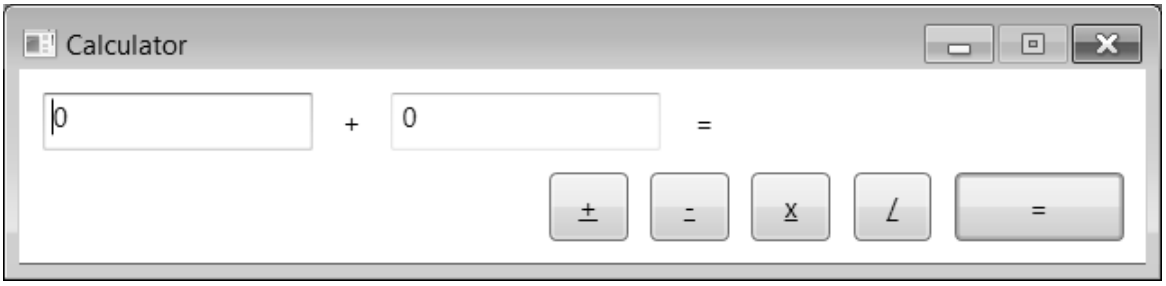

Рис. 5. Окно приложения CALC

2.1. Настройка коллективного обработчика событий

Рис. 6. Макет окна MainWindow

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

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

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

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

Дополните созданный в cs-файле обработчик следующим образом:

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

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

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

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

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

2.2. Организация вычислений

Определите обработчик события Click для кнопки button5:

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

```
private void button5_Click(object sender, RoutedEventArgs e)
{
    double x = 0, x1, x2;
```

```
    if (!double.TryParse(textBox1.Text, out x1) ||
      !double.TryParse(textBox2.Text, out x2))
    {
        label2.Content = "= ERROR";
        return;
    }
    switch (label1.Content as string)
    {
        case "+":
            x = x1 + x2; break;
        case "-":
            x = x1 - x2; break;
        case "x":
            x = x1 * x2; break;
        case "/":
            x = x1 / x2; break;
    }
    label2.Content = "= " + x;
}
```

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

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

Исправление. В начало метода button5_Click добавьте оператор

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

2.3. Простейшие приемы ускорения работы с помощью клавиатуры

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

Обратите внимание на добавленные символы подчеркивания в свойствах Content.

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

Рис. 7. Окно приложения CALC с подчеркнутыми символами в подписях кнопок

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

Исправление. Измените оператор в методе button1_Click следующим образом:

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

Недочет. Теперь, когда программа содержит средства для быстрого выполнения действий с помощью клавиатуры, более наглядно проявляется

недочет, который имелся в ней с самого начала: при запуске данной программы в ней отсутствует компонент, имеющий фокус. Для того чтобы фокус появился на первом поле ввода (и при этом в нем отобразился вертикальный курсор), необходимо либо щелкнуть мышью на этом поле, либо нажать клавишу Tab. Было бы удобнее, если бы фокус устанавливался на первое поле ввода сразу после запуска программы.

Исправление. Добавьте в конструктор класса MainWindow оператор:

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

2.4. Использование обработчика событий от клавиатуры

Определите обработчик события PreviewTextInput для MainWindow:

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

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

Кроме того, измените метод button1_Click следующим образом:

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

Результат. Теперь для ввода любой операции достаточно нажать соответствующую клавишу (поскольку клавиша «–» может использоваться для ввода отрицательных чисел, в качестве ускорителя для кнопки «–» вы-

брана комбинация Shift+«–», соответствующая символу подчеркивания «_»). При вводе чисел игнорируются все клавиши, кроме цифровых, «−», «,» и Backspace (для обозначения символа, генерируемого клавишей Backspace, в C# можно использовать управляющую последовательность '\b'; нажатие этой клавиши обеспечивает удаление символа, расположенного *слева* от курсора в активном поле ввода).

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

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

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

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

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

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

Исправление. Измените фрагмент последнего оператора в методе Window_PreviewTextInput:

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

2.5. Контроль за изменением исходных данных

Добавьте в метод button1_Click следующий оператор:

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

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

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

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

```
private void textBox1_TextChanged(object sender,
  TextChangedEventArgs e)
{
    if (label2 != null)
        label2.Content = "=";
}
```

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

3. Работа с датами и временем: CLOCK

Рис. 8. Окно приложения CLOCK

3.1. Отображение текущего времени

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

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

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

В описание класса MainWindow добавьте поле

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

В конструктор класса добавьте следующие операторы:

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

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

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

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

Рис. 9. Окно приложения CLOCK (первый вариант)

