**Министерство образования и науки Российской Федерации**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ**

**УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ,**

**МЕХАНИКИ И ОПТИКИ»**

**Факультет «ИКТ»**

**Дисциплина**

«Создание программного обеспечения инфокоммуникационных систем»

**Лабораторная работа 2**

**«**Разработка структурной модели – диаграммы классов**»**

**Выполнил:**

студент группы K3415

Дружинин К.А

**Проверил**:

Осипов Н. А.

Санкт-Петербург

2021

**Практическое задание 4. Разработка пользовательского интерфейса**

**Упражнение 1. Использование простых элементов управления**

1. Создадим новый проект WPF Application с именем «UserIn1».
2. Определим пользовательский интерфейс приложения.
3. Определим свойства добавленных элементов в соответствии с заданием, чтобы связать элементы Label с элементами ввода текста.

Привязка позволяет производить навигацию по элементам формы с помощью комбинации клавиш Alt + клавиша быстрого доступа. После ввода комбинации связанное поле textbox оказывается выбранным.

1. Построим и протестируем приложение.

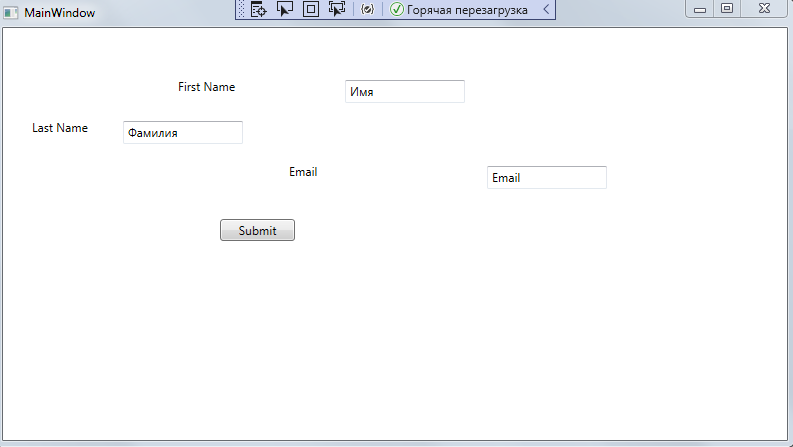


Рисунок 1 - Тестирование приложения

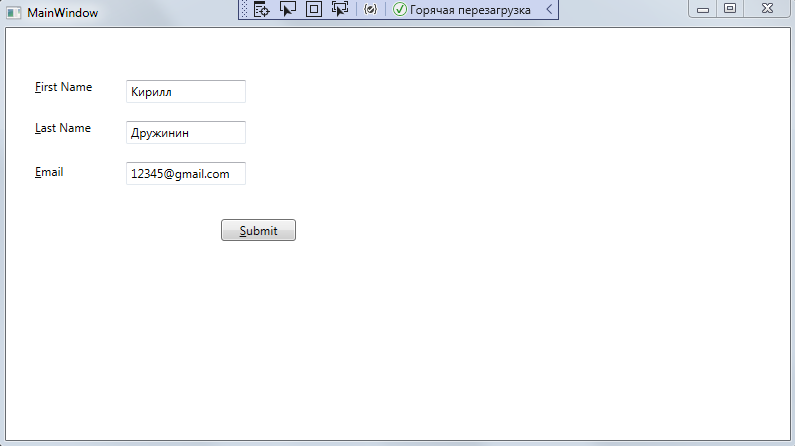


Рисунок 2 - Тестирование приложения

**Практическое задание 4. Разработка пользовательского интерфейса**

**Упражнение 2. Размещение связанных элементов**

1. Создадим новый проект WPF Application с именем «UserIn2».
2. Определим пользовательский интерфейс приложения.
3. Определим свойства добавленных элементов в соответствии с заданием, включая 4 элемента в панели инструментов.