Комментарий

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

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

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

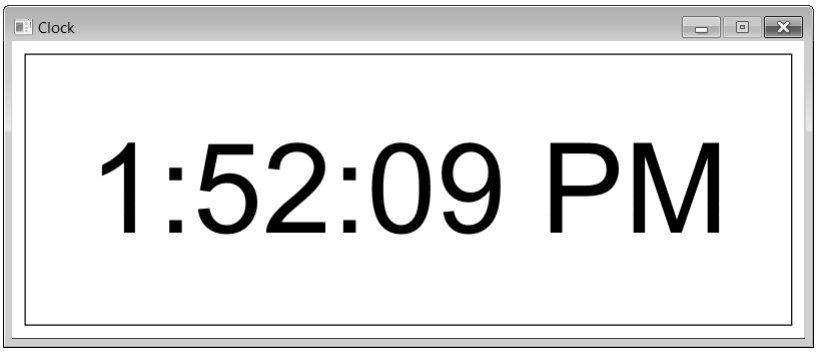

Рис. 10. Окно приложения CLOCK с измененными региональными настройками

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

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

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

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

3.2. Реализация возможностей секундомера

```
<Window x:Class="CLOCK.MainWindow"
    … >
  <StackPanel>
    <Border Margin="10,10,10,0" … >
      …
    </Border>
    <StackPanel Margin="5" Orientation="Horizontal"
        HorizontalAlignment="Center">
      <CheckBox x:Name="checkBox1" Margin="5" Content="_Timer"
          VerticalContentAlignment="Center"
          VerticalAlignment="Center" Padding="10,0"
          Click="checkBox1_Click" />
      <Button x:Name="button1"  MinWidth="75" Padding ="5"
          Margin="5" Content="_Start/Stop" IsEnabled="False"
          Click="button1_Click" />
      <Button x:Name="button2"  MinWidth="75" Padding ="5"
          Margin="5" Content="_Reset" IsEnabled="False"
          Click="button2_Click" />
    </StackPanel>
  </StackPanel>
</Window>
```

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

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

Кроме того, добавьте в класс MainWindow новое поле

```
int t;
```

а также дополните обработчик timer1_Tick:

```
void timer1_Tick(object sender, EventArgs e)
{
    if ((bool)checkBox1.IsChecked)
    {
        t++;
        label1.Text = string.Format("{0}:{1}",
          t / 10, t % 10);
    }
    else
        label1.Text = DateTime.Now.ToLongTimeString();
}
```

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

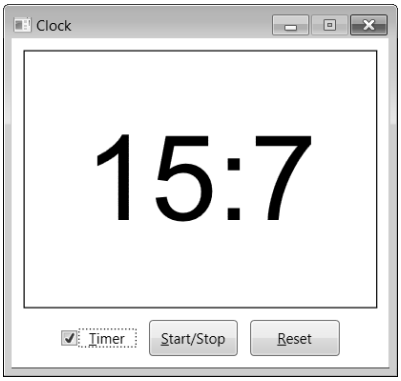

Рис. 11. Окно приложения CLOCK в режиме секундомера

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

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

```
<Border … MinWidth="600">
```

Результат. Теперь ширина окна остается неизменной в любом режиме.

Ошибка. Кажущаяся правильность работы секундомера обманчива. В этом можно убедиться, если не останавливать секундомер в течение некоторого времени (выполняя при этом другие действия на компьютере), после чего сравнить результат с точным временем. Причина заключается в том, что событие Tick наступает *примерно* через каждые 100 мс; кроме того, надо учитывать, что данное событие наступает только при отсутствии других событий, которые требуется обработать программе. Если программу выполняет какой-либо обработчик длительное время, то в течение этого времени информация секундомера не будет обновляться, а затем отсчет времени продолжится с прежнего значения. Для правильной реализации секундомера надо связать его с *часами компьютера* (используя метод Now).

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

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

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

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

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

Аналогичные изменения внесите в метод button2_Click:

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

Добавьте в метод button1_Click операторы:

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

И откорректируйте метод timer1_Tick:

```
private void timer1_Tick(object sender, EventArgs e)
{
    if ((bool)checkBox1.IsChecked)
    {
        t++;
        label1.Text = string.Format("{0}:{1}",
          t / 10, t % 10);
        TimeSpan s = DateTime.Now - startTime - pauseSpan;
        label1.Text = string.Format("{0}:{1}",
          s.Minutes * 60 + s.Seconds, s.Milliseconds / 100);
    }
    else
      label1.Text = DateTime.Now.ToLongTimeString();
}
```

3.3. Альтернативные варианты выполнения команд с помощью мыши

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

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

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

3.4. Отображение текущего состояния часов и секундомера на панели задач

В метод timer1_Tick добавьте следующий фрагмент:

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

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

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

Исправление. Определите обработчик события StateChanged для окна:

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

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

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