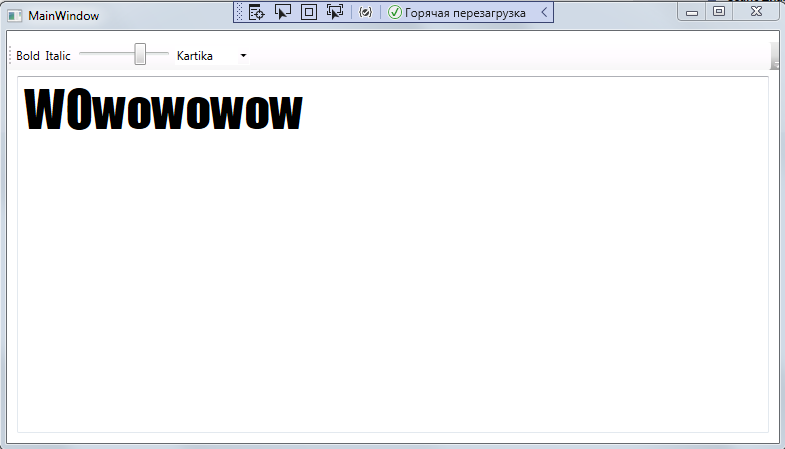


Рисунок 3 - Пользовательский интерфейс приложения

1. Добавим обработчики события **Click** для элементов кнопок на панели инструментов. Для элемента Slider добавим обработчик события **SliderChanged**, а для элемента ComboBox события **ComboBoxSelectionChanged**.
2. Добавим инициализацию элемента ComboBox коллекцией стандартных шрифтов в конструкторе формы.
3. Запустим и протестируем приложение.

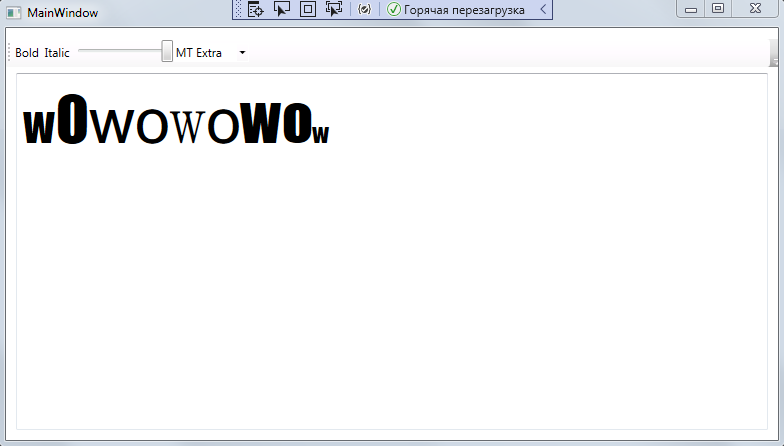


Рисунок 4 - Тестирование приложения.

Как и ожидалось элементы меню меняют оформление выделенного текста.

**Практическое задание 4. Разработка пользовательского интерфейса**

**Упражнение 3. Работа с элементами управления раскладкой**

1. Создадим новый проект WPF Application с именем «UserIn3».
2. В представлении XAML изменим открывающий и закрывающий теги **Grid** тегамии **DockPanel**.
3. Внесем изменения в представление XAML формы в соответствии с условиями задания.

<Window x:Class="UserIn3.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:UserIn3"

mc:Ignorable="d"

Title="MainWindow" Height="450" Width="800">

<DockPanel>

<ToolBar Height="26" Name="toolBar1" Width="800" DockPanel.Dock="Top" >

<Button Click="Button\_Click">Bold</Button>

<Button Click="Button\_Click\_1">Italic</Button>

<Slider Name="Slider1" Minimum="2" Maximum="72" Width="100" ValueChanged="Slider1\_ValueChanged"></Slider>

</ToolBar>

<Grid Name="grid1" DockPanel.Dock="Bottom">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="100" />

<ColumnDefinition Width="5" />

<ColumnDefinition Width="\*" />

</Grid.ColumnDefinitions>

<RichTextBox Grid.Column="2" Name="richTextBox1" />

<ListBox Grid.Column="0" Name="listBox1" SelectionChanged="listBox1\_SelectionChanged"></ListBox>

<GridSplitter Name="gridSplitter1" Margin="0" Width="5" Grid.Column="1" HorizontalAlignment="Left" />

</Grid>

</DockPanel>

</Window>

Текст 1 - XAML представление формы.

1. Определим обработчики событий для созданных пользовательских элементов.

namespace UserIn3

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

foreach (FontFamily F in Fonts.SystemFontFamilies)

{

listBox1.Items.Add(F.ToString());

}

}

private void Button\_Click(object sender, RoutedEventArgs e)

{

richTextBox1.Selection.ApplyPropertyValue(FontWeightProperty, FontWeights.Bold);

}

private void Button\_Click\_1(object sender, RoutedEventArgs e)

{

richTextBox1.Selection.ApplyPropertyValue(FontStyleProperty, FontStyles.Italic);

}

private void Slider1\_ValueChanged(object sender, RoutedPropertyChangedEventArgs<double> e)

{

try

{

richTextBox1.Selection.ApplyPropertyValue(FontSizeProperty, Slider1.Value.ToString());

}

catch { }

}

private void listBox1\_SelectionChanged(object sender, SelectionChangedEventArgs e)

{

richTextBox1.Selection.ApplyPropertyValue(FontFamilyProperty, new FontFamily((string)listBox1.SelectedItem));

}

}

}

Текст 2 - класс формы с определенными обработчиками событий

1. Построим и протестируем приложение.

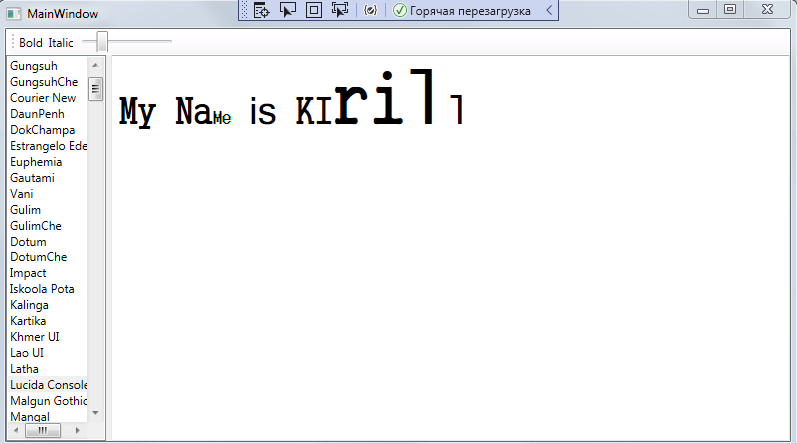


Рисунок 5 - тестирование приложения

**Практическое задание 5. Использование компонента BackgroundWorker**

**Упражнение 1. Работа с компонентом BackgroundWorker**

Класс **BackgroundWorker** появился в .NET 2.0 и был предназначен для упрощения работы с потоками в приложениях Windows Forms. Однако BackgroundWorker в той же мере применим и в WPF. Компонент BackgroundWorker предоставляет почти идеальный способ запуска длительно выполняющихся задач в отдельном потоке. Он использует диспетчер "за кулисами" и абстрагирует сложности маршализации с помощью модели событий.

По умолчанию код, вызываемый приложением, выполняется в основном потоке этого приложения. Таким образом, во время работы этого кода не выполняются никакие другие действия, в том числе и обновление интерфейса.

**BackgroundWorker** незаменим, если есть единственная асинхронная задача, которая выполняется в фоновом режиме от начала до конца (с необязательной поддержкой уведомлений о продвижении и возможностью отмены).

В классе BackgroundWorker реализованы три основных события: **ProgressChanged, RunWorkerCompleted и DoWork. При этом первые два выполняются в основном потоке программы и могут влиять на пользовательский интерфейс, а DoWork выполняется в отдельном потоке.**

**Создание BackgroundWorker**

Чтобы использовать BackgroundWorker, следует начать с создания его экземпляра. При этом на выбор доступны два подхода:

* Можно создать BackgroundWorker в коде и присоединить программно все обработчики событий.
* Можно объявить BackgroundWorker в XAML-разметке. Преимущество такого подхода в возможности присоединения обработчиков событий через атрибуты. Поскольку BackgroundWorker не является видимым элементом WPF, его нельзя поместить куда угодно. Вместо этого его понадобится объявить, как ресурс для окна.

**Выполнение работы**

1. Создадим новый проект WPF Application с именем «MainWindow».
2. Определим пользовательский интерфейс приложения.
3. Определим свойства добавленных элементов в соответствии с заданием.
4. Создадим новый экземпляр класса BackgroundWorker и определим для него обработчики событий **DoWork** и **RunWorkerCompleted**.
5. Определим обработчики событий **Click** для кнопок **Start** и **Canсel.**

namespace Window\_Main

{

public partial class MainWindow : Window

{

System.ComponentModel.BackgroundWorker aWorker = new System.ComponentModel.BackgroundWorker();

public MainWindow()

{

InitializeComponent();

aWorker.WorkerSupportsCancellation = true;

aWorker.DoWork += aWorker\_DoWork;

aWorker.RunWorkerCompleted += aWorker\_RunWorkerCompleted;

}

private delegate void UpdateDelegate(int i);

private void UpdateLabel(int i)

{

label1.Content = "Cycles: " + i.ToString();

}

private void aWorker\_DoWork(object sender, System.ComponentModel.DoWorkEventArgs e)

{

for (int i = 0; i <= 50; i++)

{

for (int j = 1; j <= 100000000; j++)

{

}

if (aWorker.CancellationPending)

{

e.Cancel = true;

return;

}

UpdateDelegate update = new UpdateDelegate(UpdateLabel);

label1.Dispatcher.BeginInvoke(

System.Windows.Threading.DispatcherPriority.Normal, update, i);

}

}

private void aWorker\_RunWorkerCompleted(object sender, System.ComponentModel.RunWorkerCompletedEventArgs e)

{

if (!(e.Cancelled))

label2.Content = "Run Completed";

else

label2.Content = "Run Cancelled";

}

private void buttonStart\_Click(object sender, RoutedEventArgs e)

{

aWorker.RunWorkerAsync();

}

private void buttonCancel\_Click(object sender, RoutedEventArgs e)

{

aWorker.CancelAsync();

}

}

Текст 3 - класс MainWindow. Определение компонента BackgroundWorker.

1. Построим и протестируем приложение.

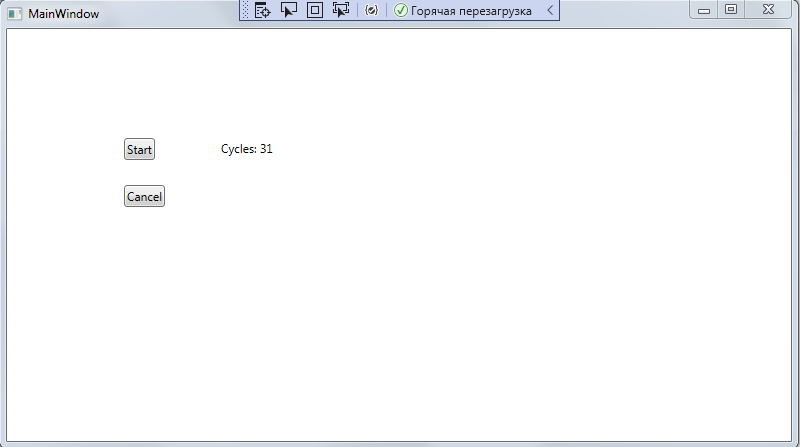


Рисунок 6 - Тестирование приложения.

**Практическое задание 6. Настройка интерфейса пользователя**

**Упражнение 1. Использование элементов Windows Forms**

**Теоретическая вставка**

Для того чтобы упростить совместимость между WPF и WinForms, разработчики Microsoft позаботились о том, чтобы элементы управления WinForms можно было использовать в приложениях WPF. Этот механизм реализован в качестве WindowsFormsHost.

**Выполнение работы**

1. Создадим новый проект WPF Application с именем «UserIn2».
2. Определим пользовательский интерфейс приложения.
3. Определим свойства добавленных элементов в соответствии с заданием.
4. В классе формы определим коллекцию номеров с именем **PhoneNumbers** и окно **SaveDialogBox** из Windows Forms.
5. Добавим обработчики события Click для кнопок «**Add To Collection**» и «**Save Collection To File**».

<myCtrls:ShowNumberControl x:Name="mSNC" CurrentNumber="100">

<myCtrls:ShowNumberControl.Triggers>

<EventTrigger RoutedEvent="myCtrls:ShowNumberControl.Loaded">

<BeginStoryboard>

<Storyboard TargetProperty="CurrentNumber">

<Int32Animation From="100" To="200"

Duration="0:0:10"></Int32Animation>

</Storyboard>

</BeginStoryboard>

</EventTrigger>

</myCtrls:ShowNumberControl.Triggers>

</myCtrls:ShowNumberControl>

Текст 4 - CurrentNumber.

1. Построим и протестируем приложение.

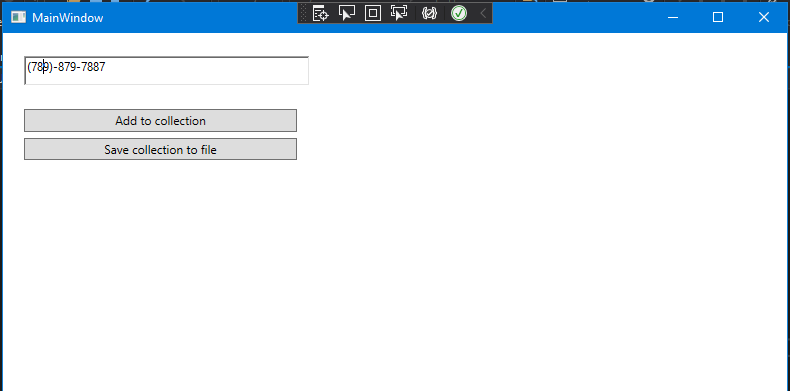


Рисунок 7 - проверка работы приложения

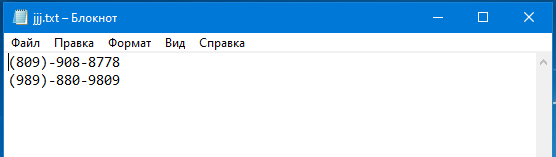


Рисунок 8 - проверка сохранения в файл

**Практическое задание 6. Настройка интерфейса пользователя**

**Упражнение 2. Построение специального свойства зависимости**

1. Создадим новый проект WPF Application с именем «WpfUserControl».
2. Добавим в проект пользовательский элемент управления с именем **ShowNumberControl.**
3. Отредактируем XAML файл, связанный с созданным элементом управления в соответствии с условием задания, добавив в **Grid** элемент управления **Label**.
4. Отредактируем файл кода данного элемента определив параметр **CurrentNumber** встроенным способом.
5. В коде разметки XAML окна **MainWindow** объявим экземпляр специального элемента управления внутри диспетчера компоновки и определим специальное пространство имен XML, отображаемое на данный элемент.
6. Добавим код для применения анимации для компонента.
7. Реализуем статический метод для проверки достоверности данных с именем **ValidateCurrentNumber** и функционалом, описанным в условии задания, и зададим его как финальный аргумент в метод DependencyProperty.Register.
8. Реализуем статический метод для реагирования на изменение свойства с именем **CurrentNumberChanged** и функционалом, описанным в условии задания, и зададим его как второй аргумент в конструктор UIPropertyMetadata.

namespace WpfUserControl

{

/// <summary>

/// Логика взаимодействия для ShowNumberControl.xaml

/// </summary>

public partial class ShowNumberControl : UserControl

{

public ShowNumberControl()

{

InitializeComponent();

}

public static bool ValidateCurrentNumber(object value)

{

if (Convert.ToInt32(value) >= 0 && Convert.ToInt32(value) <= 500)

return true;

else

return false;

}

public int CurrentNumber

{

get { return (int)GetValue(CurrentNumberProperty); }

set {

SetValue(CurrentNumberProperty, value);

}

}

private static void CurrentNumberChanged(DependencyObject depObj, DependencyPropertyChangedEventArgs args)

{

ShowNumberControl s = (ShowNumberControl)depObj;

Label theLabel = s.numberDisplay;

theLabel.Content = args.NewValue.ToString();

}

// Using a DependencyProperty as the backing store for CurrentNumber. This enables animation, styling, binding, etc...

public static readonly DependencyProperty CurrentNumberProperty =

DependencyProperty.Register("CurrentNumber", typeof(int), typeof(ShowNumberControl),

new UIPropertyMetadata(100, new PropertyChangedCallback(CurrentNumberChanged)), new ValidateValueCallback(ValidateCurrentNumber));

}

Текст 5 – классы в namespace WpfUserControl

1. Построим и протестируем приложение.

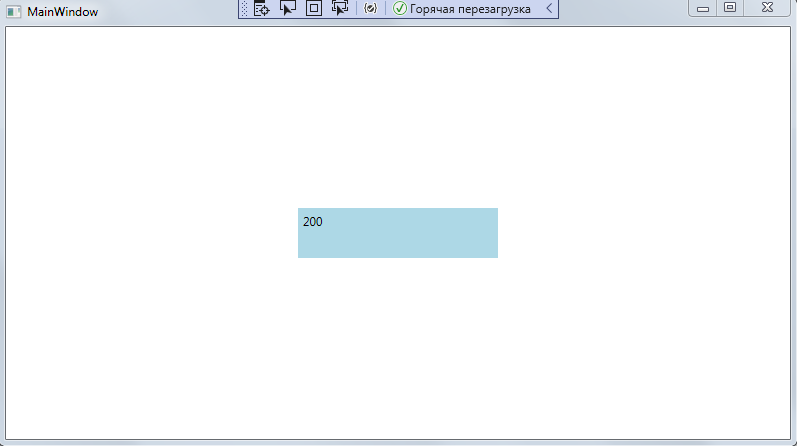


Рисунок 9 - тестирование приложения

Как и ожидалось происходит анимация контента компонента от 100 до 20

**Практическое задание 7. Графика в приложениях WPF**

**Упражнение 1. Рисование замкнутой фигуры с помощью элемента "Многоугольник"**

1. Создадим новый проект WPF Application с именем «WpfUserControl».
2. Отредактируем файл MainWindow.xaml в соответствии с условием задания, для отображения четырех треугольников, а также двух звезд.
3. Построим и протестируем приложение.

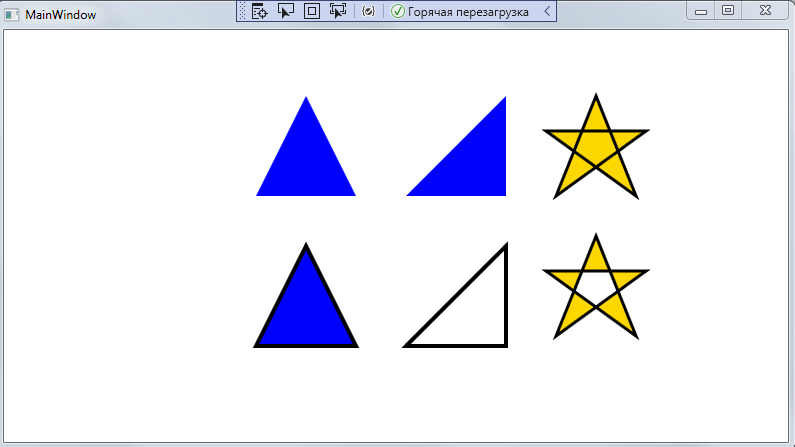


Рисунок 10 - тестирование приложения

**Практическое задание 7. Графика в приложениях WPF**

**Упражнение 2. Создание приложения рисования**

1. Создадим новый проект WPF Application с именем «**InkCanvas**».
2. Определим пользовательский интерфейс приложения. В том числе добавим элемент **InkCanvas**.
3. Определим свойства добавленных элементов в соответствии с заданием.
4. Создадим обработчики событий Click для элементов Button (кнопки «**Close**» и «**Clear**»).
5. Построим и протестируем приложение.

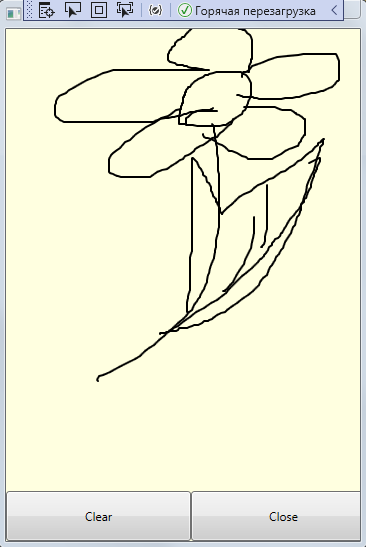


Рисунок 11 - Тестирование приложения

**Практическое задание 7. Графика в приложениях WPF**

**Упражнение 3. Создание диаграмм с использованием WPF Toolkit**

1. Создадим новый проект WPF Application с именем «WpfApplicationChart».
2. Определим зависимости System.Windows.Controls.DataVisualization.Toolkit.dll и WPFToolkit.dll.
3. Импортирует необходимые имена и пространства имен в файле WindowChartSample.xaml.
4. Добавим в элемент Grid диаграмму (элемент **chartingToolkit:Chart**).
5. Определим свойства элементов интерфейса в соответствии с заданием.
6. Добавим в проект класс **Model** и отредактируем в соответствии с заданием, определив класс **Data** и реализовав хранение данных в списке.
7. Добавим секцию **Window.Resources** в файл **Window.xaml** и свяжем их с моделью.

namespace WpfApplicationChart

{

public class Data

{

public Data(string name, int value)

{

Name = name;

Value = value;

}

public string Name { get; set; }

public int Value { get; set; }

}

class Model

{

public IList<Data> Data

{

get

{

IList<Data> list = new List<Data>

{

new Data("Иванов", 10),

new Data("Петров", 30),

new Data("Сиборов", 10),

new Data("Кузнецов", 20),

new Data("Павлов", 40)

};

return list;

}

}

}

1. Построим и протестируем приложение.

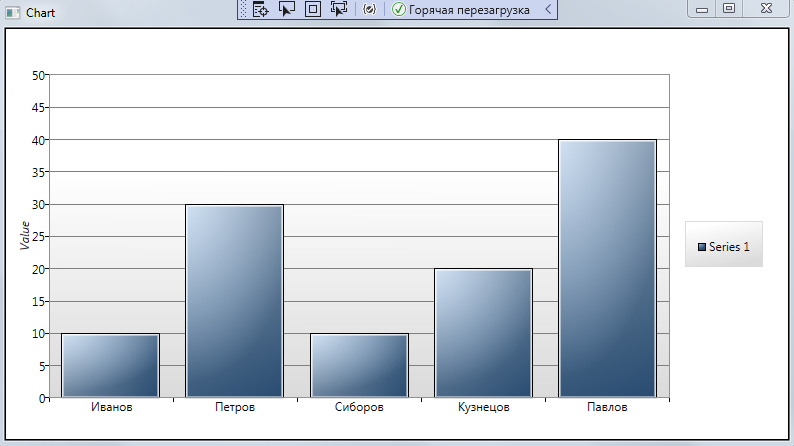


Рисунок 12 - Тестирование приложения

**Вывод**

В данной лабораторной работе были рассмотрены:

* Привязка компонента **Label** к полям элементам пользовательского интерфейса для быстрой навигации.
* Элементы пользовательского интерфейса **ComboBox** и **Slider**.
* Работа с контейнером **DockPanel**
* Работа со свойствами компонента **RichTextBox.**
* Работа с элементом **BackgroundWorker,** запуск задания в отдельном потоке.
* Встраивание компонентов WindowsForms посредством элемента **WindowsFormsHost**.
* Создание пользовательского элемента управления.
* Работа с графикой.
* Работа с диаграммами, посредством подключения сторонней библиотеки.

WPF предлагает разработчику обширный набор возможностей, среди которых:

Возможность декларативного определения графического интерфейса с помощью специального языка разметки XAML, основанном на xml и представляющем альтернативу программному созданию графики и элементов управления, а также возможность комбинировать XAML и C#/VB.NET

Независимость от разрешения экрана: поскольку в WPF все элементы измеряются в независимых от устройства единицах, приложения на WPF легко масштабируются под разные экраны с разным разрешением.

Новые возможности, которых сложно было достичь в WinForms, например, создание трехмерных моделей, привязка данных, использование таких элементов, как стили, шаблоны, темы и др.

Хорошее взаимодействие с WinForms, благодаря чему, например, в приложениях WPF можно использовать традиционные элементы управления из WinForms.

Богатые возможности по созданию различных приложений: это и мультимедиа, и двухмерная графика, и богатый набор встроенных элементов управления, а также возможность самим создавать новые элементы, создание анимаций, привязка данных, стили, шаблоны, темы и многое другое

Создание приложений под множество ОС семейства Windows - от Windows XP до Windows 10